IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2

Yuanyuan Tian
IBM Research

En Liang Xu
IBM Research

Wei Zhao
IBM Research

Mir Hamid Pirahesh
IBM Research

Sui Jun Tong
IBM Research

Wen Sun
IBM Research

Thomas Kolanko
IBM Cloud and Cognitive Software

Md. Shahidul Haque Apu
IBM Cloud and Cognitive Software

Huijuan Peng
IBM Cloud and Cognitive Software

ABSTRACT

To meet the challenge of analyzing rapidly growing graph and network data created by modern applications, a large number of graph databases have emerged, such as Neo4j and JanusGraph. They mainly target low-latency graph queries, such as finding the neighbors of a vertex with certain properties, and retrieving the shortest path between two vertices. Although many of the graph databases handle the graph-only queries very well, they fall short for real life applications involving graph analysis. This is because 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, graph, and other analytics. This means graph queries need to be *synergistic* with other analytics. Unfortunately, most existing graph databases are standalone and cannot easily integrate with other analytics systems. In addition, many graph data (data about relationships between objects or people) are already prevalent in existing non-graph databases, although they are not explicitly stored as graphs. None of existing graph databases can *retrofit* graph queries onto these existing data without transferring or transforming data. In this paper, we propose an in-DBMS graph query approach, IBM Db2 Graph, to support synergistic and retrofittable graph queries inside the IBM Db2™relational database. It is implemented as a layer inside Db2, and thus can support integrated graph and SQL analytics efficiently. Db2 Graph employs a novel *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, without expensive data transfer or transformation. In addition, it enables efficient execution of graph queries with the help of Db2 relational engine, through sophisticated compile-time and runtime optimization strategies. Our experimental study, as well as our experience with real customers using Db2 Graph, showed that Db2 Graph can provide very competitive and sometimes even better performance on graph-only queries, compared to existing graph databases. Moreover, it optimizes the overall performance of complex analytics workloads.

ACM Reference Format:
Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3386138

1 INTRODUCTION

Rapidly growing social networks and other graph datasets have created a high demand for graph analytics systems. Graph analytics systems can be generally categorized into two types: graph processing systems and graph databases. The former, graph processing systems, such as Giraph [1], GraphLab [35], and GraphX [29], focus on batch processing of large graphs, e.g. running PageRank on a graph. This type of analysis is usually iterative and long running. In contrast, the latter, graph databases, such as Neo4j [12] and JanusGraph [10], focus more on the relatively low-latency graph queries, such as finding the neighbors of a vertex with certain properties, and retrieving the shortest path between two vertices. In this paper, we focus on the low-latency graph queries in the latter case.

There are a large number of graph databases, including Neo4j [12], JanusGraph [10], Sqlg [20], SQLGraph [41], OrientDB [15], TigerGraph [22], Sparksee [19], ArangoDB [3], InfiniteGraph [9], BlazeGraph [4], GraphGen [44], GRFusion [30], Oracle Spatial and Graph [14], and SQL Server's Graph extension [6], etc. They can be roughly divided into two camps. The first camp is *native* graph databases, with specialized query and storage engines built from scratch just for graphs. Neo4j, OrientDB, TigerGraph, and Sparksee all belong to this camp. The second camp of graph databases, including JanusGraph, Sqlg, SQLGraph, ArangoDB, InfiniteGraph, BlazeGraph, GraphGen, GRFusion, Oracle Spatial and Graph, and SQL Sever Graph extension, all delegate their storage engines to existing data stores, be it either a SQL database (e.g. Sqlg, SQLGraph, InfiniteGraph, GraphGen, GRFusion, Oracle Spatial and Graph, and SQL Server Graph extension), a key-value store (e.g. JanusGraph), a document store (e.g. ArangoDB), or an RDF store (e.g. BlazeGraph). We call them *hybrid* graph databases. A hybrid graph database builds a specialized graph query engine, but resorts to an existing data store to handle the persistence of data.

Both camps of graph databases generally handle graph-only query workload very well. But they are still *inadequate* for real life applications involving graph analysis. This is because 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. Of course, given the power and prevalence of SQL, graph queries are often combined with SQL queries in practice. In addition, graph data itself can be used in SQL, ML or other types of analytics as well. For example, the properties of graph vertices can be queried in SQL analytics or be used as the features for training an ML model. And perhaps more importantly, many graph data are already prevalent in the existing non-graph stores. 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 non-graph applications can also be treated as graph data (e.g. data about relationships between objects or people) and be used for new graph applications. Another reason behind the prevalence of graph data in non-graph stores is that the applications made conscious decisions to manage graph queries themselves instead of using specialized graph databases to achieve the good performance of the overall applications with different analytics workloads. In summary, graph queries need to be *synergistic* with other analytics and *retrofittable* to existing data.

Unfortunately existing graph databases cannot satisfy the synergistic and retrofittable requirements. Native graph databases force applications to import data into the proprietary storage engine 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. Data import and export can incur a lot of overhead. In contrast, storing graph data in the same storage engine as the data for other analytics can help eliminate the data transfer overhead. This could be a potential benefit for the hybrid graph databases. However, unfortunately, almost all of the hybrid graph databases *dictate* the schema of storing graphs in the storage engines. Especially, when SQL databases and key-value stores are used as the storage engines, the schema is so *convoluted* and *unnatural* that by just querying the stored data using the query engines of the underlying data stores, it is almost impossible to get any meaningful result out. For example, SQLGraph "shreds" the vertex adjacency lists using a hash function into a number of reserved columns in a table, and spills into a second table when hash collision happens. JanusGraph stores the entire adjacency list of a vertex in a somewhat *encrypted* form in one column. Although the graph data is stored in a SQL database or a key-value store, the convoluted schema makes it impossible to *decipher* what is stored, and thus makes the graph data unusable for any analytics directly using the query engines of underlying data stores. This means that the application has to replicate the same data and transform them using a more natural schema, if it wants to access the graph data in other analytics, and of course goes through the trouble to make sure that the two copies are consistent in case of updates. Few hybrid graph databases, like Sqlg, do use a more natural schema to store graph data in a SQL database, thus making the share of graph data among different analytics possible. However, the schema is also *dictated*, so it is impossible to retrofit to existing graph data that are not stored according to the rigid schema. Microsoft SQL Server Graph extension extended SQL with some graph querying capabilities. However, it requires the explicit creation of vertex and edge tables before they can be used for graph queries. Moreover, certain *implicit* columns *have to* be present in the vertex and edge tables. As a result, it cannot retrofit to existing relational data. Oracle Spatial and Graph, GRFusion and GraphGen can support graph queries on existing relational data. However, all three adopted the approach of loading or extracting 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 a second, *transformed*, copy of the 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 paper, we propose a *true* in-DBMS graph query approach, IBM Db2 Graph, to support synergistic and retrofittable

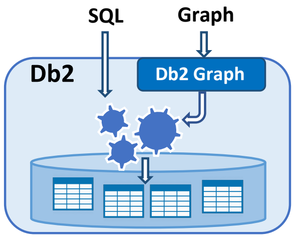


Figure 1: Synergistic graph queries inside Db2

graph queries inside IBM Db2™ [8]. Our goal is not to build the fastest graph database in the world, but rather to build graph query support inside Db2 that is synergistic with other analytics and retrofittable to existing data. On graph-only queries, it may not be the most efficient, but by avoiding the overhead of transferring and transforming data, it provides the best overall performance for complex analytics workloads in the real world.

Db2 Graph is a layer inside Db2 specialized for graph queries, as illustrated in Figure 1. It takes in a graph query as the input, and executes the graph query by utilizing the Db2 query engine through SQL. Most importantly, graph and SQL queries operate on exactly the *same data* stored in the database with a natural relational schema. Db2 Graph employs a *graph overlay* approach to expose a *graph view* of the relational data. This approach enables graph query capabilities on existing relational data, and even supports different vertex and edge types with various sets of properties in the same graph. In addition, Db2 Graph aggressively applies various compile-time and runtime optimization strategies to efficiently execute graph queries utilizing the Db2 relational engine. Db2 Graph supports the popular TinkerPop Gremlin [2] as the graph query language, since it covers a broad set of graph operations and is widely adopted in the industry. Note that our in-DBMS graph query approach can be easily generalized to other graph query languages and other databases.

