Synergistic Graph and SQL Analytics Inside IBM Db2

Yuanyuan Tian
IBM Research
ytian@us.ibm.com,

Sui Jun Tong
IBM Research
bjsjtong@cn.ibm.com

Mir Hamid Pirahesh
IBM Research
pirahesh@us.ibm.com

Wen Sun
IBM Research
sunwenbj@cn.ibm.com

En Liang Xu
IBM Research
bjxelxu@cn.ibm.com

Wei Zhao
IBM Research
weizhao@cn.ibm.com

ABSTRACT

To meet the challenge of analyzing rapidly growing graph and network data created by modern applications, a large number of specialized graph databases have emerged, such as Neo4j, JanusGraph, and Sqlg. At the same time, RDBMSs and SQL continue to support mission-critical business analytics. However, real-life analytical applications seldom contain only one type of analytics. They are often made of *heterogeneous* workloads, including SQL, machine learning, graph, and other analytics. In particular, SQL and graph analytics are usually accompanied together in one analytical workload. This means that graph and SQL analytics need to be *synergistic* with each other. Unfortunately, most existing graph databases are standalone and cannot easily integrate with relational databases. In addition, as a matter of fact, many graph data (data about relationships between objects or people) are already prevalent in relational databases, although they are not explicitly stored as graphs. Performing graph analytics on these *relational* graph data today requires exporting large amount of data to the specialized graph databases. A natural question arises: can SQL and graph analytics be performed synergistically in a same system? In this demo, we present such a working system called IBM Db2 Graph. Db2 Graph is an in-DBMS graph query approach. It is implemented as a layer inside an experimental IBM Db2™, and thus can support synergistic graph and SQL analytics efficiently. Db2 Graph employs a *graph overlay* approach to expose a *graph view* of the relational data. This approach flexibly retrofits graph queries to existing graph data stored in relational tables. We use an example scenario on health insurance claim analysis to demonstrate how Db2 Graph is used to support synergistic graph and SQL analytics inside Db2.

PVLDB Reference Format:
Yuanyuan Tian, Sui Jun Tong, Mir Hamid Pirahesh, Wen Sun, En Liang Xu, Wei Zhao. Synergistic Graph and SQL Analytics Inside IBM Db2. *PVLDB*, 12(12): 1782-1785, 2019.
DOI: https://doi.org/10.14778/3352063.3352065

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352065

1. INTRODUCTION

Rapidly growing social networks and other graph datasets have created a high demand for graph analysis systems. As a result, a large number of graph databases emerged, focusing on the low-latency graph queries, such as finding the neighbors of a vertex with certain properties, and retrieving the shortest path between two vertices. Examples of graph databases include Neo4j [9], JanusGraph [8], Sqlg [13], SQLGraph [16], OrientDB [10], Sparksee [12], ArangoDB [2], InfiniteGraph [7], BlazeGraph [3], TigerGraph [14], and SQL Server's Graph Extension [4], etc.

These graph databases generally handle graph-only query workload very well. However, graph queries are *not all* that one does in an analytics workload of a real life application. They are often only a part of an integrated *heterogeneous* analytics pipeline, which may include SQL, machine learning (ML), graph and other types of analytics. After all, SQL analytics still remain to be the most widely used for mission-critical business analytics. As a result, graph queries are often combined with SQL analytics in practice. And perhaps more importantly, many graph data are already prevalent in the existing relational databases. Sometimes this is due to legacy reasons. For example, many graph data and graph queries have already existed before the boom of graph databases. In addition, very often, the same data which powered the existing SQL applications can also be treated as graph data (e.g. data about relationships between objects or people) and be used for new graph applications. In summary, graph queries need to be *synergistic* with SQL analytics and *retrofittable* to existing relational data.

