KGFabric: A Scalable Knowledge Graph Warehouse for Enterprise Data Interconnection

Peng Yi*, Lei Liang*, Da Zhang, Yong Chen, Jinye Zhu, Xiangyu Liu, Kun Tang, Jialin Chen, Hao Lin, Leijie Qiu, Jun Zhou†
{rongyan.yp,leywar.liang,jianyu.zd,liqi.cy,zhujinye.zjy,liuxiangyu.lxy,tangkun.tk,bingchu.cjl,linhao.linh}
{leijie.qlj,jun.zhoujun}@antgroup.com
Ant Group

ABSTRACT

Based on the diversified application scenarios at Ant Group, we built the Ant Knowledge Graph Platform (AKGP). It has constructed numerous domain-specific knowledge graphs related to merchants, companies, accounts, products, and more. AKGP manages trillions of structured knowledge graphs, serving search, recommendation, risk control and other businesses. However, as the demand increasing for various workloads such as graph pattern matching, graph representation learning, and cross-domain knowledge reuse, the existing warehouse systems based on relational DBMS or graph databases are unable to meet the requirements. To address these issues, we propose KGFabric, an industrial-scale knowledge graph management system built on the distributed file system (DFS). KGFabric offers a nearline knowledge storage engine that utilizes a Semantic-enhanced Programmable Graph (SPG) model, which is compatible with the Labeled Property Graph (LPG) model. The data is persistently stored in DFS, such as HDFS, which leverages the POSIX file system API, making it suitable for deployment in multi-cloud environment at low cost. KGFabric provides a native graph-based and hybrid storage format that can serve as a shared backend for parallel graph computing systems, significantly accelerating the analysis of multi-workload. Additionally, KGFabric includes a graph fabric framework that minimizes data duplication and guarantees data security.

KGFabric is able to manage Peta-scale data and has supported graph fabric and analysis with over 100 billion relations at Ant Group. We conduct experiments on various datasets to evaluate the performance of KGFabric. Compared with popular relational DBMS and graph databases, the storage space for semantic relations is reduced by over 90%. The performance of graph fabric improves by 21× in real-world workloads. In multi-hop semantic graph analysis, KGFabric enhances performance by 100×.

PVLDB Reference Format:
Peng Yi, Lei Liang, Da Zhang, Yong Chen, Jinye Zhu, Xiangyu Liu, Kun Tang, Jialin Chen, Hao Lin, Leijie Qiu, Jun Zhou. KGFabric: A Scalable Knowledge Graph Warehouse for Enterprise Data Interconnection. PVLDB, 17(12): 3841 - 3854, 2024.
doi:10.14778/3685800.3685810

*Equal Contribution. †Corresponding Author.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685810

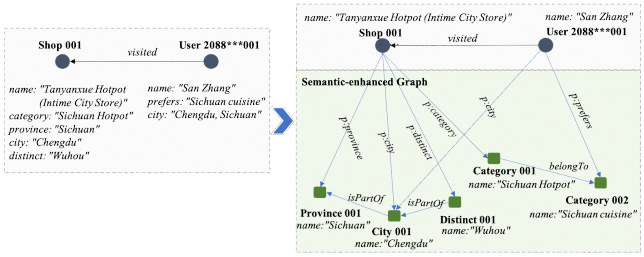

Figure 1: The semantic enhancement from LPG to SPG

1 INTRODUCTION

Knowledge Graph (KG) was introduced by Google in 2012, primarily for search system [27]. As a novel approach to data management, KG leverages the combination of graph structure and knowledge semantics to model, manage, reason, and make decisions about the real world. This method has been widely used in many industries [1, 44], especially in fields such as finance, public security, and medical care where enterprise data is rich and interconnected. With the diversification of scenarios, Industry-scale KGs face more challenges [1, 52] including knowledge acquisition and disambiguation, large-scale dynamic knowledge management, cross-domain knowledge reuse, data security and privacy, complex graph analysis, and graph representation learning.

AKGP manages a huge structured knowledge repository, which is constructed from two types of data sources: a large amount of structured data, such as user preference tags and transaction events, and a smaller volume of unstructured data, such as news articles. To process unstructured data, we use the knowledge extraction tool OneKE [30]. The generated structured knowledge is classified into three categories: entities, concepts [16, 41], and events. Entities represent objective facts of multi-dimensional structures, such as accounts, merchants, companies, and their associated properties and relationships. Correspondingly, concepts represent abstract classes, which are the induction of facts from the specific to the general. They serve as taxonomic representations of a set of facts, such as product categories, user groups, administrative divisions, etc. With continuous enhancement of semantics, more and more relationships are established between entities and concepts in different knowledge graphs. Figure 1 shows how this semantic enhancement increases the density and connectivity of graph data, facilitating the exploration of potential relationships between persons and products. Events can capture dynamic behavior and reflect the state of

things in different spatial and temporal intervals, such as industry events, and user behavior events.

Figure 2 shows KGs of specific domains managed by KGFabric in Ant Group. The slogan of KG is "things, not strings". These KGs utilize accurate domain types for entity and event property values, such as IndustryCategory, Brand, and more. There are also relationships between concepts such as hypernyms, hyponyms, combinations, and causality. By standardizing the types of property values, we enable connections across entities of different domain KGs through public concepts. This capability is particularly valuable for characterizing and gaining risk insights into long-tail merchants which may have sparse relationships within the graph structure. Establishing relationships between entities in different KGs is indeed an efficient way to achieve cross-domain data interconnection, such as the (*red*) connection between Company and Merchant in Figure 2. Furthermore, entity fusion techniques can be employed to implement a cross-domain data fabric. For instance, by identifying identical accounts, connections can be established between Black Market KG and Merchant KG.

Previously, we used ODPS [45] (i.e. relational warehouse of Alibaba Cloud platform) and graph databases like Neo4j [48], as the underlying architecture for managing KGs. However, these systems do not sufficiently meet the following needs. *(1) Cross-domain KG Fusion and Security*: Financial data is distributed in multiple domains and needs to support heterogeneous graph fusion. The fusion process requires privacy field discernment and encryption to ensure data security, which are lacking in existing graph engines. While Neo4j has proposed graph federation, it only supports the union of homogeneous entities and lacks data conflict resolution capability. ODPS can achieve data fusion through multi-table join, but its scalability and multi-hop analysis performance are bottlenecks. *(2) Semantic-enhanced Graph Storage and Computation:* The entity-centric basic relations and concept-centric semantic relations have different graph densities, as shown in Figure 13. LPG-based graph engines lack effective support for semantic models, resulting in the difficulty of achieving high-performance analysis of semantic graphs. Resource Description Framework (RDF) possesses strong semantic characteristics and has been extensively studied in academia as a solution for managing KGs. However, it has not widely applied in large-scale industries. We will discuss the limitations of related systems or solutions in Section 8. *(3) Schema Evolution*: As businesses grows, schema evolution is needed to enhance data connections. Graph engines that use LPG face challenges in ensuring data consistency between vertices and edges during updates. For example, transforming Person.city property into a relationship requires adding an entity City and a relationship Person-[locateAt]->City, while deleting Person.city.

KGFabric focuses on developing an industrial-scale knowledge graph management system in finance and multi-domain scenarios. The paper identifies the following as its main contributions.

- We propose a highly scalable KG storage engine based on DFS, which incorporates the SPG [54] model to extend semantic relationships between entities, events, and concepts. The engine offers a native graph-based storage and automates triple indexing to enable connections between entities and concepts, and utilizes multi-indexing and time slicing to optimize query performance in eventic graphs. Furthermore, our architecture employs an LSMTree-based design for handling streaming and batch updates.
- We introduce a distributed graph fabric framework which supports cross-tenant encryption and fusion of heterogeneous KGs. This framework enables the creation of virtual graph views and user-defined operators. In the process of merging relations and properties, conflicts are resolved by using linking and fuse operators. Generally, linking operator can be a rule expression or an NLP-based entity similarity algorithm, and fusion operator uses rule expressions.
- We present a shared backend for parallel graph analysis systems like GeaFlow [57] and GraphX [26], to support various KG workloads, including second-level latency OLAP and graph pattern matching [14, 17] with over 100 billion relations. The backend eliminates data shuffling and transformation overhead during the graph partition stage.