Db2 Graph is designed to be a practical graph database solution for real applications. Therefore, besides query performance, synergy with other analytics, and retrofittability to existing data, other practical issues, such as transaction support, access control, and compliance to audits and regulations, are equally important for Db2 Graph. Luckily, by riding on the mature and robust Db2 technology, many of these features come for free. Transaction support has been traditionally a sore spot for graph databases: most existing graph databases either have no support or very weak support for transactions. In contrast, transaction support has been the strongest suit for RDBMSs. By embedding itself inside Db2, Db2 Graph relies on the powerful transaction support in Db2 to handle graph updates. Since SQL and graph share the same data underneath, any update to the relational tables from the transactional side is immediately available to the graph queries. Access control is another weak point for many existing graph databases. Again, as no secondary copy of data (either on disk or in memory) is created in Db2 Graph, Db2 Graph directly inherits Db2's mature access control mechanisms. Finally, Db2 also brings in the bi-temporal support (i.e. supporting both system time and business time) for free. This feature is crucial for compliance to audits and regulations (e.g. GDPR), but unfortunately is missing in most existing graph databases.

This paper is organized as follows. Section 2 reviews related works. Section 3 provides technical background on property graphs and Gremlin. Section 4 presents an example scenario used throughout this paper. Section 5 discusses the graph overlay approach. Section 6 describes the system architecture and implementation. Section 7 then presents the real world usage of Db2 Graph. We report empirical studies in Section 8 and finally conclude in Section 9.

2 RELATED WORK

Graph Query Languages. Despite the numerous graph databases on the market, there is no standard graph query language. However, most of the existing graph databases adopt Tinkerpop Gremlin [2]. Cypher [28] is a declarative graph query language introduced by Neo4j. Oracle proposed another declarative language based on SQL, called PGQL [18]. GSQL [7] is the SQL-like graph query language adopted by TigerGraph. The LDBC [21] Graph Query Language Task Force has proposed G-Core [24]. Finally, Graph Query Language (GQL) [5] is a recent ongoing effort towards a standard graph query language, which builds on SQL and integrates ideas from Cypher, PGQL, GSQL, and G-CORE.

Graph Databases. The need to support graph queries has led to a plural of graph databases. Comparisons on various graph databases can be found in [26, 32–34]. Neo4j [12], OrientDB [15], TigerGraph [22], and Sparksee [19] are examples of native graph databases, among which Neo4j is the most popular. Neo4j employs an index-free adjacency technique, which stores each vertex together with its adjacent vertices and edges to get good performance on data retrieval at runtime. To further improve performance, Neo4j also heavily caches graph data in memory. Examples of hybrid graph databases include JanusGraph [10], Sqlg [20], SQLGraph [41], ArangoDB [3], InfiniteGraph [9], BlazeGraph [4], GraphGen [44], GRFusion [30], Oracle Spatial and Graph [14], and SQL Server's Graph extension [6].

Although SQLGraph bases on the same back-end database, IBM Db2 [8], as our Db2 Graph, it focuses on graph-only queries. To get the best performance on graph queries, it "shreds" the vertex adjacency lists using a hash function into a number of reserved columns in a table, and spills into a second table when hash collision happens. It is impossible for the shredded graph data to be used in normal SQL analytics.

Similar to Db2 Graph, Sqlg stores graph data in normal schema, but it dictates the schema. Therefore, it cannot retrofit to existing relational data. At loading time, Sqlg analyzes all the data and decides on how to store the graph.

GraphGen *extracts* graphs from relational data via a Datalog-based DSL and stores a condensed in-memory representation of the extracted graphs. Graph queries and analytics are then carried out on the in-memory representation. GRFusion [30] extends VoltDB [23] to define *graph views* on relational tables, and to materialize graph structures in memory for the graph queries to execute on. Oracle Spatial and Graph [14] employs an access layer to ingest data from the Oracle databases or other sources into the Parallel Graph AnalytiX (PGX) [16], which is an in-memory graph analytic framework. All the above three systems suffer from the same problem of querying staled data, in face of frequent updates.

Microsoft SQL Server's graph extension [6] also allows some limited graph query capability inside the SQL Server database. However, it cannot retrofit to existing relational tables. One has to create vertex table(s) and edge table(s) first, then populate data into these tables. Implicit columns are automatically added to each vertex/edge table by the system, in order to uniquely identify each vertex/edge.

Cytosm [40] is a middleware application that automatically converts property graphs to the appropriate schema in a target backend storage (including relational databases) and translates Cypher queries to the target querying language. However, Cytosm doesn't apply the sophisticated optimization strategies that Db2 Graph employs during query compilation and execution.

Graph Processing on RDBMSs. There are a number of works [27, 31, 45] that advocate using RDBMSs for batch graph processing. However, they are not designed for low-latency graph queries, which is the focus of this paper.

3 BACKGROUND

Property Graph Model. 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. Figure 2(b) shows an example property graph, modeling the relationships between patients and diseases, as well as between different diseases. The vertex ids are the numbers shown inside the circles, and the edge ids are the numbers on the edges. There are two types of vertices labeled as `patient` and `disease` respectively (represented by the blue text next to each vertex in Figure 2(b)). An edge connecting a patient to a disease represents the `hasDisease` relationship, and an edge connecting a disease to another denotes the `isa` relationship (the former disease is a subtype of the latter) in the disease ontology (edge labels are the red texts on the edges). Vertex properties are shown inside blue shaded boxes, whereas edge properties are shown inside yellow shaded boxes (vertices or edges are allowed to have no properties). For example, vertex ① has properties `patientID`, `name`, `address`, and `subscriptionID`; and edge $-9\rightarrow$ has one property called `description`.

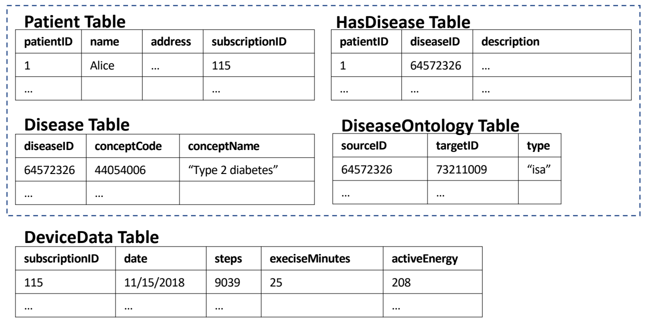
Apache TinkerPop and Gremlin. Apache TinkerPop [2] is an open source graph analytics framework that allows users to model their data as property graphs and analyze graphs using the Gremlin graph traversal language. Tinkerpop provides a core API for vertices, edges, and graphs. This core API essentially serves as an abstraction over different graph database implementations. The Gremlin language layer is built on top of the core API. Gremlin is more of an imperative language for traversing through a graph, although it also has some declarative features. The Gremlin language layer allows for extensions to support Domain Specific Languages (DSLs) and to add Provider Strategies for more optimized traversals for a specific graph database implementation. In addition to the two main layers, Tinkerpop provides a Read-Eval-Print Loop (REPL) environment, called Gremlin Console, and a service for remotely executing Gremlin scripts, called Gremlin Server.

In essence, TinkerPop is a collection of tools and facilities for supporting querying property graphs using Gremlin. The TinkerPop stack has made it very easy to develop a new graph database by simply implementing the core API. As a result, Gremlin, as well as other parts of the Tinkerpop stack, has been adopted by most graph databases. We will follow the same approach in this paper.

4 AN EXAMPLE SCENARIO

We will use the following example throughout the paper to motivate and illustrate Db2 Graph. This example is modeled after a real customer application and represents a wide range of real application needs.

Consider a health care application that combines patients' medical records with exercise data collected from wearable devices. Figure 2(a) shows the tables of data used in this application. The `Patient` table contains some basic information about a patient, as well as the `subscriptionID` that links the patient to the wearable device data in the `DeviceData` Table. A patient's disease information is captured in the `HasDisease` table. The actual disease is identified by a `diseaseID` which is also the unique key in the `Disease` table. The relationships between diseases (e.g. type 2 diabetes is a subtype of diabetes) are captured in the `DiseaseOntology` Table. Finally, the `DeviceData` table contains the daily exercise information for a user.



Patient Table

patientID	name	address	subscriptionID
1	Alice	...	115
...			...

HasDisease Table

patientID	diseaseID	description
1	64572326	...
...	...	...

Disease Table

diseaseID	conceptCode	conceptName
64572326	44054006	"Type 2 diabetes"
...	...	