Unfortunately existing graph databases cannot satisfy the synergistic and retrofittable requirements. First of all, most of them are *standalone* and cannot easily integrate with relational databases. They force applications to import data into the specialized graph databases either at runtime or in a preprocessing step, perform the graph queries, and export the result data to continue with the rest of the analytics pipeline. Even though some *hybrid* graph databases (which build a graph engine on top of an existing data store) are built on top of relational databases, such as Sqlg [13], SQLGraph [16], and SQL Server Graph Extension [4], they dictate the way graph data are stored inside the database, hence cannot retrofit to existing relational graph data. The existing graph data have to be replicated in the desired form before graph queries can be applied in these systems, and of course, the applications have to make sure that the two copies of the data are consistent in case of updates. GR-Fusion [15] and GraphGen [17] adopted the approach of ex-

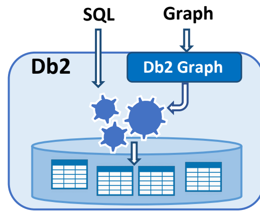


Figure 1: Overview of Db2 Graph

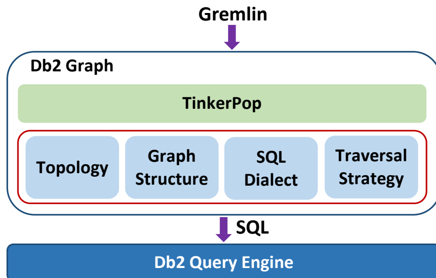


Figure 2: Db2 Graph architecture

tracting graphs from the relational tables, and then materializing the graph structures in memory. Graph queries are only served on the in-memory copy of the data. Essentially, this approach also keeps two copies of data, only that the secondary copy is in memory. As a result, when updates happen to the underlying relational data, the graph queries won't see the latest data.

In this demo, we propose to demonstrate an in-DBMS graph query approach, called IBM Db2 Graph, for supporting synergistic and retrofittable graph queries inside the IBM Db2™relational database [6]. As illustrated in Figure 1, Db2 Graph is implemented as a layer inside an experimental Db2 prototype, and thus can support synergistic graph and SQL analytics efficiently. Most importantly, graph and SQL queries operate on exactly the *same data* stored in the database. Db2 Graph employs a *graph overlay* approach to expose a *graph view* of the relational data. This approach flexibly retrofits to existing graph data stored in relational tables. We use an example scenario on health insurance claim analysis to demonstrate how Db2 Graph can support synergistic graph and SQL analytics inside Db2.

2. Db2 Graph OVERVIEW

Db2 Graph is a layer inside Db2 specialized for graph queries. It overlays a *property graph* onto a set of relational tables. Users then can issue graph queries written in Tinkerpop Gremlin graph language [1] on top of the overlay graph. Db2 Graph translates each graph query into a set of SQL queries, and finally executes the graph query by utilizing the Db2 query engine through SQL.

2.1 Graph Overlay

We now describe how Db2 Graph overlay a property graph onto relational tables.

Background on Property Graphs. A property graph contains vertices and edges. Vertices represent discrete objects, and edges capture the relationships between vertices. Both vertices and edges can have arbitrary number of *properties*, represented as key-value pairs. Each vertex/edge is uniquely identified with an *id* in a property graph. Vertices/edges can also be tagged with *labels* to distinguish the different types of objects/relationships in the graph.

Graph databases always present a *single* property graph with a vertex set and an edge set for users to query. Overlaying a single property graph onto a set of relational tables really boils down to mapping the vertex set and the edge set of the graph to the relational tables. In particular, for the vertex set, the mapping specifies: 1) what table(s) store the vertex information, 2) what table column(s) are mapped to the required *id* field, 3) what is the label for each vertex (defined from a table column or a constant), and 4) what columns capture the vertex properties, if any. Similarly, for the edge set, the mapping specifies: 1) what table(s) store the edge information, 2) what table columns are mapped to the required *id*, *inv* (incoming vertex id), and *outv* (outgoing vertex id) fields, 3) what is the label for each edge (defined from a table column or a constant), and 4) what columns correspond to the edge properties, if any. This graph overlay mapping is achieved by an *overlay configuration file* in Db2 Graph. Note that the mapping is not restricted to tables only, it can be also on created views of tables.

The overlay configuration files can be manually created by the application developers who wish to query relational tables as graphs. Db2 Graph also provides a toolkit, called AutoOverlay, to automate the generation of overlay configuration for a database in a principled way. AutoOverlay relies on the primary and foreign key constraints to infer relationships among the data in relational tables.