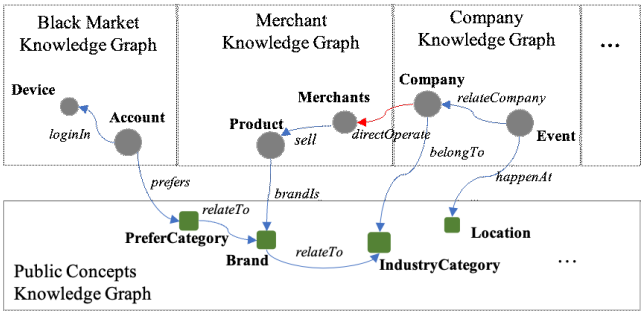

Figure 2: Multi-domain knowledge graphs in Ant Group

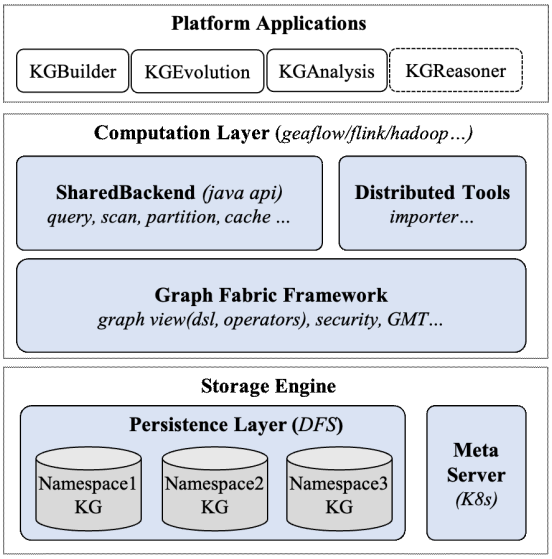

Figure 3: The architecture of KGFabric

The paper's organization is as follows: Section 2 outlines the system architecture and design. Sections 3 and 4 describe the data model and the storage engine. Section 5 introduces the cross-domain

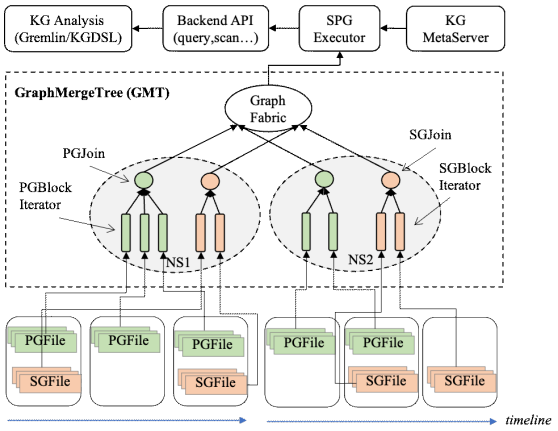

Figure 4: The query planning of KGFabric

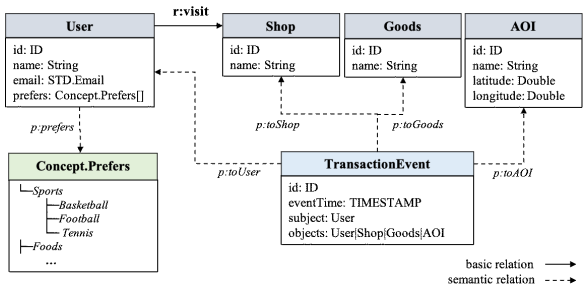

Figure 5: Data model of SPG

graph fabric framework. Section 6 discusses the solutions for various graph workloads and query optimizations. Section 7 presents the evaluation results, comparing KGFabric with Neo4j [48] and ODPS [45]. Sections 8 and 9 discuss related work and conclude the paper, addressing future research directions.

2 OVERVIEW

KGFabric facilitates the creation of a large-scale data fabric within semantic-enhanced graphs, enabling efficient management for enterprise data with complex connections, multi-source, dynamic, and temporal characteristics. The architecture of KGFabric, as shown in Figure 3, includes storage engine, graph fabric framework, backend and distributed tools for data reading/writing.

At the storage layer, The data from different domains is persistently stored in separate DFS clusters or directories. Data isolation and management are achieved through namespaces. MetaServer offers CRUD services specifically for metadata management, including schema, namespaces and task manager. It can be easily deployed on K8s-based cloud services such as SofaStack [63]. We provide directory-level versions and schema-level MVCC to validate data updates in pre-release environments, including scenarios such as dynamic graphs, semantic evolution. The features support task-level resolution of conflicts arising from read-write and write-write operations. We offer a nearline storage architecture based on LSMTree[40], which supports incremental updates of graph data, and leverages offload compaction [2] to optimize read amplification. To achieve low-latency random reading and high-throughput distributed scanning, KGFabric employs a block-based persistent storage architecture. This approach is similar to disk-based storage engines like RocksDB [19]. As shown in Figure 4, the persistence layer consists of two types of files: PGFile (Property Graph format File) and SGFile (Semantic Graph format File), which consist of multiple PGBlocks and SGBlocks. Each block stores the relations, properties, and indexes of a continuous range of vertexIDs. We utilize PGBlock Iterator and SGBlock Iterator to read the blocks, and merge different graph formats by PGJoin (Property Graph format Join) and SGJoin (Property Graph format Join). SPG Executor communicates with MetaServer and generates a multi-namespace fabric plan, called GMT (Graph Merge Tree). GMT is used to generate a complete graph data structure.

The shared backend offers graph-structured query or scan APIs. The APIs enable on-demand access to the data managed by KGFabric in parallel graph computing systems like GeaFlow [57], while supporting graph query languages such as Gremlin [61]. The query API is designed to support multi-hop KG OLAP, which involves traversing the graph step by step, starting from several vertices. It is particularly useful for ad-hoc visual graph analysis, such as anti-fraud cases. The scan API is employed for parallel-graph computing, like detecting cycle pattern. It allows for the specification of parameters like workerNumber and workerIndex, enabling the loading and processing of subgraph data on different workers. This capability supports edge-cut [43] and vertex-cut [25, 26] partition. To enhance performance, the backend provides configurable caching options, which include replica caching for meta data and LRU caching for data blocks. The importer is an incremental bulkload tool that can run on various big data platforms like Hadoop [32] and Flink [22].

KGFabric, a storage-computing separated architecture, is well-suited for nearline and offline management of graph data, providing scalability, high-throughput and cost-effectiveness. KGFabric also has following limitations and design compromises. *(1) Schema Constraints*: KGFabric enforces a strict schema for domain-specific KGs in finance, improving knowledge accuracy and storage compression. However, this approach is not friendly to schema-free KGs with dynamic properties and predicates. Some graph databases, such as Neo4j, support schema-free. *(2) Throughput and Latency Tradeoff*: Our nearline LSMTree-based [40] storage architecture supports maximum updates of over 100 billion graphs per day. The Shared-Backend supports parallel graph analysis systems, like GeaFlow, for multi-workload KG Analysis. However, a large number of real-time read and write operations may not be efficient in this architecture. *(3) Version Control*: KGFabric offers directory-level versions for centralized data management of streaming window and batch updates in DFS, providing cost-effectiveness and scalability. However, fine-grained transaction capabilities, available in graph engines like LiveGraph [71], are lacking.

3 DATA MODEL

The data model of KGFabric is SPG[1], which supports three types of graph model: Entity, Concept, and Event, as shown in Figure 5.

[1]SPG is an industrial semantic framework and has been jointly released by Ant Group and OpenKG at the CCKS conference on August, 2023

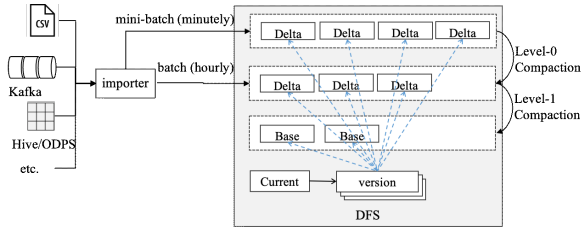

Figure 6: the LSMTree-based architecture on DFS

1 **Entity**: It is represented as *entityType* and *properties*, similar to the nodes in Neo4j. The *entityType* denotes the entity class, such as User, Shop, etc.
   1.1 *properties* consist of multiple *<name, value>* pairs. The *value* can be basic types such as integers, doubles, strings, or semantic types like standardized or concept class. It also supports List. For instance, the sports preferences of an user include basketball, tennis, etc. Unlike LPG, the semantic property will be converted into a relation by default. For instance, when the property values of User.email are identical between two entities of User type, semantic relations can be established between the two entities.
   1.2 *relation* is expressed as *<relationName, sourceType, targetType, properties>*. It is similar to the edges in Neo4j, connecting different entity instances.
2 **Concept**: Concepts [16] represent abstract collections of entities, such as user preference categories, which are relatively static and reusable. The concepts are represented as a hierarchical graph structure connected by predicates such as hypernyms and hyponyms, typically forming a tree structure. A concept instance consists of *conceptName* and *parent*. The *parent* indicates the hypernyms concept. It is possible to have duplicate *conceptNames*, but the paths generated by traversing hypernyms relationships to the root concept will not contain duplicates.
3 **Event**: As a special type of entity, event is represented as *<subject, objects, eventTime, properties>*, and it can support HyperEdge [21, 38]. The *eventTime* indicates the timestamp when the event occurs, that is usually used as a condition for query expressions. The *subject* and *objects* represent multiple elements associated with the event. For instance, TransactionEvent may be associated with entity objects such as User, Shop, Goods and AOI (Area of Interest).