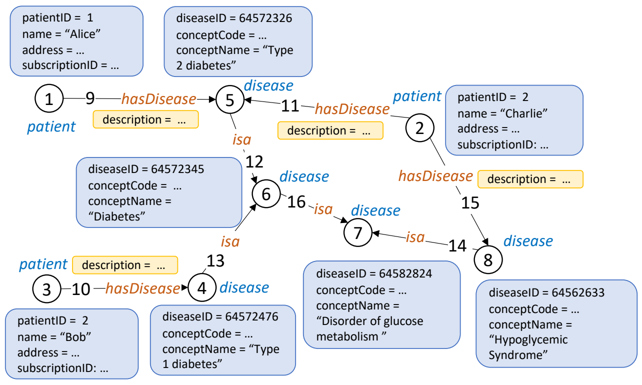
DiseaseOntology Table

sourceID	targetID	type
64572326	73211009	"isa"
...	...	

DeviceData Table

subscriptionID	date	steps	execiseMinutes	activeEnergy
115	11/15/2018	9039	25	208
...	...	...		...

(a) Example tables



(b) The property graph modeled on some example tables

Figure 2: An example scenario

The patients' medical records and the disease ontology data have always been stored in the relational database to support the existing applications of the customer. However, the customer also wants to support new applications that combine these data with wearable device data. In addition, they also wish to view the patient-to-disease and the disease-to-disease relationships (the 4 tables in the dashed-line box of Figure 2(a)) in the context of a graph as shown in Figure 2(b) and query the graph. Moreover, the new applications will require integrating graph queries with SQL analytics together.

Db2 Graph provides three interfaces for users to conduct graph queries: a command line interface called Gremlin console, a simple Db2 table function for submitting graph queries inside SQL, and programming APIs in a number of host languages including Java, Scala, Python and Groovy. As a result, there are different ways Db2 Graph can support synergistic SQL and graph queries together in one application. At the developing stage, users can have a SQL console and a Gremlin console opened side by side to query the same underlying data either as relational tables or as a property graph. The demo in [42] showed such an ad hoc insurance claim analysis using Db2 Graph (video available at https://www.youtube.com/watch?v=C5vmcYKEN-U).

Db2 Graph introduces a simple *polymorphic table function* [17] in Db2, called graphQuery, to submit graph queries inside SQL and bring the results back as a table[1]. For example, the following SQL statement finds patients that have similar diseases as those of a particular patient (with `patientID=1`), and compares their daily exercise patterns.

```
SELECT patientID, AVG(steps), AVG(execiseMinutes)
FROM DeviceData AS D,
 TABLE (graphQuery('gremlin', 'similar_diseases = g.V()
  .hasLabel(\'patient\').has(\'patientID\', \'1\').out(\'hasDisease\')
  .repeat(out(\'isa\').dedup().store(\'x\')).times(2)
  .repeat(in(\'isa\').dedup().store(\'x\')).times(2).cap(\'x\').next();
  g.V(similar_diseases).in(\'hasDisease\').dedup()
  .values(\'patientID\', \'subscriptionID\')'))
  AS P (patientID long, subscriptionID long)
WHERE D.subscriptionID = P.subscriptionID
GROUP BY patientID
```

In this query, finding the patients with similar diseases is done inside a graph query (highlighted in blue): it first traverses from the patient vertex to the connecting disease vertices, then traverses the disease ontology 2-hops up and 2-hops down, collecting all the diseases encountered along the way as similar diseases, finally finds all the patients that have any of these diseases and returns their `patientID`s and `subscriptionID`s. Afterwards, SQL joins the resulting table with the `DeviceData` table, and aggregates the average `steps` and `execiseMinutes` per patient.

At production stage, applications often use the JDBC API along with the programming API of Db2 Graph to mix SQL and graph queries in one workload. This approach is the most flexible and powerful in performing synergistic graph and SQL queries together in one application.

The example scenario described in this section is the perfect embodiment of the synergy between SQL and graph queries. Each does what it is good at, and together accomplish the task synergistically. Graph queries excels at navigating through complex relationships, whereas SQL is good at the heavy-lifting group-by and aggregation.

Using Db2 Graph, there is no need to have a separate system to handle graph queries, and even no need to replicate the four tables in a different format just for graph queries. Everything can be done in the same Db2 with no change to the existing tables. As a result, the graph data is always up to date with the latest information from the transactional system. For example, when patient information changes, the graph queries see these changes immediately. Moreover, the temporal support in Db2 allows all of our graphs to be temporal as well. For example, one can view a graph "as of" different time snapshots.

[1]Note that the graphQuery function only supports Gremlin graph queries with return results that can be converted into a collection of rows.

5 OVERLAYING GRAPHS ON TABLES

In this section, we describe the graph overlay approach in Db2 Graph. Graph databases always present a *single* property graph with a vertex set and an edge set for users to query. Of course, the single graph doesn't have to be fully connected, and there can be different types of vertices and edges. In Db2 Graph, we essentially map each vertex/edge in the vertex/edge set to a single row in the database. 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.

For the vertex set, the mapping needs to specify: 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 needs to specify: 1) what table(s) store the edge information, 2) what table columns are mapped to the required *id*, *src_v* (source vertex id), and *dst_v* (destination 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. Below, we show an example overlay configuration file in JSON format for mapping a property graph like in Figure 2(b) to the 4 tables in the dashed-line box of Figure 2(a).

```
1  "v_tables": [
2  {
3       "table_name": "Patient",
4       "prefixed_id": true,
5       "id": "'patient'::patientID",
6       "fix_label": true,
7       "label": "'patient'",
8       "properties": ["patientID", "name", "address", "
        subscriptionID"]
9  },
10 {
11      "table_name": "Disease",
12      "id": "diseaseID",
13      "fix_label": true,
14      "label": "'disease'",
15      "properties": ["diseaseID", "conceptCode", "
        conceptName"]
16 }],
17 "e_tables": [
18 {
19      "table_name": "DiseaseOntology",
20      "src_v_table": "Disease",
21      "src_v": "sourceID",
22      "dst_v_table": "Disease",
23      "dst_v": "targetID",
24      "prefixed_edge_id": true,
25      "id": "'ontology'::sourceID::targetID",
26      "label": "type"
27 },
28 {
29      "table_name": "HasDisease",
30      "src_v_table": "Patient",
31      "src_v": "'patient'::patientID",
32      "dst_v_table": "Disease",
33      "dst_v": "diseaseID",
34      "implicit_edge_id": true,
35      "fix_label": true,
36      "label": "'hasDisease'"
37 }]
```

The configuration file defines a set of vertex tables (`v_tables`, Line 1-16) for representing the vertex set of the property graph, as well as a set of edge tables (`e_tables`, Line 17-37) for representing the edge set of the property graph. In this example, `Patient` and `Disease` are the vertex tables, and `DiseaseOntology` and `HasDisease` are the edge tables. Then for each such vertex/edge table, it specifies how to define the required fields of a vertex/edge, as well as the properties. For a vertex, the required fields are `id` and `label`; for an edge, the required fields are `id`, `label`, `src_v` (source vertex id), and `dst_v` (destination vertex id).

In the property graph model, the *id* field is a requirement for each vertex and it has to be unique across the entire graph. The id field can be defined by one or more columns that uniquely identify a vertex, like column `diseaseID` in Line 12. However, when there are multiple relational tables mapping to the vertex set of a property graph, the unique key of a table may not always uniquely identify the associated vertex in the whole vertex set (across tables). As a result, we need to prefix the unique key with a unique table identifier to define the id field of a vertex. The unique table identifier can be the table name or some other unique constant value. This is the reason why Line 4 sets `prefixed_id` to true, and Line 5 defines `id` as `"'patient'::patientID"`. Here `'patient'` is a string constant, which serves as a unique table identifier of the `Patient` table. If `prefixed_id` is not set, it is false by default. As will be discussed in Section 6.3, prefixed ids can lead to more optimized performance at runtime.

The *label* field is also required in the property graph model. It can be mapped either to a column of the table, e.g. column `type` in Line 26, or to a constant, e.g. `'patient'` in Line 7. In common practice, different types of vertices or edges are typically stored in separate tables, which implies that all vertices or edges in a single table share the same *label* value. This means it is not necessary, and sometimes not possible, to use a column (in most cases, there is no such column) in the relational table to define the *label* field. For this reason, we introduce a feature to specify that all vertices or edges from a table have the same label and set that label to a constant string value. Lines 6 and 7 show such an example for the `Patient` table. As we will discuss in Section 6.3, this feature provides an optimization opportunity to narrow down the set of tables to query from at runtime.