2.2 System Architecture

Figure 2 shows the overall architecture of Db2 Graph. It is a layer inside Db2, between Gremlin graph queries and the Db2 query engine. Db2 Graph includes the TinkerPop stack on the top, which parses the input Gremlin and generates query plans with Tinkerpop API calls, and four tightly connected components at the bottom, which together implement the TinkerPop core API with some optimized traversal strategies. The Topology component maintains the overlay mapping between a property graph and the relational tables; the Graph Structure component is our implementation of the TinkerPop graph structure API; the SQL Dialect component deals with implementation details specific for Db2; and the Traversal Strategy component contains optimized traversal strategies for performance improvement. Together they execute the Gremlin query plans with the help of the Db2 query engine through SQL queries. Below, we highlight two important optimizations.

Optimization for fixed labels. One common operation in graph traversals is to query vertices/edges by label(s), e.g. `g.V().hasLabel('person')`. When the vertex/edge set maps to multiple tables, by default the implementation has to query all the vertex/edge tables with the label predicate. But, when vertex/edge tables have fixed labels ((i.e. `fixed_label = true`) in the overlay configuration, we can use the specified label(s) to narrow down a subset of tables to query from. More specifically, any table that has a fixed label but not matched with the query label(s) can be eliminated from the query. This optimization will provide a huge performance improvement.

Optimized traversal strategies. A graph traversal in Gremlin is expressed in a number of steps, e.g. `g.V() .has('name','Alice').values('age', 'address')`. By de-

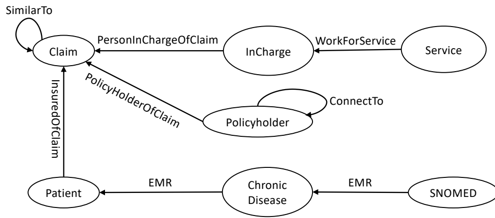


Figure 3: Relationships among data entities.

fault, the steps are executed one by one. In this naive approach, SQL queries are independently translated from individual steps without considering their neighbor steps. This means a query like `SELECT * FROM VertexTable` (assuming there is only one vertex table named VertexTable) would be executed first (corresponding to `g.V()`) to retrieves *all* the vertex data into Db2 Graph, although the second step can filter out most of the vertices. In Db2 Graph, we add optimized strategies in the Traversal Strategy component to combine multiple steps into one composite step and to translate the composite step into an optimized SQL query. We start from a step that accesses the basic graph structure information requiring a SQL query to Db2, like `g.V()` or `v.outE()`, and try to fold subsequent filter (predicate), property (projection), group, or aggregation steps into the previous step as much as possible. The operations of the folded steps are pushed down to the SQL query of the original step. The above example query can now be translated into an optimized SQL: `SELECT age, address FROM VertexTable WHERE name = 'Alice'`.

3. DEMONSTRATION DESCRIPTION

Example Scenario. We have two datasets stored in a Db2 database: a health insurance claim dataset that contains information about insurance claims, policyholders, persons in charge, and service providers, etc.; and an EMR (Electronic Medical Record) dataset that contains information about patients, chronic diseases, and SNOMED disease ontology [11], etc. The goal of the anaysis is to detect and investigate fraudulent insurance claims, by querying the datasets together using both SQL and graph queries. When using SQL, we view the datasets as a set of relational tables. However, using graph overlay, we can also view all the data from the two datasets as a giant graph that links entities together. Figure 3 shows how the data entities are related to each other in these two datasets. For example, an insurance claim is linked to an insured person, also called a patient in the EMR setting, who is then connected to diseases, which in turn link to the disease descriptions in SNOMED; a claim is also linked to a policyholder, who may further connect to other policyholders through social networking or association; a claim also involves a person in charge (e.g. a physician), who then is associated with a service provider (e.g. a hospital); claims are also connected to each other by their similarities, which are resulted from running a machine learning algorithm for detecting claim similarities. In our demo, we will conduct the following 4 steps to detect potential fraudulent insurance claims.