4 STORAGE ENGINE

Instead of using Write Ahead Log (WAL) by online graph databases, we offer nearline LSMTree-based [40] storage architecture. We execute multiple import tasks to handle high throughput write requests, and employs offload compaction [2], which reduces the impact of serialization and IO operations on read performance. The storage layer of KGFabric implements a hybrid storage to support SPG models. Semantic graph storage adopts triple index [3] solution used in RDF storages such as Jena-TDB [55]. We also optimize massive storage redundancy and update efficiency of triple index. Property

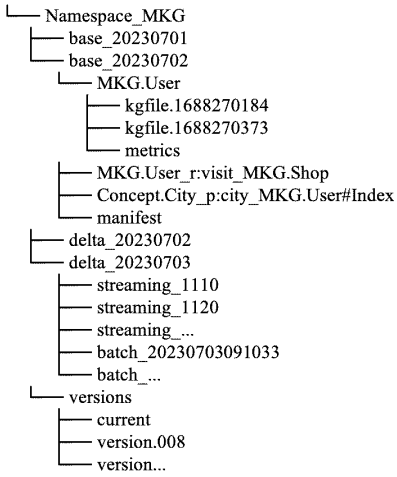

Figure 7: The data layout

graph storage employs a CSR(Compressed Sparse Row) solution, similar to LLAMA [42] and LiveGraph [71]. We also supports *supervertex* (i.e., vertex with a large number of neighbors) sharding and temporal graph.

4.1 LSMTree-based Architecture

Figure 6 shows the storage architecture in KGFabric, which consists of two layers in DFS: the base layer and the delta layer. The importer tasks can handle different data sources. The delta layer is divided into two levels: level-0 and level-1. Level-0 is primarily utilized for streaming updates, where data is received from message queues like Kafka. In this situation, data is persisted through minutely mini-batches. Periodic (e.g., Hourly) compaction is executed to compact the level-0 data into the level-1. For frequent mini-batch tasks at level-0, we also use tiering merging [40] strategy. Batch processing is employed for data obtained from Hive or ODPS. We merge tasks to control the number of delta files in the level-1. In Ant Group, more than 1,000 tasks per day are scheduled. The compaction strategy effectively controls IO amplification of query within 5×, while also keeping IO amplification of scan within 1.6×, due to the updating data size of importer tasks being <10% of the base.

KGFabric has a directory-level version manager, that can resolve mutual exclusion problems related to task-level read and write operations. Whenever a new data directory is generated, it creates a new version file called "version.$ID". This version file maintains a checkpoint pointer. By using a Timestamp, the corresponding version ID generates, which can access the data at any snapshot. The "current" file always keeps track of the latest version ID.

Figure 7 shows the data layout of KGFabric, which follows a directory tree structure.It consists of various types, as follows.

(1) *Namespace*: It isolates and manages graph data from different domains. The metadata of namespace is managed by MetaServer, including tenant information, DFS cluster details, directory names, etc.
(2) *RelationGroup (RG)*: RG is responsible for managing data groups. It includes grouping by entity or relation types and

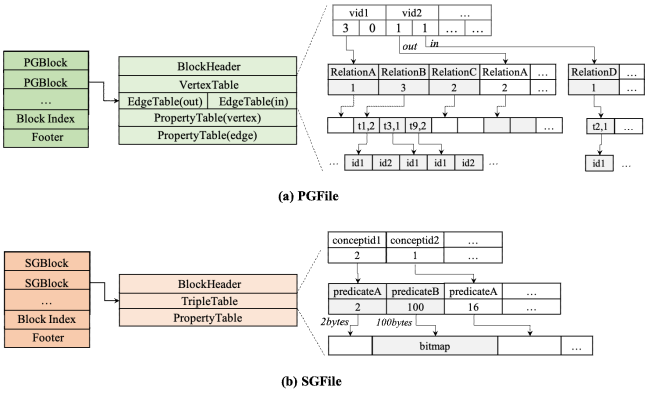

Figure 8: The hybrid storage format on DFS file

indexing slices of time range. RG is similar to Column Family [9], providing IO optimization. For example, the large-scale relation MKG.User-[visit]->MKG.Shop is configured as an independent RG to enhance data update and compaction efficiency. MKG.User stores properties and other relations related to entities of MKG.User. The "manifest" file records the metadata of RGs and the range of vertexIDs/<s, p, o> in KGFiles (i.e., PGFiles or SGFiles) in RGs.

(3) *base_%Y%m%d*: It manages periodical snapshots. The RG directory of static data in the base layer is only a soft link that points to the corresponding RG directory in the historical base layer, as it does not exist in the delta layer.
(4) *delta_%Y%m%d*: It manages incremental data for a specific period, typically with one directory created per day.
(5) *streaming_%H%M*: It manages streaming data, with mini-batch directories created at regular intervals, such as 10 minutes.
(6) *batch_$jobID*: It manages batch imported data by mapreduce or flink jobs.

4.2 Property Graph Storage

Property graphs is stored in PGFiles, which consists of PGBlocks, Block Index and Footer, as shown in Figure 8a. Footer is a fixed-length section that stores information such as format version. Block Index maintains the offset of PGBlocks in DFS file. PGBlocks are sorted by vertexID. To find the PGBlocks of the desired entity, we utilize a binary search on the block index. The block size varies according to data type and graph density, usually from 64KB to 1MB. For example, a smaller block size is set for a simple graph, and a larger block size is set for a dense graph. PGBlock consists of VertexTable, EdgeTable (incoming and outgoing), and PropertyTable (for vertices and edges). VertexTable and EdgeTable utilize a CSR format to compress temporal graph data. PropertyTable supports row or column storage, such as column storage for encoded semantic properties. PropertyTable includes a bit matrix to identify property cells with NULL value. BlockHeader records the offsets of different tables in the block. The block is the writing IO unit, while the tables act as the compression unit. The format supports on-demand reading of property and edge tables.

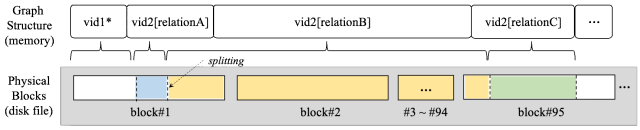

Figure 9: The high-degree vertices storage and processing

The incoming and outgoing edges from a *super-vertex* cannot be stored in one block. To address this, we propose splitting the edges of the *super-vertex* into multiple blocks. Block index records the range of *<s, p, o>* (equivalent to *<vertexID, relationType, targetID>* in LPG) for each block. This approach allows for precisely calculating the blocks in which different types of relations within a *super-vertex* are stored. Splitting boundary blocks is possible when converting physical blocks to memory graph structures. As shown in Figure 9, the data of relationB of vertexID2 store in physical blocks #1 to #95. The boundary blocks #1 and #95 will trigger splitting. This solution is beneficial for sequential writing, enhancing IO efficiency of query limits by relation types, and vertex-cut graph partition.

4.3 Semantic Graph Storage

Semantic graphs include Concept graphs and Eventic graphs. Concept graphs have three storage components, as follows.

❶ **Dictionary**. It maintains the mapping between ConceptName and ConceptID, storing them in distinct DFS files according to concept types. We use LogStructure [40] storage to support efficient updates. The structure of log records consists of actionFlag, ConceptID, nameSize, nameStr, and parentID fields. The actionFlag identifies add/delete/modify operations. ConceptID is sequentially encoded, and deleted IDs are not recycled. The nameSize and nameStr record the variable-length concept name, only storing the leaf (e.g., "Hangzhou"). The parentID records the ID of hypernyms concept, enabling the generation of complete paths (e.g., "China-Zhejiang-Hangzhou") by tracing to the root. Concept trees are typically characterized by small-scale and low-frequency updates, and have tens of million concept instances in Ant Group. Subtrees are loaded by concept type and persisted in batches to ensure the atomicity of ID encoding.

❷ **Semantic Property**. It stores only the ConceptID in PGBlock, which allows for linking to a unique concept instance. This approach reduces the storage cost compared to physical edges between entities and concepts. The storage space required for one ConceptID is typically 2-8 bytes. With varint compression, most ConceptIDs only require 2 bytes, whereas graph databases like Nebula [69] consume 30 bytes for each edges. Additionally, we can decode ConceptID from the dictionary and get the property value.

❸ **Triple Index**. It enables connections from concepts to entities and stores in SGFiles. As shown in Figure 8b, SGFile is comprised of SGBlocks. each SGBlock consists of TripleTable and PropertyTable. TripleTable stores many semantic relations, which can have properties such as *confidence score*. The concept graph being dense, we have implemented a bitmap index based on RBM (Roaring BitMap) [15]. This approach aims to enhance the compression rate and computing performance of the semantic graph. Figure 10 shows how the bitmap index is utilized through bitwise operations

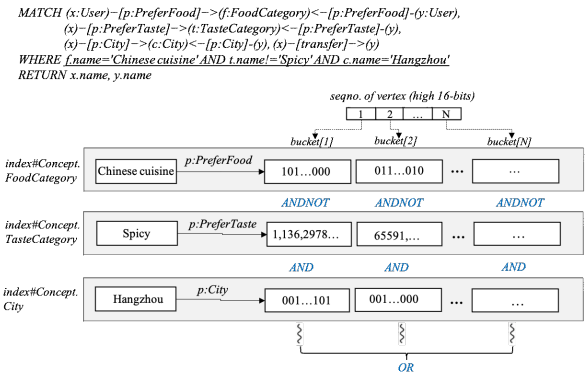

Figure 10: A combining-concepts query based on triple index and bitwise operations

such as AND/OR/NOT, along with buckets-based parallel computations. These operations help accelerate the complex graph queries of combining concepts conditions. It also supports graph partition techniques such as vertex-cut and *ghost* vertex (redundant target vertex in each worker). We store spo (Subject-Predicate-Object) and ops indexes. Semantic properties support the spo index, while RG supports predicates groups to achieve the effects of the pos and pso indexes. In industrial scenarios, the predicate (or relationName) is typically utilized as a query condition rather than a matching result. Therefore, we do not construct osp and sop indexes. In addition to reducing storage space, this approach improves index update efficiency and consistency. When a semantic property is updated, it triggers the update of triple indexes. Updating the ops index requires a read-modify-write (RMW) operation. For instance, when "*John-[p:prefers]->Football*" is changed to "*John-[p:prefers]->Tennis*", three operations are needed: (1) read relation (i.e., spo) "*John-[p:prefers]->Football*", (2) delete ops *<Football, p:prefers, John>*, and (3) add ops *<Tennis, p:prefers, John>*. We offer the BaseJoin mechanism, which uses a sort-merge join algorithm to convert RMW operations into sequential reads and writes. The mechanism improves the throughput of triple index updates and supports over 100 billion RMWs per day. However, if we use graph database, to ensure that query latency is not affected, the RMW operations are limited to less than 100 million per day.