Besides id and label fields, each edge table also needs to describe how the source and destination vertex ids, `src_v` and `dst_v` respectively, are defined. If all the source/destination vertices of an edge table come from one vertex table, one can also specify that source/destination vertex table, `src_v_table`/`dst_v_table`. In such cases, the source/destination vertex id definition has to match exactly with the id definition of the corresponding vertex table. For example, the `src_v` definition `"‘patient’::patientID"` of the `HasDisease` edge table (Line 30-31) matches with the `id` definition `"‘patient’::patientID"` (Line 4-5) of the `Patient` vertex table. In this case, the `HasDisease.patientID` column and the `Patient.patientID column` happen to have the same name, but they don't need to. As another example, the `src_v` definition of the `DiseaseOntology` edge table (Line 20-21) still matches with the `id` definition of the `Disease` vertex table (Line 12), although the column names are different. When `src_v_table`/`dst_v_table` is specified, it establishes a relationship between a vertex table and an edge table, and subsequently provides an optimization opportunity to reduce the search space during graph traversal, as will be discussed in Section 6.3.

The definition of edge ids encounters similar problems as the vertex ids, so we can follow the similar approach (e.g. Line 24-25). However, for the edges, in most cases, the combination of the three fields, `src_v::label::dst_v` can already uniquely identify an edge in the entire edge set. As a result, we don't need to always explicitly specify edge id fields. In Lines 34, by setting `implicit_edge_id` to true, we indicate that the edges from `HasDisease` table will use the implicit `src_v::label::dst_v` as the edge ids. As we will show in Section 6.3, the implicit id definitions for edges will bring in some optimization opportunities for query execution.

The properties of a vertex/edge can be mapped to columns of the table (e.g. Line 8). Vertices/edges are allowed to have no property. In that case, one can define an empty set `[]`. Note that if *properties* is not specified in the configuration, Db2 Graph automatically takes all the columns in a table except the ones already used for the required fields for defining the properties. For example, this configuration file doesn't specify *properties* for the `HasDisease` table, but it is equivalent to defining `["description"]` in the file.

Note that one table can serve as both a vertex table and an edge table in a graph overlay. This is common for tables with foreign keys, as they store both the information about objects and their relationships to other objects. In addition, sometimes one table can serve as multiple edge tables, which is very common for the fact table in a star schema.

Our graph overlay approach maps each row of a table to either a vertex or an edge. For cases that do not fit in this model, users can usually define views on existing tables to work around. For example, if a vertex requires information from a row in Table A and another row in Table B, we can create a view C that joins A and B, and then use C in the graph overlay. As another example, if a row of a table can be mapped to two different vertices, we can create two views that extract the corresponding information for each vertex, and use the views in the graph overlay.

A Surprising Benefit. Most existing graph databases start from a graph, whereas Db2 Graph starts from relational tables and maps them into a graph. While we never envisioned it, through customer usage we found a surprising benefit of this graph overlay approach. Customers often would like to define new types of vertices or relationships based on existing graphs. For instance, in an existing graph, patients are linked to doctors who are then linked to service providers. A customer wanted to create new edges that directly link patients to service providers, i.e. if p⟶d and d⟶s, then p⟶s. Previously using a standalone graph database, the customer had to insert millions of edges to the graph. But using Db2 Graph, it is as simple as defining a non-materialized view that joins two existing edge tables and map that view as an edge table in the graph overlay. Moreover, when existing edges change (e.g. the edge d⟶s is deleted), the customer previously had to write customized logic to ensure that the derived edges are changed accordingly (e.g. deleting p⟶s), whereas in Db2 Graph these changes are automatically reflected in the view, and thus in the overlaid graph.

ALGORITHM 1: Identify Vertex Tables and Edge Tables

```
input  : tables    //The set of input tables with schema and
                     primary key and foreign key information
1 Initialize: vertexTables = ∅, edgeTables = ∅
2 for t ∈ tables do
3     if t.hasPrimaryKey() then
4         vertexTables.add(t)              //t is a vertex table
5         if t.foreignKeys.nonEmpty() then
6             edgeTables.add(t)            //t is also an edge table
7     else
8         if t.foreignKeys.size() ≥ 2 then
9             edgeTables.add(t)            //t is an edge table
output : (vertexTables, edgeTables)
```

5.1 Automatic Overlay Generation

The overlay configuration files can be manually created by the application developers who wish to query relational tables as graphs. One can create multiple overlay configuration files on the same set of tables, so that they can be queried as different graphs. Manual specification provides a lot of flexibility, but can also be labor intensive, if there are a large number of tables. For example, Db2 Graph was once used to overlay a property graph onto 135 tables! We now discuss how to automate the generation of overlay configuration for a database in a principled way.

Our approach relies on table schemas, along with the primary and foreign key constraints, to infer relationships among the data in relational tables. This shares a lot of similarity with the work on converting relational databases to

ALGORITHM 2: Generate Overlay Configuration

```
input  : vertexTables, edgeTables
Initialize: vertexTableConfs = ∅, edgeTableConfs = ∅
for t ∈ vertexTables do
    vtc = newVertexTableConf()
    vtc.idField = combine(t.uniqueID, t.primaryKey)
    vtc.hasFixedLabel = true
    vtc.fixedLabel = t.tableName
    vtc.propertyFields = t.columns - t.primaryKey
    vertexTableConfs.add(vtc)
for t ∈ edgeTables do
    if t.hasPrimaryKey() then
        for fk ∈ t.foreignKeys do
            etc = newEdgeTableConf()
            etc.hasImplicitEdgeID = true
            etc.srcvTable = t.tableName
            etc.srcvField = combine(t.uniqueID, t.primaryKey)
            etc.dstvTable = fk.refTable.tableName
            etc.dstvField = combine(fk.refTable.uniqueID, fk)
            etc.hasFixedLabel = true
            etc.fixedLabel = concat(t.tableName, fk.refTable.tableName)
            etc.propertyFields = t.columns - t.primaryKey - fk
            edgeTableConfs.add(etc)
    else
        for fk1, fk2 ∈ t.foreignKeys and fk1≠fk2 do
            etc = newEdgeTableConf()
            etc.hasImplicitEdgeID = true
            etc.srcvTable = fk1.refTable.tableName
            etc.srcvField = combine(fk1.refTable.uniqueID, fk1)
            etc.dstvTable = fk2.refTable.tableName
            etc.dstvField = combine(fk2.refTable.uniqueID, fk2)
            etc.hasFixedLabel = true
            etc.fixedLabel = concat(fk1.refTable.tableName, t.tableName,
              fk2.refTable.tableName)
            etc.propertyFields = t.columns - fk1 - fk2
            edgeTableConfs.add(etc)
output : (vertexTableConfs, edgeTableConfs)
```

graph databases [25, 36–38, 43], since the data conversion also uses schema and functional dependencies to convert relational data to graphs. But the major difference between the two is that data conversion creates a completely new copy of the original data, whereas graph overlay merely creates a *virtual* graph view on top of the original data. As a result, the graph overlay generation needs to come up with a graph schema that more faithfully captures the semantics of the relational data, whereas the data conversion approaches can have more freedom. For example, the approach in [25] makes every attribute (column) of a table as a vertex in the graph.

Db2 Graph provides a toolkit, called AutoOverlay, for automatically generating a graph overlay. A user can specify which database he/she wants to generate the overlay configuration for. If only a subset of tables in a database are of interest, the user can also explicitly list these tables. AutoOverlay automatically generates the overlay configuration in the following steps:

Step 1: AutoOverlay first queries Db2 catalog to get all the metadata information for each table such as table schema, and primary key/foreign key constraints.

Step 2: Then, it iterates through all tables to find out the vertex tables and edge tables as shown in Algorithm 1. Note that a table can serve as both a vertex table and an edge table. Any table with a primary key will be served as a vertex table. If a table has a primary key and one or more foreign keys (e.g. a fact table in a star schema), it will also be used as one or more edge tables, one for each foreign key. If a table has k ($k \geq 2$) foreign keys but no primary key (many-to-many relationships), then it will be used as $\binom{k}{2}$ edge tables, one for each pair of foreign keys.

Step 3: Finally, AutoOverlay maps the required fields in the property graph model to columns in the vertex/edge tables, as shown in Algorithm 2. For each vertex table, the primary key columns together with a unique table identifier are used to define the id field. The label field is defined as a fixed label with the table name as its value. All remaining columns, except the primary key columns, will be properties.