3.1 Find Out Suspicious Claims

In this step, our task is to find out all suspicious insurance claims in the datasets and sort them by their severity. A claim is considered suspicious if its charge is more than 4x the average charge of similar claims. We can use the following complex SQL statements to accomplish this task. SQL is the best fit for this, since the analysis doesn't require navigating through numerous entities, but needs to compute and sort various statistics. Heavy lifting group-by, aggregation, and sorting is hard to do in Gremlin that focuses more on traversal along the graph structures.

```
SELECT claimid, charge,
  DECIMAL(simAvgCharge, 10, 2) AS simAvgCharge,
  simCount, simMinCharge, simMaxCharge
FROM Claims AS ThisClaim,
  LATERAL (
    SELECT AVG(charge) AS simAvgCharge,
      COUNT(*) AS simCount, MIN(charge) AS simMinCharge,
      MAX(charge) AS simMaxCharge
    FROM Claim_Similarity AS Sim, Claims AS OtherClaims
    WHERE ThisClaim.claimid = Sim.claimid
      AND Sim.simClaimid = OtherClaims.claimid
  ) AS SimClaims
WHERE charge > float(4)* simAvgCharge
GROUP BY claimid, charge, simAvgCharge, simCount,
  simMinCharge, simMaxCharge
ORDER BY charge/simAvgCharge DESC;
```

3.2 Look Into One Suspicious Claim

After suspicious claims are identified, Gremlin graph queries can then be used to shed light on why a claim is suspicious by exploring all related data associated with the claim. For example, we can find out what diseases the insured of the claim have, to figure out why the claim incurs exceptionally high (more than 4x the average) charge. The following Gremlin query can be used to accomplish this task. It traverses from a claim node with ID 'C4377' to the insured of the claim (also a patient), then to the diseases that he/she has, and finally the SNOMED description of the diseases. Although SQL can also produce the same result, but it requires joins on 4 tables. In comparison, the graph query is not only more concise but also much more intuitive.

```
g.V('C4377').hasLabel('Claim').out('InsuredOfClaim')
  .out('EMR').hasLabel('ChronicDisease').out('EMR')
  .hasLabel('SNOMED').values('CONCEPT_NAME')
```

The results can be further visualized to facilitate interactive analysis, as shown in Figure 4. This figure shows that the insured (yellow node) of the claim (orange node) suffers from multiple chronic diseases (dark blue nodes). That's probably why the charge of the claim is so much higher than normal.

3.3 Investigate The Policyholder

Now, let's investigate the policyholder of the above suspicious claim. What other claims were filed by the same policyholder? Again, we can use a graph query to perform this analysis, as shown in Figure 5. The yellow node is the policyholder. It's connected to a number of claims, shown as the dark blue nodes. And all the claims link to the same patient as the insured. What's more, in each claim, the patient was treated by a different physician from a different service provider. It's possible that the insured goes to different providers to get the same service repeatedly. Since different providers don't usually share data, they would not know this person has already got the same service elsewhere. This is very suspicious behavior.

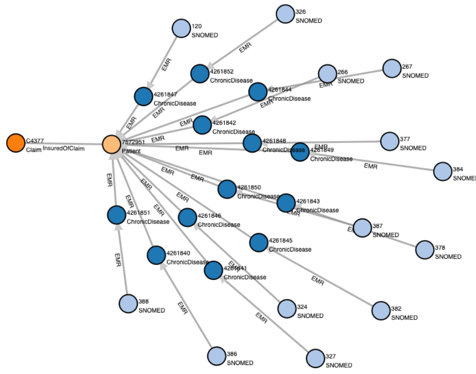


Figure 4: Diseases of the insured.

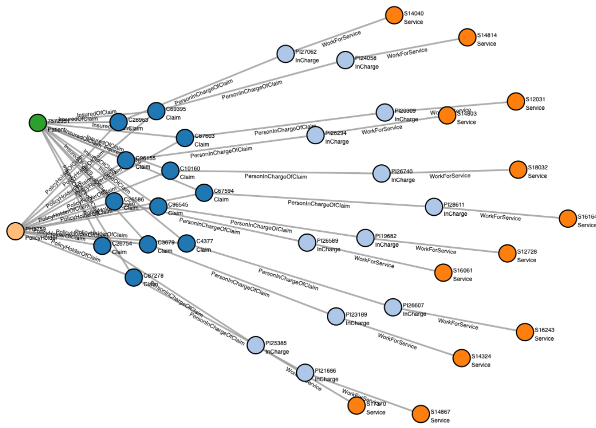
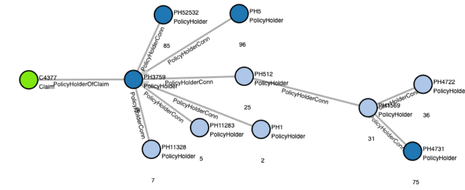


Figure 5: Claims filed by the policyholder.