Eventic graphs have two storage components: ❶ **Multi-index**, which is employed to facilitate the connections of types of entities to events, includes *<entityType1, p1, eventType>*, *<entityType2, p2, eventType>*, etc. ❷ **Time slice**. It enables queries based on windows and other timing diagrams, and TTL (Time-to-Live). The events and their indexes are divided into multiple slices based on eventTime. However, as the number of slices increases, the efficiency may decrease due to IO amplification. To avoid accessing invalid slices, we generate a bloom filter for each slice's related entities.

4.4 Updates and Versions

Schema updates may require data conversion. For instance, changing property types will trigger the construction of SGFiles. MetaServer provides schema-level MVCC to handle conflicts between adaptive schema updates and data import. Evolution jobs (i.e., E-Jobs) related

Table 1: Schema-level MVCC (* timestamp refers to job start status. I-Job and E-Job refers to importer and evolution job).

(a)

status	ST	DT
Initial	T0	T1
I-Job start	T0*	T2*
E-Job start	T3*	T1*
E-Job finish	T3	*T1*
I-Job finish	T0*<T3	
trigger E-Job'	T3*	T2*
E-Job finish	T3	T2

(b)

status	ST	DT
Initial	T0	T1
I-Job start	T0*	T2*
E-Job start	T3*	T1*
I-Job finish	T0	T2
E-Job finish		T1*<T2
trigger E-Job'	T3*	T2*
E-Job' finish	T3	T2

to the same property must be executed sequentially. To manage schema and data versions, each property is associated with two timestamp fields: ST and DT. ST records the schema version, while DT records the data version. Importer jobs (i.e., I-Jobs) update DT, and E-Jobs update ST. As shown in Table 1a, when the I-Job is executed, if there is an E-Job update ST, it will trigger the automatic execution of an E-Job for the imported data at time T2. Table 1b describes another situation. If the I-Job updates DT during the execution of the E-Job, it will also trigger another E-Job task for the data produced during (T1, T2]. Despite the low frequency of E-Job, it may be continuously triggered without successful completion, such as in streaming import scenarios. This could block new I-Jobs.

As concepts change can affect the graph distribution, such as mount rate (*number of semantic relations / number of properties*), it is necessary first to verify the impact in a pre-release environment. Unlike costly snapshot setting for rollback in Relational/Graph DBs, we utilize directory-level versions to achieve this. We use "current.online" file and "current.pre-release" file to point to different concept dictionaries and data directories, enabling isolation of the impact of concept changes on online graphs. When resulting in an increased mount rate, the ID record of current.online file is modified and points to the new concept dictionary.

5 GRAPH FABRIC

Traditionally, data fabric in relational warehouse and graph database is achieved through data copying between clusters. In contrast, we provide a graph fabric framework to reduce the cost of data copying. Firstly, source entities are selected from different namespaces. These source entities can generate a fusion entity (called FusedType) through link and fuse operators. When reading data, the graph structure of FusedType is constructed by GMT, that enables the connection of source entities.

5.1 Graph View

The graph view defines three components that facilitate cross-namespace data interconnection.

(1) **FusedType**: This component represents a fused entity type, which includes its corresponding source entity types (SourceType1, SourceType2, etc.) from different namespaces. FusedType incorporates the relations and properties from the source types. It is notable that FusedType is a virtual entity and not persistent data.
(2) **LinkOp**: Link operator is responsible for identifying similar entities from different KGs. It utilizes techniques such

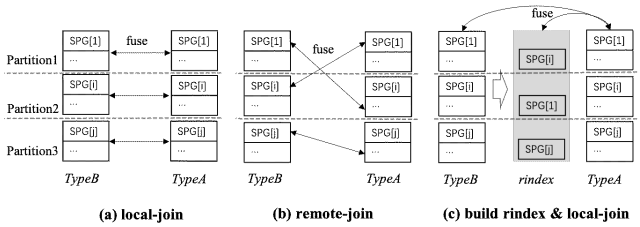

Figure 11: Distributed graph fabric. SPG[i] refers to the properties, basic and semantic relations of Entity[i]

as user-defined rules, and text/LBS(Location-Based Services)/vector similarity. To accomplish this, we can leverage search engines like ElasticSearch.

(3) **FuseOp**: Fuse operator merges two or more source entities into one entity. It handles the logic for combining properties and relations. The operator can define QLExpress [59] rules for merging and resolving conflicts among source entities.

We also provide a SQL-like method (similar to GQL [29] or TypeQL [67]) for creating graph view and supporting user-programmable operators through Java/Python interfaces, as follows.

```
CREATE LINK OPERATOR linkusers {
  main_class:"com.alipay.kgfabric.view.UserLinker"
  jar:"kgfabric-operator-linker-1.0.0-shaded.jar"
}
CREATE FUSE OPERATOR fuseusers [type=RULE params=e1,e2,e3] {
  $e3.name = ($e1.name != NULL) ? $e1.name : $e2.name
  $e3.prefers = union($e1.prefers, $e2.prefers)
  ($e3)-[transfer]->($e3) = copy(($e1)-[transfer]->($e1))
  ($e3)-[visit]->Shop = copy(($e2)-[visit]->(Shop))
}
CREATE GRAPH VIEW fabricview {
  (e1:SourceType1)-[transfer]->(e1) AS r1
  (e2:SourceType2)-[visit]->(Shop) AS r2
} WHERE {
  e1.age>30 and e2.age>30 and r1.amount>100 and r2.count>1
} WITH OPERATOR {
  linkusers(e1, e2, FusedType)
  fuseusers(e1, e2, FusedType)
}
```

5.2 Graph Merge Tree

Graph analysis system invokes query/scan API to access KGFabric. SPG Executor parses the query conditions and generates the GMT, which consists of graph join and graph fabric. GMT is a multi-way tree, and graph fabric stage may perform nested processing. Algorithm 1 shows the GMT's post-order traversal algorithm. If the entity is a FusedType, a recursive traversal is performed on that entity. Otherwise, a one-hop query is executed on the entity. Graph join is triggered by the query, involving PGJoin and SGJoin, which are responsible for merging hybrid formats in RGs of the delta and base layers. After collecting the entities and relations of all children, the rule of FuseOp is executed to generate the fused graph.

In query processing of the FuseOp, the FusedType can be used as the source or target vertex. However, it is notable that the specified fused entity C and its source entities A and B cannot appear in one graph structure. The in and out edges of A and B will become the edges of C. For example, r1:A->X and r2:V->B will be transformed to r1':C->X and r2': V->C. Similarly, the ring edge r:A->A or r:B->B will be transformed to r': C->C.

Algorithm 1 GMT-traversal

Input: V ← graph view, e ← query entity, x ← isMultiTenant(V, e)
Output: g ← fused graph

```
if e = FUSEDTYPE then
  G' ← ∅
  for s ∈ SourceTypes do
    g' ← GMT-traversal(V, s)
    G'.append(g')
  end for
  rule ← FuseOp(V)
  g ← rule(G')
else
  g ← query(e)
  if x = TRUE then
    encrypt(g)
  end if
end if
return g
```

GMT has two execution modes: Fuse on Read (FOR) and Fuse on Write (FOW). In FOW mode, the GMT is executed when the source data is updated. This mode provides low-latency querying but may result in data redundancy and does not support FuseOp updates. The FOR mode executes the GMT during the read operation, reducing storage redundancy and enabling FuseOp updates. In graph analysis applications (e.g., KG OLAP), the FOR mode is commonly used. the FOW mode is high-cost and inflexibility, so it is only used in ultra-low latency online scenarios.