For each edge table with a primary key and a foreign key, the src_v_table field is defined as the table itself, and the src_v field is defined as the primary key columns prefixed with the table identifier. The dst_v_table field is the referenced table (table referenced by the foreign key), and the dst_v field is the foreign key columns prefixed with the identifier of the referenced table. The label field is the concatenation of the table name and the referenced table name.

For each edge table with a pair of foreign keys, the src_v_table field is the 1^{st} referenced table, and the src_v field is the 1^{st} referenced table identifier combined with the 1^{st} foreign key columns. The dst_v_table field is the 2^{nd} referenced table, and the dst_v field is the 2^{nd} referenced table identifier combined with the 2^{nd} foreign key columns. The label field is the concatenation of the table name and the names of the referenced tables. For both cases of edge tables, we use the implicit src_v::label::dst_v as the edge id. Finally, all remaining columns, except for primary key and foreign key columns, are properties.

As can be seen in these steps, we heavily rely on the primary and foreign key constraints to infer relationships among the data in relational tables. If no constraint is specified, one can still manually specify overlay configuration. After the overlay configuration is generated automatically by AutoOverlay, the user can still edit the configuration file to modify the mapping information to better fit the application. Currently, the AutoOverlay toolkit is not integrated with the Db2 catalog, so when there is any DDL change (table

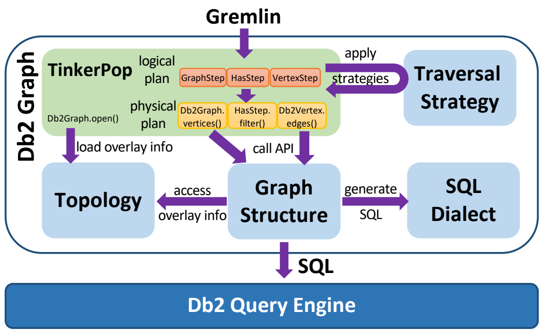


Figure 3: Db2 Graph architecture

creation, deletion, or schema change), the user needs to rerun AutoOverlay to generate the new overlay. We plan to better integrate it with the Db2 catalog so that DDL changes can be automatically dealt with in the future. We also plan to further employ machine learning techniques to infer the relationships among tables.

6 ARCHITECTURE

Figure 3 shows the overall architecture of Db2 Graph. It includes the TinkerPop stack and four native system modules. The TinkerPop stack compiles an input Gremlin query, generates a query plan, and further optimizes the query plan by applying the optimized traversal strategies provided in our Traversal Strategy module. After that the query plan is executed by calling the graph structure API provided in our Graph Structure module. The implementation of the graph structure API, in turn, requires accessing the graph overlay information maintained in the Topology module and submitting SQL queries generated via the SQL Dialect module.

6.1 End-to-End System

Let's use the query `g.V().has('name', 'Alice').outE()` (retrieving the outgoing edges of the vertices that have a property called 'name' with value 'Alice') as an example to discuss how the different modules of Db2 Graph work together to execute the query.

Opening a Graph for Traversal. Before we can run the query, the corresponding property graph g has to be opened for traversal first via `g=Db2Graph.open('config.properties').traversal()`. The file config.properties specifies the database access information, the graph overlay configuration file, and other properties for the property graph. The Topology module reads the overlay configuration file and establishes the overlay mapping from the property graph onto the relational tables in the database by accessing the database metadata. Later, the Graph Structure module refers to the overlay mapping in the Topology module to properly carry out its functions. In particular, the overlay topology can tell us which table(s) contains vertices/edges with a particular label or a particular property name, and whether the source/destination vertices of all the edges in an edge table are from a specific vertex table. As we will show later in Section 6.3, this information is crucial in reducing the search space during query execution.

Query Compilation and Optimization. When the query `g.V().has('name', 'Alice').outE()` is issued, it first goes through the Tinkerpop stack, where it's parsed and compiled into a logical query plan with a number of steps. Each step captures a transformation (e.g. `outE()`), filter (e.g. `hasLabel()`), side-effect (e.g. `store()`), or branch function (e.g. `union()`). The example query will be compiled into a logical plan with 3 sequential steps: the first step, corresponding to `V()`, is a transformation step, called GraphStep (since the transformation is applied on a graph), which returns all the vertices of the graph; then the second step that corresponds to `has('name', 'Alice')` is a filter step, called HasStep, which filters the vertices and only retains the ones having a property called 'name' with value 'Alice'; and finally the last step (that corresponds to `outE()`) is another transformation step, called VertexStep (since the transformation is applied on a vertex), which returns all the outgoing edges of the resulting vertices from the second step.

Analogous to query optimization in a database optimizer, a Tinkerpop logical plan can be mutated into a more optimized plan by applying a number of *traversal strategies*. Tinkerpop has already included a number of such built-in traversal strategies, but it also opens up a Provider Strategy API for graph database developers to add customized optimization strategies specific to the particular graph database implementation. In Db2 Graph, we add a number of such optimized provider strategies in the Traversal Strategy module, which will be discussed in Section 6.2. Actually, the 3-step plan for the example query will be mutated into a more optimized plan using the optimized strategies provided in Db2 Graph. But for the simplicity of explanation in this subsection, let's assume the 3-step plan is unchanged.

Query Execution. After query optimization, the logical plan will be translated into the actual physical implementation. Some of the steps in the plan will result in calling the graph structure API. In the above example query, the GraphStep (corresponding to `V()`) and the VertexStep (corresponding to `outE()`) both need to call the graph structure API, since they need to access the basic vertex and edge information in order to execute. However, the HasStep (corresponding to `has('name', 'Alice')`) only needs to filter vertices based on their property values which are already obtained in the previous step, thus there is no need to access the graph structure API. We call a step that requires access to the graph structure API as a *Graph-Structure-Accessing* step, or a GSA step for short. In Db2 Graph, each GSA step typically results in one or more SQL queries to Db2. We can affect the execution of these steps through our implementation of the corresponding graph structure API.

The Graph Structure module contains our implementation of the Tinkerpop graph structure API. The basic graph structure API includes Graph, Vertex, Edge, VertexProperty, and Property, as well as graph operations on them, such as getting vertices/edges by ids and getting the adjacent vertices/edges of a vertex/edge. We also extend the basic API to carry out more sophisticated functionalities (e.g. predicate, projection, and aggregate pushdown) in response to the optimized query plans resulted from applying the optimized strategies from the Traversal Strategy module, as will be discussed in Section 6.3. The Graph Structure module refers to the overlay mapping in the Topology module to decide on how to implement graph operations. The implementation of the graph structure API affects the execution of all the GSA steps in Db2 Graph, so we strive to optimize as much as possible. In Section 6.3, we will highlight how we apply the data-dependent optimizations at runtime by utilizing the graph overlay topology. Finally, the Graph Structure module utilizes the SQL Dialect module to translate the graph operations into SQL queries.

The SQL Dialect module deals with everything related to Db2. It generates all the SQL queries needed for implementing graph operations. This module also keep tracks of these SQL queries and finds out frequent query patterns. For example, if the `name` column is frequently used (above a pre-set threshold) in the predicates for querying the `Patient` table, the SQL Dialect module will consider it as a frequent query pattern. It then creates a set of pre-compiled SQL templates for these frequent patterns and issues the corresponding prepare statements in Db2 to avoid the SQL compilation overhead at runtime. Based on these SQL templates, it also suggests indexes (e.g. an index on the `name` column of the `Patient` table) that would speed up the execution of the translated SQL queries. This module can also provide hints to the Db2 index advisor, which can look at the entire workload and advise indexes.

Let's take the 3-step logical plan for the example query, and illustrate how it is executed. Note again that the actual execution of this query is much more optimized in Db2 Graph. But, we stick with this naive version of the execution for the simplicity of explanation. Only the GraphStep and the VertexStep in this plan are GSA steps, thus only these two actually call our graph structure API implementation. Each API function implementation needs to decide 1) what relational tables to query from, and 2) for each table, what SQL query to submit. The GraphStep, corresponding to `V()`, retrieves all vertices from the graph. So, only vertex tables need to be queried. But there is no extra information to help us narrow down a subset of the vertex tables. As a result, for every vertex table, we need to submit a SQL query like `"SELECT * FROM VertexTable"`. The second HasStep, corresponding to `has('name','Alice')`, is executed inside Db2 Graph (no SQL query is needed). And finally the VertexStep, corresponding to `outE()`, needs to query every edge table and submit a SQL query like `"SELECT * FROM EdgeTable where src in (id1, id2, ...)"`, where id1, id2 and so on are the ids of the vertices from step 2.