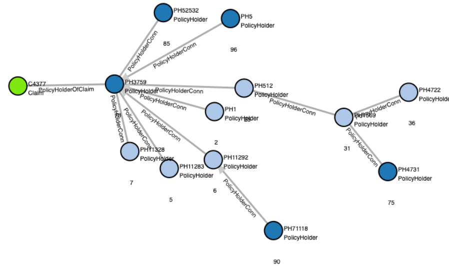
3.4 Examine Connections of Policyholders

Now, we examine the social connections of the above policyholder. As shown in Figure 6(a), we can use a graph query to find out how other policyholders are directly or indirectly connected to the policyholder in question. The green node is the suspicious claim, and the dark blue node connected to it is the policyholder in question. The dark blue nodes are all high-risk policyholders, and the light blue nodes are low-risk policyholders. The risk scores for policyholders are also shown next to the nodes. These scores are computed using a machine learning algorithm ahead of time. As shown in the figure, the policyholder in question is directly connected to some high-risk policyholders, but also some low-risk policyholders. He is also indirectly connected to another high-risk policyholder by 3 degrees of separation. Seeing how the policyholder in question connections to other high-risk policyholders further reinforces the likelihood that he/she is involved in some fraudulent activities, and also potentially sheds light on who-else might be involved in the potential fraud.

Finally, we demonstrate how updates to the underlying database can be reflected in graph queries in real time. In Db2 Graph, SQL and graph queries operate on the same data, thus any update from the relational side (e.g. through transactions) is immediately query-able from the graph side. In this example, we add two more policyholder connections to the database, and re-run the last graph query. The two added connections are shown in the graph query result, as in Figure 6(b). Now the policyholder in question is indirectly connected to one more high-risk policyholder by 2 degrees of separation.



(a) Before Updates



(b) After Updates

Figure 6: Social connections of the policyholder.

4. CONCLUSION

In this proposal, we demonstrated how to use IBM Db2 Graph to perform synergistic graph and SQL queries on the same relational data inside IBM Db2™, in the context of an insurance claim analysis. As we have shown, each type of analytics has its own strength: SQL is very good at heavy lifting grouping, aggregation, and sorting, whereas graph excels at exploitative navigation of relationships. In practice, they are often mixed together in an integrated analytics pipeline as illustrated in this demo. The in-DBMS graph query approach employed by Db2 Graph really makes the synergistic graph+SQL analytics workload much easier and more efficient.

5. REFERENCES

[1] Apache TinkerPop. `http://http://tinkerpop.apache.org`.

[2] ArangoDB. `https://www.arangodb.com`.

[3] BlazeGraph. `https://www.blazegraph.com`.

[4] Graph processing with SQL Server and Azure SQL Database. https://docs.microsoft.com/en-us/sql/relational-databases/graphs/sql-graph-overview?view=sql-server-2017.

[5] Graphexp: graph explorer with D3.js. `https://github.com/bricaud/graphexp`.

[6] IBM Db2. `https://www.ibm.com/analytics/us/en/db2`.

[7] InfiniteGraph. `https://www.objectivity.com/products/infinitegraph`.

[8] JanusGraph. `http://janusgraph.org`.

[9] Neo4j. `https://neo4j.com`.

[10] OrientDB. `https://orientdb.com`.

[11] SNOMED International. `http://www.snomed.org`.

[12] Sparksee. `http://www.sparsity-technologies.com`.

[13] Sqlg. `http://www.sqlg.org`.

[14] TigerGraph. `https://www.tigergraph.com`.

[15] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi. Grfusion: Graphs as first-class citizens in main-memory relational database systems. In *SIGMOD '18*, pages 1789–1792, 2018.

[16] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *SIGMOD '15*, pages 1887–1901, 2015.

[17] K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *SIGMOD '17*, pages 897–912, 2017.