The LinkOp includes two types: IDE (ID Equivalent) and UDL (User-Defined Link). IDE is a situation where multi-domain entities share the same primary key but have different properties or relationships. For example, the primary key of BMKG.User and MKG.User are based on the alipay account ID. As Figure 11a shows, in IDE situation, execute fuse operators by local-join. UDL refers to linking entities based on custom rules or algorithms, such as vector similarity. The UDL generates linkpairs (the linked entity IDs) randomly. As shown in Figure 11b, remote-join is performed through random reading across workers. To reduce I/O operations, configuring a block cache can be beneficial. In dense scenarios with numerous linkpairs, we employ a technique called rindex (Resorted Index) to convert remote-join to local-join, which can speed up the fusion. As shown in Figure 11c, the rindex only stores data of the source entities, such as TypeB fused with TypeA. It only resorts and stores the entity data of TypeB in the linkpairs.

5.3 Data Security

We provide permission control at the property level, allowing users to set permissions for reader, writer and manager. In order to create a graph view, users must apply for the read permissions. We have integrated AntPrivacy encryption interface into our SDK to enhance data security. This interface can identify fields that contain personal privacy information, such as certNo. Before graph fabric stage of GMT, the query graph is encrypted, as shown in Algorithm 1. The encryption key depends on the tenant of the reader. Consequently, the encryption keys for different domain KGs in the graph fabric stage remain consistent and do not interfere with the local-join

operation of FOR. This solution enables efficient cross-domain data fabric while ensuring the confidentiality and integrity of the data.

Even though the entity properties are encrypted, some graph analysis tasks, such as detecting cycle patterns and multi-hop indirect associations, can still be accomplished. The permission confirmation and decryption are completed in real-time interaction with AntPrivacy Service, and we strictly adhere to the minimization principle. For example, in the case of abnormal graph patterns, only the properties of Accounts that can assist the public security anti-fraud investigation will be allowed to be decrypted.

6 GRAPH ANALYSIS

We utilize a centralized approach based on DFS to manage knowledge graph data effectively, and provide a shared backend for graph analysis systems like GeaFlow [57], enabling support for graph analysis multi-workload in financial scenarios. Presently, we support two graph analysis languages, Gremlin [61] and KGDSL[2].

6.1 The Workloads

The applications and characteristics of various KG analysis workloads in Ant Group, as follows.

KG OLAP. It is commonly used in ad-hoc graph analysis scenarios, for example, ❶ **transaction tracing for anti-fraud cases**. With over 1 billion transaction events daily in Ant Group, we encounter frequent data updates and high timeliness requirements, requiring queries with second-level latency and traversal of 2 to 10 hops. The depth of transaction tracing significantly impacts the recall rate for target accounts, with the recall rate rising from 60.67% to 99.98% as the depth of transaction tracing increases from 2 to 10. ❷ **pairwise paths analysis**. Often used to discover indirect associations among entities such as merchants, companies, it is also applied in interpretability of link prediction. When identify an increasing number of entity pairs, batch calculations is required for optimization.

KG OLAP focuses on request latency. So we deploy resident tasks or services to support users initiate real-time query requests. For KG OLAP, we perform random read KGBlocks (i.e., PGBlocks or SGBlocks). A 1MB KGBlock can store over 100,000 compressed relations, handling most one-hop queries. The *super-vertex* processing supports truncation based on relation type, and parallel query of multiple neighbors.

GPM (Graph Pattern Matching). It is applied in complex graph analysis scenarios. For example, ❶ **detecting anomaly patterns for identifying risky merchants**. Mining cycle, many-to-one, one-to-many, and other graph patterns in transaction graphs can facilitate the detection of money laundering and fraudulent merchants. Cycle pattern is a significant and representative type of multi-hop GPM workloads. ❷ **semantic crowd analysis**. It refers to identifying target users based on user tags and the relationships between these tags. Tags are mapped to concepts, and relationships are established between these concepts. For example, when providing marketing recommendations for NBA-related products, inputting the NBA tag can lead to the discovery of tags associated with different concepts, such as basketball, etc., through combining-concepts and multi-hop reasoning. This helps to expand the recall of relevant users. Additionally, this approach supports cold start recommendation scenarios without any seed users.

In GPM tasks, a large number of vertices are involved in graph computation, necessitating high throughput. The *super-vertex* processing includes vertex-cut or properties filtering/truncation to enhance patterns recall. Presently, AKGP manages hundreds of tasks, with the largest graph scale reaching 100 billion. When the number of starting vertices reaches 1 billion, the QPS can reach tens of millions. GPM tasks frequently employ algorithms such as binary join and worst-case optimal join [14, 46], requiring high-throughput access via parallel graph computing. The runtime of GPM tasks is approximately within 10 hours, and the tasks are executed periodically.

6.2 Query Optimization

We offer a shared backend for different graph analysis tasks. It enables parallel graph computing through graph partitioning, utilizes a backend cache to enhance query throughput, optimizes graph operator performance through pushdown, and supports zero-copy serialization to reduce memory and CPU consumption.

Graph Partition. GPM task consists of two stages: partition and iteration. Partitioning involves loading a portion of KGBlocks from DFS into the local disk and memory of the workers to improve query throughput. KGBlocks represent the graph format and support Map-only distributed loading. In contrast to relational warehouses like ODPS [45] or Hive [35] that require data shuffling to build vertex set and edge set, KGBlock enables continuous storage of vertexID ranges, and allows for edge-cut [25, 26] partitioning, with better IO balance based on block size. In cases where a *super-vertex* has a block array, it can be divided across multiple workers to support vertex-cut [43] partitioning. By default, the business primary key is the entity ID, such as alipay account ID. To enhance graph storage and partitioning locality, we also support the use-defined ID generator, such as the 30-level S2CellID [28] of AOI center point, and graph embedding.

Backend cache. It consists of meta and data cache. The meta cache reduces access to the metafile of RGs and block index. In the case of KG OLAP, supported by resident tasks or services, the watcher triggers the update of the meta cache when the current version switches. In the case of GPM, the version is determined, and the meta cache is preloaded when the task starts. The data cache employs a dual-layer approach with disk and memory caches. The disk cache as a files cache cannot swap in and out, while the memory cache is an LRU cache with KGBlocks. Typically, *super-vertices* have higher hit rates in the cache.

Pushdown. It supports filter, aggregate and window operators. Filter involves entity/relation/property type filtering, which is pushed down to the leaf node (i.e., RGs) of the GMT. Window operator supports event and index reading based on time slices. Aggregate includes operations like sum, max, min, count, etc. Additionally, a metrics file records statistics for different properties for each block. The mechanisms are similar to the indexing functions in graph databases like JanusGraph [36].

[2]KGDSL is a GQL-like KG-reasoner language developed by Ant Group. It is suitable for symbolic representation of logic rules. The paper will not go into details.

Zero-copy serialization. It is employed to minimize IO and data transformation when executing GMT in graph analysis or compaction. The memory data structure aligns with the graph format on disk, reducing the need for serialization from disk to memory. This approach consumes less memory by avoiding using Java objects like Maps. Graph analysis systems, such as GeaFlow, support custom data structures for Vertex/Edge/Property and provide set/get interfaces, facilitating the implementation of this solution.

7 EVALUATION

The performance of KGFabric is evaluated by comparing it with popular relational DBMS ODPS [45] (which outperforms the open-source Hive [35]), graph database Neo4j [48], and KV Store RocksDB [19] in terms of graph storage, fabric, and analysis. RocksDB is used as a state backend in parallel computing engines like Flink [22] and GeaFlow, as well as a standalone storage engine in graph databases like Nebula [69]. KGFabric is mainly developed using Java, offering localized and distributed deployment. Distributed deployment relies on K8s and Hadoop components. BenchCases[12] shows the experimental cases.

7.1 Datasets

Table 2: Details about the used datasets.

Datasets	Type	#Entities	#Relations	#Concepts
LDBC-FinBench(SF1)	LPG	643K	6.09M	0
LDBC-FinBench(SF10)	LPG	6.06M	48.02M	0
LDBC-FinBench-X(SF1)	SPG	643K	8.09M	5433
LDBC-FinBench-X(SF10)	SPG	6.06M	65.52M	5433
AKG-A	SPG	0.8G	2.8G	5081
AKG-M	SPG	3.4G	26.1G	-
AKG-F	SPG	22G	110G	-

We select one public dataset, one custom dataset and three real-life datasets in Ant Group for the experiments, as shown in Table 2.

- *LDBC-FinBench* [58]: This dataset represents a financial scenario and consists of a heterogeneous temporal graph structure based on LPG schema. It includes 5 types of entities, 13 types of relations, and 64 types of properties. The dataset can be scaled by setting a scale factor, and for this experiment, two larger-scale datasets, SF1 and SF10, were generated. SF1 has approximately 600K entities and 6M relations, while SF10 has 6M entities and 48M relations.
- *LDBC-FinBench-X*: This dataset is a custom extension of the LDBC-FinBench dataset, which adds semantic relations between Company, Person, and Account to Concepts, increasing the number of relations by about 30%. We release the data generator and schema of LDBC-FinBench-X on GitHub[3]. The schema extension is based on SPG's business practices, encompassing 7 concept types (*Concept.Country, Concept.City, Concept.BusinessType, Concept.AccountType, Concept.AccountLevel, Concept.MediumType, Concept.RiskLevel*) and 3 standard types (*STD.PhoneNumber,STD.Email,STD.Url*). These semantic types are the object types of 13 properties, such as the transformation of the city property type of Person from string to *Concept.City*.