Obviously, the above execution plan is very inefficient: the first step returns all the vertices in the graph even though the second step would filter out most of the vertices. In the following two subsections, we describe the optimization techniques employed in Db2 Graph to address the inefficiency. These optimization techniques aim at 1) eliminating the unnecessary tables to query from, and 2) generating more optimized SQL queries to reduce the query latency and the returned results for each necessary table. The optimization techniques can be grouped into two categories. The first category is all about the optimized traversal strategies in the Traversal Strategy module applied during query optimization. These techniques are *data-independent*, i.e. they don't need to know anything about the underlying relational data or how they are mapped to the property graph. In comparison, the techniques in the second category are all applied at the runtime execution in the Graph Structure module. And they are *data-dependent*, i.e. they need to access the graph overlay mapping information in the Topology module.

6.2 Optimized Traversal Strategies

We first discuss the data-independent, optimized, traversal strategies in the Traversal Strategy module applied during query optimization. The traversal strategies mutate a query plan whenever a pattern is matched, eventually generating more optimized SQL queries. For all of these optimized strategies, we start from a GSA step, since it results in SQL calls.

Predicate Pushdown with Filter Steps. When a GSA step is followed by a sequence of filter steps, we can fold these filter steps as extra predicates into the GSA step. Consider the first two steps of the previous example query `g.V().has('name', 'Alice')`. Now, the HasStep can be folded into the GraphStep. And the new GraphStep with the extra predicate can be translated into one SQL query `"SELECT * FROM VertexTable WHERE name = 'Alice'"`. We basically push down all the filter steps into the "where" clause of the SQL statement for the GSA step (the filter steps are all removed in the optimized plan), which results in a significant reduction of the SQL runtime and the results returned from the database. This is, in some sense, very similar to query push down in the setting of federated databases [39].

Projection Pushdown with Properties Steps. Graph traversals often fetch some particular vertex or edge properties, for example `g.V().values('name', 'address')`, which fetches the values of the `name` and `address` properties. When a GSA step is followed by such a Properties Step, we can use the provided set of property names to help reduce the projected columns for the GSA step. In this example, the GraphStep can be translated into `"SELECT id, label, name, address FROM VertexTable"` instead of `"SELECT * FROM VertexTable"`.

Aggregate Pushdown with Aggregation Steps. Gremlin supports a number of aggregate functions, such as count, sum, mean, min, and max. When such an aggregate function follows a GSA step, we can push down the aggregate function into the SQL statement of the GSA step. For the example query `g.V().count()`, instead of retrieving all the vertices from the database (with `"SELECT * FROM VertexTable"`) and then computing the count, we can combine the two steps and submit a SQL query `"SELECT COUNT(*) FROM VertexTable"`. This obviously significantly reduces the amount of data transferred from Db2 to Db2 Graph and dramatically improves the query performance.

GraphStep::VertexStep Mutation. This optimized strategy applies when a GraphStep that retrieves vertices is followed by a VertexStep. For the example query `g.V(ids).outE()`, which retrieves all the outgoing edges of vertices with id in the set of `ids`, the default Gremlin strategy would turn it into two SQL queries: `"SELECT * FROM VertexTable WHERE id in (ids)"` followed by `"SELECT * FROM EdgeTable WHERE src_v in (ids)"`. It is obvious that the first SQL query is a total waste. The second SQL query alone can provide the needed results for the graph query, as the vertex ids are also stored in the edge tables as source vertex ids.

To eliminate the unnecessary scan on the vertex table (corresponding to `g.V()`), we mutate the GraphStep::VertexStep steps into a new GraphStep that retrieves edges, and pass the `ids` as a predicate on the edges into the GraphStep. And if the original VertexStep retrieves vertices (e.g. in the case of `g.V(ids).out()`), we also add an EdgeVertexStep (corresponding to `inV()` in this case) that retrieves the desired vertices after the new GraphStep.

All the above strategies can be combined together during query optimization. For the example query `g.V(ids).outE() .has('metIn', 'US').count()`, the GraphStep::VertexStep mutation will be applied first, followed by the predicate pushdown, and finally aggregate pushdown will be applied. So, the end result is one optimized SQL query `"SELECT COUNT(*) FROM EdgeTable WHERE src_v in (ids) AND metIn='US'"`.

6.3 Data-Dependent Optimizations

We now discuss the data-dependent runtime optimizations employed in the Graph Structure module of Db2 Graph. In Db2 Graph, every vertex/edge in the property graph comes from a particular table. We record this information in the basic vertex and edge data structures so that we can access this information at runtime. Since we allow a property graph to overlay on top of multiple vertex and edge tables, by default, when we query vertices/edges from the graph, we need to query all the vertex/edge tables to ensure the correctness of the query. However, as we will show in this subsection, we can utilize the belonging table of a vertex/edge along with the graph overlay topology from the Topology module to eliminate, as much as possible, the unnecessary tables that we need to query from at runtime.

Using Source/Destination Vertex Tables. The source/destination vertex table definition (`src_v_table/dst_v_table` in Section 5) for an edge table offers a most straightforward way to eliminate unnecessary tables. Suppose that we need to query the adjacent outgoing vertex of a given edge e (i.e. `e.outV()`). Without any optimization, this query will result in a SQL query `"SELECT * FROM VertexTable WHERE src_v = e.src_v"` for every single vertex table. However, if the `src_v_table` is defined for e's edge table in the graph overlay, then we just need to query exactly one table. This is a significant improvement, especially when the graph maps to a large number of vertex tables.

When A Vertex Table Is Also An Edge Table. It is fairly common to have a relational table to serve as both a vertex table and an edge table in the graph overlay, as the table may contain both information about objects and their relationships to other objects (e.g. in the case of a fact table in a star schema). This special case sometimes provides a great optimization opportunity to avoid unnecessary queries all together. Let's again take `e.outV()` (querying the adjacent outgoing vertex of edge e) as an example. If not only the `src_v_table` is defined and but also it is the same as e's edge table, then the queried vertex and the edge e refer to exactly the same row in the common table. If, additionally, all the columns used to define the properties and required fields for the vertex are subsumed by those for the edge, then we can simply construct the vertex from the edge itself, thus avoiding a SQL query all together.

Using Property Names in Pushdown Information. Recall that we add a number of optimized traversal strategies in the Traversal Strategy module to pushdown predicates, projections, and aggregates into the GSA steps. We also extend the graph structure API accordinly to take in these extra pushdown information for more efficient implementation. It turns out that these pushdown information not only can help generate more optimized SQL queries (by adding predicates, projections, and aggregates in the SQL), but also can help eliminate unnecessary query tables. Whenever a pushdown predicate or projection is present, e.g. `has('name', 'Alice')` or `values('name')`, the specified property, e.g. `'name'`, has to exist for the query to return a result. Since we have all the property information for each vertex/edge table defined in our overlay configuration, we easily know whether the specified property exists or not in a vertex/edge table. Then, only the tables having the required property need to be queried.

Using Label Values. Label is a very special property in a property graph, thus a very common operation in Gremlin is to retrieve vertices/edges by label(s), e.g. `g.V().hasLabel('patient')` and `v.outE('hasDisease')`. Naively, this would result in querying through all the vertex/edge tables with a predicate on the given label(s). But, our graph overlay configuration allows a vertex/edge table to have a fixed label (i.e. `fixed_label` is true) for all the vertices/edges in it. When this happens, we can use the specified label(s) to narrow down a subset of vertex/edge 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. Note that the implementation still has to search all the tables without fixed labels to make sure it is not missing any results. This optimization provides a huge performance improvement.

Using Prefixed Id Values. Another very basic operation in the graph structure API is looking up vertices/edges by ids. When a given id is a prefixed id (unique key prefixed with a unique table identifier), e.g. `'patient'::1`, we can use the unique table identifier to pin down the exact table to search from (in this example, the Patient table), instead of blindly querying through all the tables. In addition, when the id field is the concatenation of multiple table columns (e.g. `'TableName'::c1::c2`), Db2 Graph extracts the individual column values from an id value and forms conjunctive predicates (e.g. `c1=c1_value and c2=c2_value`) in SQL queries.