[3]http://github.com/jo3yzhu/ldbc_finbench_datagen_spg_extension

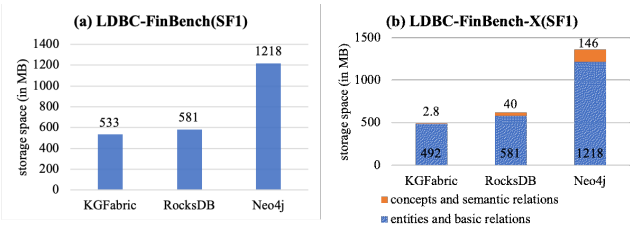

Figure 12: Storage space on various graph models and datasets

- *AKG-**: We used 3 datasets to evaluate real-world workloads. (1) *AKG-A*, the Alipay User KG, contains 2.8 billion semantic relations, such as User-[p:occupation]->Concept.Occupation. (2) *AKG-M*, the Merchant Risk KG, contains 3.4 billion entities and 26.1 billion relations, such as risk labels of merchants, goods ownership and trade relations. (3) *AKG-F*, a dataset fused by BlackMarket KG and Funds KG, with 22 billion entities and 110 billion relations, such as transaction and medium access relations within 60 days.

7.2 Storage Space

We utilize an NVMe SSD machine and conduct two storage experiments of synthetic datasets. In the experiments, both KGFabric and RocksDB are configured with a block size of 512KB.

LPG dataset. The storage space of LDBC-FinBench(SF1) utilized by KGFabric is 43.7%(533MB/1218MB) of that used by Neo4j and 91.7%(533MB/581MB) of that used by RocksDB, as shown in Figure 12a. This result clearly illustrates the benefits of KGFabric in its efficient storage of temporal graph structure and properties data. A CSR graph format enables more compact storage for graph structure. KGFabric achieves this by reducing the storage requirements for doubly linked list pointers in Neo4j. KGFabric employs compression for the VertexTable and EdgeTable. This experiment uses Deflate. Additionally, KGFabric follows a strict schema. In contrast, Neo4j is schema-free. KGFabric employs hybrid compression for storing various types of properties. For example, properties like createTime, are stored using column storage within each block. Properties of string and other types are stored using row storage.

SPG dataset. In the case of LDBC-FinBench-X(SF1), which includes the addition of concepts and semantic relations, the number of relations increases from 6.09M to 8.09M (+32.8%). When importing this dataset into Neo4j, the concept graph is converted into vertices and edges. As shown in Figure 12b, the storage overhead of KGFabric is only 1.9% (2.8MB/146MB) of that of Neo4j and 7% (2.8MB/40MB) of that of RocksDB. This storage efficiency is achieved through several techniques described in Section 4.2. Firstly, KGFabric employs encoding for concepts, using conceptID instead of storing strings. Secondly, KGFabric supports graph traversal from concepts to entities by generating a triple index based on bitmap compression. It is worth noting that the storage overhead growth rate is relatively small for this dataset because the semantic relations in the experimental do not have properties. In industrial scenarios, semantic relations may have additional properties, such as *confidence score*.

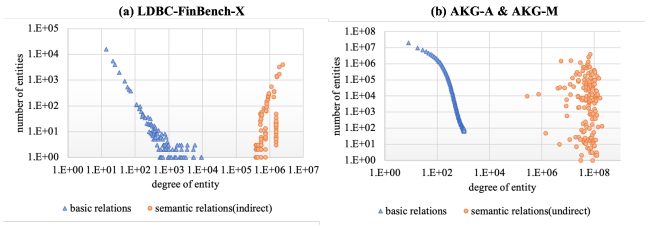

Figure 13: The graph density of various datasets, where semantic relation(indirect) refers to the relationship between entities connected to the same concept.

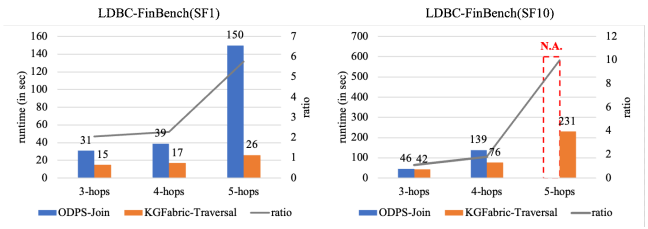

Figure 14: The performance of detecting cycle pattern on various datasets. (N.A. means runtime exceeds 1.5 hours)

real-life dataset of SPG. We collected the storage space performance of AKG-A from the DFS cluster in Ant Group. The 2.8 billion semantic relations occupy 2.73GB, i.e., ∼1 B/semantic relation, compared to LDBC-FinBench-X(SF1) which is 1.4 B/semantic relation (2.8MB/2M). The compression ratio of bitmap triple indexes is influenced by the data sparsity, for example, the relation STD.Email-[p:email]->Account of LDBC-FinBench-X(SF1) is sparser than Concept.Occupation-[p:occupation]->User of AKG-A.

7.3 Graph Analysis Performance

The graph analysis experiments consist of two synthetic workloads (i.e., *Exp 1* and *Exp 2*) and two real-world cases. Their applications are described in Section 6.1. In the experiments, we compare KGFabric backend on Geaflow (i.e., KGFabric-Traversal) with ODPS-Join, Neo4j-Cypher and RocksDB backend on Geaflow. ODPS-Join is implemented by SQL, while KGFabric-Traversal uses GeaFlow VC (vertex-centric) [43, 57] interface and KGDSL. A step-by-step binary join algorithm is used to ensure fairness. The experiments of synthetic workloads are conducted in a cluster with 8 workers, each having 8 vCPUs, 16GB memory and 128GB NVMe SSD. Due to Neo4j's lack of support for graph partitioning, we conduct the Neo4j experiments in a single worker.

Exp 1: pairwise paths analysis on LDBC-FinBench-X. The synthetic workload is used to evaluate the performance of multi-hop KG OLAP on semantic graphs. Randomly selecting about 1,000 vertices from the "Accounts" entity and mining pairwise paths within this vertex set (∼1M pairs).

KGFabric vs ODPS. Table 3 presents the results for two scales of semantic graphs. It can be observed that ODPS-Join's runtime for 6-hops exceeds 100× that of KGFabric-Traversal. On the larger dataset, ODPS-Join takes more than 24 hours to complete and generates output of over 400GB records. The performance of ODPS-Join drops significantly due to the presence of more intensive semantic relations. Figure 13a compares and analyzes the graph density of basic relations and semantic relations(indirect). The degree of entities based on semantic relations(indirect) ranges from 10^5 to 10^7, indicating a higher graph density. We optimize the multi-hop semantic graph query by implementing a multi-threaded parallel query on single worker. In extreme cases, each worker may access all triple indexes and *ghost* vertices. By triple indexes of high compression, we can improve the cache hit rate and reduce IO operations.

KGFabric vs Neo4j. In the SF1 dataset experiment, for 2-hops, Neo4j-Cypher takes 59 seconds, while KGFabric takes 5 seconds. For 4-hops, the latency of Neo4j-Cypher increases to over 1 hour due to reaching the physical memory limit and triggering pagecache swapping, caused by message generation of Expand Operator. In contrast, KGFabric takes only 11 seconds by utilizing combining-concepts joiner.

Exp 2: detecting cycle patterns on LDBC-FinBench. The synthetic workload is used to evaluate the performance of 3 to 5 hops GPM. Cycle pattern is also representative in LDBC-FinBench's Complex-read workloads [58].

KGFabric vs ODPS. Figure 14 illustrates the performance of KGFabric in cycle pattern matching compared to ODPS-Join on the datasets of LDBC-FinBench(SF1) and LDBC-FinBench(SF10). At 3 to 5 hops, KGFabric outperforms ODPS-Join by 2.06× to 5.77× on the dataset of 1 million relations. Furthermore, when the number of relations increases to 10 million and traverses 5 hops, KGFabric completes the task in 231s, whereas ODPS-Join's runtime exceeding 1.5 hours. The multi-join stages in ODPS-Join encounter difficulties in processing 6.3 billion records (57.16GB IO size) of intermediate data, which lead to timeouts. In contrast, KGFabric-Traversal partitions the graph data without requiring shuffling and sorting. Additionally, native graph format facilitates querying neighbors in both directions, reducing intermediate data and cross-worker messages. For instance, only one query is necessary to calculate all 2-hops cycles of a vertex. Moreover, KGFabric-Traversal offers backend cache and sequential IO, which replace random IO of queries.

We conduct the experiment of detecting 5-hops cycle pattern on 1 million relations in a Hive cluster with the same resources. Hive-Join takes 261s, while ODPS-Join takes 150s, indicating that ODPS-Join performs better. This shows the significance of comparing KGFabric-Traversal and ODPS-join.

KGFabric vs RocksDB. Graph analysis tasks often involve storing the vertex set and edge set in various backends, such as memory or RocksDB. In industrial scenarios, disk-based solutions like RocksDB are typically more suitable. For instance, in anti-fraud tasks, large amounts of graph data (10 billion to 100 billion) need to be loaded. Table 4 presents a performance comparison of each stage using three types of backends on GeaFlow. With KGFabric backend, the graph partition stage eliminates the overhead of data shuffle and format transformation. Conversely, ODPS-RocksDB requires 31.6s and 31.2s to convert relational data into vertex sets and edge sets, and then write them to RocksDB. During the iteration stages, the performance of KGFabric backend increases by 1×. In RocksDB, a one-hop query involves a range scan, and the presence of level-0 ssts

Table 3: The performance of pairwise paths analysis on semantic graph of LDBC-FinBench-X(SF1 and SF10).

	2-hops	4-hops	6-hops
#results(number of paths)	2.3K	1.9M	8.2G
ODPS-Join	7s	11s	13423s
KGFabric-Traversal	0.48s	0.94s	21.2s
#results(number of paths)	2.2K	14M	448.1G
ODPS-Join	10s	37s	>24h
KGFabric-Traversal	3.5s	9.1s	1317s

Table 4: The each stage performance of various backends on 3-hops cycle pattern.

backend type	stage1#graph partition		stage2#graph traversal			total
	shuffle	transform	iter-1	iter-2	iter-3	
ODPS-RocksDB	31.6s	31.2s	60s	9.7s	0.2s	132.7s
DFS-RocksDB	0	37.1s	53.1s	7.9s	0.2s	98.3s
KGFabric backend	0	0	35.8s	6.8s	0.1s	42.7s

Table 5: The graph analysis performance of various backends on real-world workloads.

	ODPS-RocksDB	KGFabric backend
detecting anomaly patterns	63.4+17.7 minutes	10 minutes
semantic crowd analysis	>24h	8.5 minutes

(3 ssts in the experiments) leads to read amplification. DFS-RocksDB is a solution that eliminates the data shuffle stage. Additionally, writing ordered data of KGFiles is more compatible with RocksDB.

Exp 3: two real-world workloads. In industrial scenarios, saving overhead of the *stage1#graph partition* is often more important, as the *stage1* loads the entire graph, while *stage2#graph iteration* traverses subgraph through constraining starting vertices and properties filter. As shown in Table 5, in a cluster with 100 workers, each having 8 vCPUs, 16GB memory and 128GB NVMe SSD, *detecting 3-hops anomaly patterns on AKG-M*, ODPS-RocksDB takes 63.4 minutes and 17.7 minutes in the partition and iteration stages. Switching to KGFabric backend can reduce the task's runtime to 10 minutes. The *semantic crowd analysis on AKG-A* involves a 3-hops query. The runtime of ODPS-RocksDB exceeds 24 hours, while KGFabric backend only takes 8.5 minutes in a cluster with 16 workers. As shown in Figure 13b, Many concepts in AKG-A have a degree of 10^8 with User entities. The edge-cut partitioning approach of ODPS-RocksDB can only compute high-degree concept on a single worker. Consistent with *Exp 1*, KGFabric's bitmap triple index and combining concept computation present significant advantages in the performance of large-scale semantic graph analysis.

7.4 Graph Fabric Scalability

We use synthetic datasets to evaluate the scalability and memory usage of the multi-graph fabric. We test the one-hop query FusedType on two types of LinkOP: IDE and UDL. The experimental environment is identical to that used in the graph analysis experiments of synthetic workloads. ODPS-MR (MapReduce) implements the vertex and edge multi-table fabric solution, which involves union, partition by, and group by startID/endID to aggregate relations between source entities.

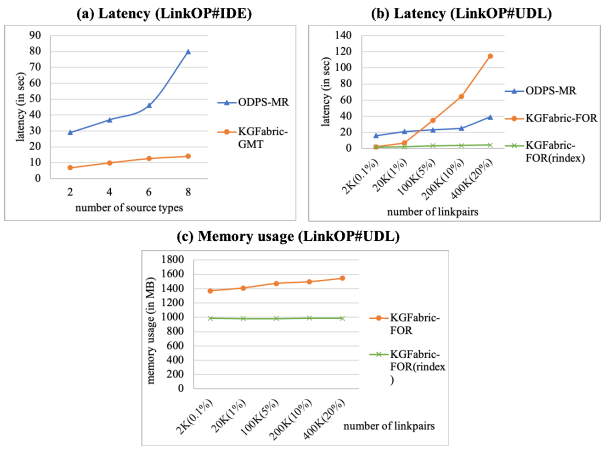

Figure 15: The scalability of fabric solutions, where linkpairs refer to the similar entity pairs generated by LinkOP#UDL.

Exp 1: LinkOP#IDE. It selects 2 million entities and 10 million relations from the LDBC-FinBench(SF10) dataset, evenly divided into 8 subgraphs. Each subgraph contains 1.3 million relations. As shown in Figure 15a, when the number of sources ranges from 2 to 8, the latency of ODPS-MR is 3.7× to 5.7× than that of KGFabric-GMT. Furthermore, as the number of sources increases, the latency of ODPS-MR grows more significantly. For instance, when the number of sources increases from 4 to 8, the latency of ODPS-MR increases by 116% (from 37s to 80s), whereas KGFabric-GMT only increases by 41% (from 9.9s to 14s). The main performance overhead of ODPS-MR is observed in the reduce stage. In contrast, KGFabric-GMT utilizes a Map-only approach, leveraging asynchronous IO and parallel multi-way join to improve local GMT performance, as shown in Figure 16.

Exp 2: LinkOP#UDL, including FOR and FOR-rindex. The experiment utilizes two datasets from the IDE experiment and randomly generates 2,000 to 400,000 linkpairs within 2 million entities. As shown in Figure 15b, the performance advantage of KGFabric-FOR becomes more evident as the number of linkpairs decreases. When the linkpairs are only 2,000, the FOR latency is 14% that of ODPS-MR and 133% that of FOR-rindex. Under fixed concurrency, the FOR latency approximately linearly increases with the number of linkpairs. For instance, when there are 100,000 linkpairs (covering 5% of the entities), the FOR latency exceeds that of ODPS-MR (34s > 23s). As the number of FOR remote-join random reads increases, the performance decline becomes more significant than batch processing. Conversely, FOR-rindex converts random IO into sequential IO. When the number of linkpairs increases to 400,000, the FOR latency is 27× that of FOR-rindex. The performance advantage of FOR-rindex becomes more pronounced as the number of linkpairs increases. The rindex mechanism is similar to ZOrder index [8, 18] utilized in OLAP systems such as ClickHouse. However, FOR-rindex has the disadvantage of storage redundancy and potential data delay issues caused by linkpairs updates.

Figure 15c illustrates the memory overhead of KGFabric solutions. FOR-rindex exhibits a more stable memory overhead based

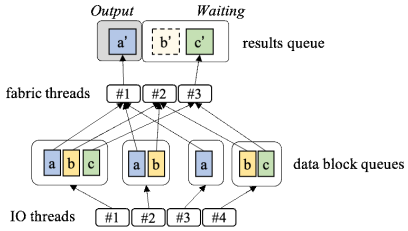

Figure 16: Asynchronous IO and multi-threaded computing of local GMT

on scanning. The relative memory overhead of FOR increases by 400MB to 600MB because random queries lead to block cache growth. Once the upper limit of the cache size is reached (set to 1GB in the experiment), the memory overhead tends to stabilize.

Exp 3: real-world workloads on AKG-F. AKG-F utilizes LinkOP#IDE. We conduct tests on the scan FusedType entities and their one-hop graph performance in a cluster with 800 vCPUs and 1600 GB memory. The results include 7.9 billion entities and 71.3 billion relations. ODPS-MR takes 27.58 hours, whereas KGFabric-GMT only takes 1.28 hours (i.e., 21× improvement), as explained in *Exp 1*.

In multi-hop analysis, FusedType entities may participate in each superstep, and the fabric process of FOR may be executed multiple times. The drawback of FOW is the issue of space redundancy and data latency (27.58 hours of ODPS-MR). KGFabric-GMT, even with FOR solution, can ensure OLAP latency and data timeliness. GMT first accurately locates blocks of multi-namespace, reducing IO operations. Secondly, it improves performance by utilizing a sort-merge to reduce data shuffling and a multi-threading model as shown in Figure 16. We evaluate *transaction tracing for anti-fraud cases on AKG-F* in a cluster with 75 workers, each having 8 vCPUs, 16GB memory and 128 GB NVMe SSD. The range of traversal is 2 to 10 hops. The P50 and P99 query latency are 4.45s and 33.41s.

8 RELATED WORK

Relational warehouses like Hive [35] and ODPS [45] offer low-cost and scalability advantages. However, They have limited support for multi-scenario data reuse and complex graph analysis. DataLake [8, 47] addresses slow updates and flexibility, but does not solve the challenges of complex connections. KG combines graph structure and semantics to model real-world entities, facts, and their relationships, enabling enterprises to organize and manage data more effectively. Cambridge Semantic [44] has introduced a new data management paradigm based on knowledge graphs. Gartner predicts that Data Fabric [24] based on knowledge graph technology will be the next-generation data architecture in 2021.