Using Implicit Edge Id Values. For edge ids defined with the implicit src_v::label::dst_v combination, when fixed labels are specified for edge tables, we can also utilize the label encoded in the id field to narrow down the tables to search from for looking up an edge by its id, similar to how we eliminate tables using label values. Similar to the prefixed id values, Db2 Graph also breaks apart the implicit edge id values to form conjunctive predicates in SQL queries.

7 REAL WORLD USAGE

Db2 Graph has been used in a number of real customer engagements. In finance, an example application is mule fraud detection, where graph queries are used to detect how a set of fraudsters are connected to a set of beneficiaries through a sequence of mule accounts. The dataset for this application is bank transaction data, which are updated frequently through the bank's operational functions and also used by existing SQL analytical applications. The timeliness of the fraud detection requires the graph queries to access the latest transaction data. As a result, importing all the transaction data to a standalone graph database would be expensive and hard to satisfy the timeliness requirement.

In health care, Db2 Graph has been used for patient case study on a dataset that contains patient EMR records, hospital discharges, and health insurance information. This application views patients, hospital discharges, diagnoses, lab results, medical procedures, drugs, and insurance enrollment information etc. all as vertices of a graph. The queries traverse through the graph to find out how a patient is treated in an inpatient service (e.g. what tests have been done, what procedures have been performed, what diagnoses have been made, and what drugs have been given), as well as how the costs are covered by insurance. These are path queries starting from a single vertex. There are already a large number of existing applications reading and writing this dataset, and the graph queries need to work on the latest data. In addition, the demo in [42] showed another detailed scenario for health insurance claim analysis using Db2 Graph. In this application, SQL and graph queries need to work synergistically, hence using a standalone graph database would not be ideal.

In law enforcement, Db2 Graph has been used to query a police department dataset that contains information about persons (suspects, victims, or witnesses), organizations (legitimate organizations or gangs), arrests, warrants, complaints, vehicles, locations, emails, and phones. The application views all these entities as vertices and conducts case studies, such as finding the phone numbers and addresses of the suspects in an arrest, and figuring out the criminal organizations that all suspects of an arrest belong to. Again the workload consists of path queries starting from a single vertex. The dataset is updated in real time, thus using a standalone graph database would be hard to keep data up-to-date for graph queries.

So far, the largest real graph that Db2 Graph has worked on, contains about 4 billion vertices and 6 billion edges with roughly half a terabyte of data. The graph workload consists of Gremlin queries that traverse from some vertices, satisfying certain conditions, up to 4 hops away, with optional predicates on the traversal paths. The observed average query response time using Db2 Graph is sub 100 milliseconds.

8 EVALUATION

In the following, we report experimental results of Db2 Graph on a synthetic benchmark. Recall that we focus on the scenarios where graph data already exist in the relational database, and the design goal of Db2 Graph is not to be the fastest in graph-only queries. Nevertheless, we still compare the graph query performance of Db2 Graph against two state-of-the-art graph-only databases: GDB-X, a commercial high-performance native graph database (the name is anonymized due to the sensitivity of reporting its performance numbers), and JanusGraph [10], a popular open-source graph database (with Berkeley DB [13] as the backend store).

We used the Linkbench [11] graph benchmark for performance evaluation. Here, we only focus on the query only

Table 1: LinkBench Queries

LinkBench Query	Gremlin
getNode(id, lbl)	g.V(id).hasLabel(lbl)
countLinks(id1,lbl)	g.V(id1).outE(lbl).count()
getLink(id1,lbl,id2)	g.V(id1).outE(lbl).filter(outV().id() == id2)
getLinkList(id1,lbl)	g.V(id1).outE(lbl)

Table 2: Linkbench Datasets

Linkbench Dataset	Num Of Vertices	Num Of Edges	Avg Degree	Max Degree	CSV File
10M	10M	43M	4.3	961,970	4.3G
100M	100M	419M	4.2	962,000	42G

workloads in LinkBench. Table 1 lists the 4 types of graph queries in LinkBench. We understand that the Linkbench queries are not complex graph queries. However, just as advocated by the experimental work in [34], we also observed that a *microbenchmark* with simpler queries provides a better understanding of the pros and cons of each graph database, compared to a *macrobenchmark*.

For all the experiments, we used a Ubuntu server with 32 CPU cores and 256GB memory. For fair comparison, we used the same Gremlin Server configuration and gave the same 64GB JVM to all three graph databases, as well as building all the indexes necessary for each system to get good performance. Table 2 shows the Linkbench datasets we used in our experiments. For the two datasets, each vertex has 3 properties and each edge has 4 properties. There are 10 types of vertices and also 10 types of edges.

Effect of Optimized Traversal Strategies. We first evaluate the effect of the optimized traversal strategies in Section 6.2 on the overall performance of Db2 Graph. Figure 4 compares the average latency of the Linkbench queries on the Linkbench-10M dataset with all the optimized strategies turned on and off, respectively. Note that the data-dependent runtime optimizations in Section 6.3 are still applied in both situations. As can be seen, all of the queries significantly benefit from the optimized traversal strategies, with 2.8X to 3.3X speed up in performance. In particular, the getNode query mainly benefits from the predicate pushdown strategy; the remaining three queries all benefit from the GraphStep::VertexStep mutation strategy; the countLink query additionally benefits from the aggregate pushdown strategy;

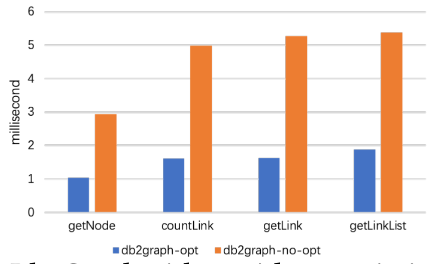


Figure 4: Db2 Graph with vs without optimized traversal strategies: latency on Linkbench-10M dataset

and the getLink query additionally benefits from the predicate pushdown strategy.

Graph Loading Time. Since we assume that graph data have already existed in the relational database, GDB-X and JanusGraph require graph data to be reloaded into their own graph databases before they can be queried as a graph. This loading time includes exporting the data out of the relatonal database, loading the data as a graph into GDB-X or JanusGraph, and opening the graph to be queried. Table 3 lists the breakdown of the loading times. As a reference, we also include the numbers for Db2 Graph in the table. Db2 Graph would require no time for loading relational data as a graph, since it supports directly querying relational data as a graph. The only overhead is the graph opening time, which is a couple of seconds. As the table shows, even exporting data out of the relational database takes from 4 minutes to half an hour. Then loading the data into the graph database takes 42 minutes to 8 hours for GDB-X! Opening the graph takes another 14 to 15 seconds in GDB-X. This slow open time is caused by the aggressive prefetching and caching strategies adopted in GDB-X. Another thing worth noting is that the disk space used to store the graph data in GDB-X is 6-7X of the original relational tables, on which Db2 Graph can directly operate. For JanusGraph, loading the graph is even more painful (13.5 hours for LinkBench-100M), and the disk usage is also on par with GDB-X.

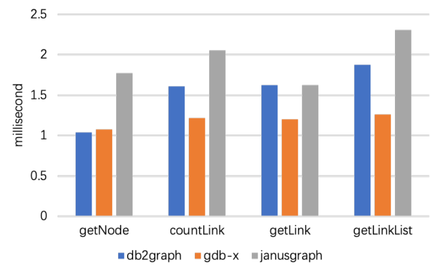
In summary, the results show that it is simply *infeasible* to use GDB-X or JanusGraph to interactively query graph data already stored in a relational database *at runtime*. The only way for them to carry out graph analysis on the existing relational data is to pre-load the data to the graph database ahead of time. Of course, this also raises consistency issues on the two copies of data.

Graph Query Performance. Finally, we compare the query performance of Db2 Graph against GDB-X and JanusGraph. All graph databases were running in server mode and responding to requests from clients at localhost.

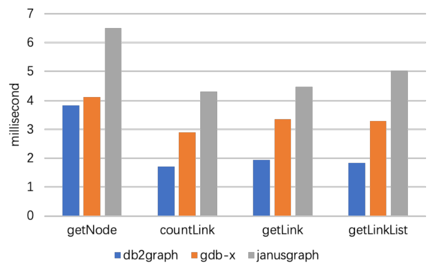
Figure 5 demonstrates the latency of LinkBench queries. JanusGraph is always the slowest (up to 2.7X slower than Db2 Graph). On the smaller Linkbench-10M dataset, GDB-X presents the best latency for almost all queries, although Db2 Graph is slightly better for the getNode query. This is not a surprise, since GDB-X is a native graph database. Its optimized graph storage layout and the aggressive caching mechanism really help lower the query latency. This smaller dataset can fit entirely in the in-memory cache. Nevertheless, the difference between Db2 Graph and GDB-X is still within 1.5X. However, when we move to the much larger Linkbench-100M dataset, the story completely changed: Db2 Graph even beats GDB-X up to 1.7X. This is because, for GDB-X, the graph data that occupy 327GB disk space, cannot be cached entirely in memory anymore. In comparison, the

Table 3: Graph loading time for different graph databases

	Db2 Graph		Export	GDB-X			JanusGraph		
Linkbench Dataset	Disk Usage	Open Graph	From DB	Disk Usage	Load Data	Open Graph	Disk Usage	Load Data	Open Graph
10M	4.6GB	1.4 sec	5 min	28GB	42 min	14 sec	29GB	65 min	15 sec
100M	45.8GB	2.1 sec	32 min	327GB	8 hr	15 sec	326GB	13.5 hr	17 sec

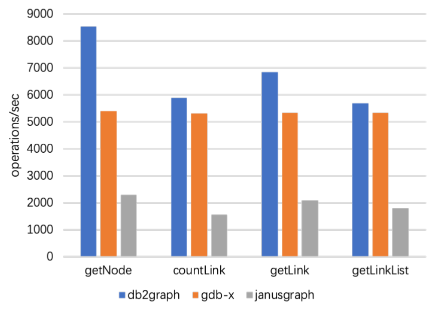


(a) Linkbench-10M

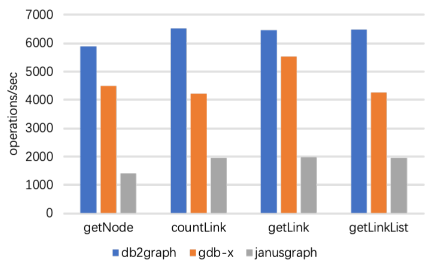


(b) Linkbench-100M

Figure 5: Latency of Linkbench Queries



(a) Linkbench-10M



(b) Linkbench-100M

Figure 6: Throughputs of Linkbench Queries

entire 45.8GB relational data can sit comfortably in Db2's buffer pool.

We compare the throughput of the three systems in Figure 6, by having 50 clients simultaneously submitting Linkbench queries. Db2 Graph is the clear winner in all cases, beating GDB-X up to 1.6X and JanusGraph up to 4.2X. This is because the underlying Db2 engine is extremely good at handling concurrent queries. Even though GDB-X might run each query faster on the smaller Linkbench-10M dataset, it cannot keep up with the large amount of concurrency.

Overall, in terms of query latency, GDB-X's sweet spot is when the graph data are relatively small and can fit comfortably in its in-memory cache. But even for this case, Db2 Graph is within a very reasonable range of GDB-X. As graph data grow bigger, Db2 Graph starts to excel, due to the aggressive optimization strategies applied in Db2 Graph and, perhaps more importantly, the robust performance of the underlying Db2 engine. When talking about query throughput, Db2 Graph clearly outperforms the competitors, due to Db2's superiority in handling concurrent queries. JanusGraph is always the worst in terms of both latency and throughput.

Again, all the experiments in this subsection are for graph-only queries. When considering graph queries as a part of a more complex analytics workload, the standalone GDB-X and JanusGraph graph databases will have to pay for other more expensive overhead such as data transfer and transformation. In contrast, the advantage of the in-DBMS graph querying approach adopted by Db2 Graph becomes more prominent in these real application scenarios.

9 CONCLUSION

This paper introduced IBM Db2 Graph, a layer inside Db2 that supports Gremlin graph queries. Db2 Graph took an in-DBMS graph query approach, making graph queries synergistic with SQL analytics. By employing a novel graph overlay approach, it flexibly retrofits graph queries to existing relational data. To achieve query efficiency, Db2 Graph adopted various compile-time and runtime optimization techniques. Through empirical studies and real customer experiences, we showed that Db2 Graph achieves very competitive and sometimes even better performance compared to existing graph databases. Furthermore, in the context of real application scenarios with integrated SQL, graph, and other analytics, Db2 Graph demonstrated more prominent advantage over existing approaches.

REFERENCES

[1] Apache Giraph. http://giraph.apache.org.
[2] Apache TinkerPop. http://http://tinkerpop.apache.org.
[3] ArangoDB. https://www.arangodb.com.
[4] BlazeGraph. https://www.blazegraph.com.
[5] GQL. https://www.gqlstandards.org.
[6] 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.
[7] GSQL. https://www.tigergraph.com/gsql.
[8] IBM Db2. https://www.ibm.com/analytics/us/en/db2.
[9] InfiniteGraph. https://www.objectivity.com/products/infinitegraph.
[10] JanusGraph. http://janusgraph.org.
[11] LinkBench: A database benchmark for the social graph. https://www.facebook.com/notes/facebook-engineering/linkbench-a-database-benchmark-for-the-social-graph/10151391496443920.
[12] Neo4j. https://neo4j.com.
[13] Oracle Berkeley DB. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html.
[14] Oracle Spatial and Graph. https://www.oracle.com/database/technologies/spatialandgraph.html.
[15] OrientDB. https://orientdb.com.
[16] Parallel Graph AnalytiX (PGX). https://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytix/overview/index.html.
[17] Polymorphic table functions in SQL. https://standards.iso.org/ittf/PubliclyAvailableStandards/c069776_ISO_IEC_TR_19075-7_2017.zip.
[18] PQGL. http://pgql-lang.org.
[19] Sparksee. http://www.sparsity-technologies.com.
[20] Sqlg. http://www.sqlg.org.
[21] The graph and RDF benchmark reference. http://ldbcouncil.org.
[22] TigerGraph. https://www.tigergraph.com.
[23] VoltDB. https://www.voltdb.com.
[24] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt. G-core: A core for future graph query languages. In *SIGMOD '18*, pages 1421–1432, 2018.
[25] R. De Virgilio, A. Maccioni, and R. Torlone. Converting relational to graph databases. In *GRADES '13*, pages 1:1–1:6, 2013.
[26] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *WAIM'10*, pages 37–48, 2010.
[27] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR '15*, 2015.
[28] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD '18*, pages 1433–1445, 2018.
[29] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI'14*, pages 599–613, 2014.
[30] 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.
[31] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using vertica relational database. In *IEEE Big Data '15*, pages 1191–1200, 2015.
[32] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *SOCIALCOM '13*, pages 708–715, 2013.
[33] V. Kolomičenko, M. Svoboda, and I. H. Mlýnková. Experimental comparison of graph databases. In *IIWAS '13*, pages 115:115–115:124, 2013.
[34] M. Lissandrini, M. Brugnara, and Y. Velegrakis. Beyond macrobenchmarks: Microbenchmark-based graph database evaluation. *PVLDB*, 12(4):390–403, 2018.
[35] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.
[36] Y. A. Megid, N. El-Tazi, and A. Fahmy. Using functional dependencies in conversion of relational databases to graph databases. In *Database and Expert Systems Applications*, pages 350–357, 2018.
[37] O. Orel, S. Zakosek, and M. Baranovic. Property oriented relational-to-graph database conversion. *Automatika*, 57(3):836–845, 2016.
[38] S. Pradhan, S. Chakravarthy, and A. Telang. Modeling relational data as graphs for mining. In *COMAD '09*, 2009.
[39] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.
[40] B. A. Steer, A. Alnaimi, M. A. B. F. G. Lotz, F. Cuadrado, L. M. Vaquero, and J. Varvenne. Cytosm: Declarative property graph queries without data migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, pages 4:1–4:6, 2017.
[41] 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.
[42] Y. Tian, S. J. Tong, M. H. Pirahesh, W. Sun, E. L. Xu, and W. Zhao. Synergistic graph and SQL analytics Inside IBM Db2. *PVLDB*, 12(12), 2019.
[43] D. W. Wardani and J. Kiing. Semantic mapping relational to graph model. In *IC3INA '14*, pages 160–165, 2014.
[44] K. Xirogiannopoulos and A. Deshpande. Extracting and analyzing hidden graphs from relational databases. In *SIGMOD '17*, pages 897–912, 2017.
[45] K. Zhao and J. X. Yu. All-in-one: Graph processing in rdbmss revisited. In *SIGMOD '17*, pages 1165–1180, 2017.