LPG and RDF are the popular knowledge graph data management solutions. RDF has strong semantic characteristics, utilizing RDFS/OWL [13, 33] as the schema language and SPARQL [33] as the query language. However, RDF is complex and expensive in terms of business understanding and modeling. Various RDF stores [3, 10, 20, 51, 55, 68, 70] are available. RDF-3X [51], Jena-TDB [55], and Trident [68] use the triple index, that is a six-permutation index to process <s,p,o> in any query conditions. However, six-permutation index suffers from storage redundancy and index update consistency issues. KGFabric optimizes the triple index costs and updates efficiency. It also incorporates bitwise-based solutions like TripleBit [70], BitMat [10], and Columnar [31] for triple index and parallel computing. We use RBM [15] for index compression in sparse scenarios. Jena [55] and Gradoop [37] rely on hbase, while Trident [68] stores data in DFS to support distributed applications. LPG [4, 5], characterized by a strong structure, has widely used Graph DBMSs [6, 60] like Neo4j [48], Tugraph-db [66], Neptune [50], Nebula [69], and ByteGraph [39], providing online graph query and write. Our SPG model extends concepts and events on LPG model, and provides nearline large-scale read and write. Its storage layer is a native graph-based distributed engine on DFS. Neptune [11, 50] and Trinity/Trinity.RDF [62] support RDF and LPG models, but they do not support model transformation on a copy of stored data. Some research [7, 34] has explored the conversion of LPG and RDF at only the model layer. KGFabric provides a hybrid format for SPG model, and offers version manager for semantic transformation.

GraphX [26] and GeaFlow [57] are LPG-based and parallel computing systems for graph analysis that are dependencies of KGFabric. Unlike solutions based on RocksDB, memory, ODPS or Hive, KGFabric provides a shared backend to optimize semantic graph partitions and reduce the overhead of cross-platform data shuffling and transformation.

Data islands and interconnections are crucial and challenging aspects of enterprise data management. Neo4j [49] and Apollo [65] propose graph federation, which only provides simple federated services, and cannot ensure data security. Federated graph learning [23] is a machine learning application solution that does not address data management problems. Our graph fabric framework supports cross-namespace encryption and user programmability. Stardog [64] and Ontotext [53] have introduced the Enterprise Knowledge Graph platform, aiming to address data interconnection challenges through knowledge graph, similar to KGFabric. However, their utilization of RDF/OWL and RocksDB, differs from our SPG model and native graph-based storage.

9 CONCLUSION

KGFabric is an enterprise knowledge graph management system developed by Ant Group specifically for multi-domain scenarios. It focuses on integrating large-scale data into standardized and semantic knowledge by employing an SPG model and a DFS-based storage engine. The system provides a programmable distributed framework that enables graph fabric across different domains of KGs, and offers practical solutions to reduce data duplication and ensure data security. Experimental results indicate that KGFabric outperforms Neo4j and ODPS in graph storage, analysis, and fabric in various scenarios such as property graphs and semantic graphs.

Our important plans include vector storage and universal storage format compatibility. Data vectorization can enhance the locality of graph storage and optimize the performance of cross-domain graph fabric and reasoning. It also can support KG+LLM [56]. To minimize cross-platform data migration, we will be compatible with open formats like Parquet and ORC.

REFERENCES

[1] Bilal Abu-Salih. 2021. Domain-specific knowledge graphs: A survey. *Journal of Network and Computer Applications* 185 (2021), 103076.

[2] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.

[3] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2022. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal* (2022), 1–26.

[4] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[5] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W Hare, Jan Hidders, Victor E Lee, Bei Li, Leonid Libkin, Wim Martens, et al. 2021. Pg-keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 2423–2436.

[6] Renzo Angles and Claudio Gutierrez. 2018. An introduction to graph data management. *Graph Data Management: Fundamental Issues and Recent Developments* (2018), 1–32.

[7] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. 2020. Mapping RDF databases to property graph databases. *IEEE Access* 8 (2020), 86091–86110.

[8] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[9] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 583–598.

[10] Medha Atre, Jagannathan Srinivasan, and James A Hendler. 2009. BitMat: A main memory RDF triple store. *Tetherless World Constellation, Rensselar Plytehcnic Institute, Troy NY* (2009).

[11] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. 2018. Amazon Neptune: Graph Data Management in the Cloud.. In *ISWC (P&D/Industry/BlueSky)*.

[12] BenchCases 2023. *BenchCases*. Retrieved November 24, 2023 from http://github.com/FessGo/knowledge-graph-warehouse-bench

[13] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. 2011. OWLIM: A family of scalable semantic repositories. *Semantic Web* 2, 1 (2011), 33–42.

[14] Sarra Bouhenni, Said Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. 2021. A survey on distributed graph pattern matching in massive graphs. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–35.

[15] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with roaring bitmaps. *Software: practice and experience* 46, 5 (2016), 709–719.

[16] Concept 2023. *Concept*. Retrieved November 11, 2023 from http://en.wikipedia.org/wiki/Concept

[17] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data*. 2246–2258.

[18] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282* (2020).

[19] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[20] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35, 1 (2012), 3–8.

[21] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. 2019. Hypergraph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 3558–3565.

[22] Flink 2023. *Apache Flink — Stateful Computations over Data Streams ...* Retrieved October 10, 2023 from http://flink.apache.org/

[23] Xingbo Fu, Binchi Zhang, Yushun Dong, Chen Chen, and Jundong Li. 2022. Federated graph machine learning: A survey of concepts, techniques, and applications. *ACM SIGKDD Explorations Newsletter* 24, 2 (2022), 32–47.

[24] Gartner: Data Fabric 2023. *Using Data Fabric Architecture to Modernize Data Integration*. Retrieved October 25, 2023 from https://www.gartner.com/smarterwithgartner/data-fabric-architecture-is-key-to-modernizing-data-management-and-integration

[25] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[26] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. {GraphX}: Graph processing in a distributed dataflow framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 599–613.

[27] Google Knowledge Graph 2024. *Google Knowledge Graph*. Retrieved February 20, 2024 from http://en.wikipedia.org/wiki/Google_Knowledge_Graph

[28] Google S2 2023. *S2 Geometry | S2Geometry*. Retrieved November 7, 2023 from http://s2geometry.io/

[29] GQL 2023. *Graph Query Language GQL*. Retrieved October 30, 2023 from http://www.gqlstandards.org

[30] Honghao Gui, Lin Yuan, Hongbin Ye, Ningyu Zhang, Mengshu Sun, Lei Liang, and Huajun Chen. 2024. IEPile: Unearthing Large-Scale Schema-Based Information Extraction Corpus. *CoRR* abs/2402.14710 (2024). https://doi.org/10.48550/ARXIV.2402.14710 arXiv:2402.14710

[31] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar storage and list-based processing for graph database management systems. *arXiv preprint arXiv:2103.02284* (2021).

[32] Hadoop 2023. *Apache Hadoop*. Retrieved October 25, 2023 from http://hadoop.apache.org

[33] Olaf Hartig. 2017. RDF* and SPARQL*: An Alternative Approach to Annotate Statements in RDF.. In *ISWC (Posters, Demos & Industry Tracks)*.

[34] Olaf Hartig and Bryan Thompson. 2014. Foundations of an alternative approach to reification in RDF. *arXiv preprint arXiv:1406.3399* (2014).

[35] Hive 2023. *Apache Hive*. Retrieved October 11, 2023 from http://hive.apache.org

[36] JanusGraph Index 2023. *Indexing for Better Performance*. Retrieved October 11, 2023 from http://docs.janusgraph.org/schema/index-management/index-performance

[37] Martin Junghanns, André Petermann, Kevin Gómez, and Erhard Rahm. 2015. Gradoop: Scalable graph data management and analytics with hadoop. *arXiv preprint arXiv:1506.00548* (2015).

[38] Geon Lee, Minyoung Choe, and Kijung Shin. 2021. How do hyperedges overlap in real-world hypergraphs?-patterns, measures, and generators. In *Proceedings of the web conference 2021*. 3396–3407.

[39] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. 2022. ByteGraph: a high-performance distributed graph database in ByteDance. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3306–3318.

[40] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[41] Xin Lv, Lei Hou, Juanzi Li, and Zhiyuan Liu. 2018. Differentiating concepts and instances for knowledge graph embedding. *arXiv preprint arXiv:1811.04588* (2018).

[42] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 363–374.

[43] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[44] Sean Martin, Ben Szekely, and Dean Allemang. 2021. *The Rise of the Knowledge Graph*. O'Reilly Media, Incorporated.

[45] MaxCompute 2023. *MaxCompute: Conduct Petabyte-Scale Data Warehousing*. Retrieved October 11, 2023 from https://www.alibabacloud.com/product/maxcompute

[46] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076* (2019).

[47] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. 2019. Data lake management: challenges and opportunities. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1986–1989.

[48] Neo4j 2023. *Neo4j Graph Database Analytics | Graph Database*. Retrieved October 20, 2023 from http://neo4j.com

[49] Neo4j Fabric 2023. *Sharding Graph Data with Neo4j Fabric - Developer Guides*. Retrieved October 25, 2023 from http://neo4j.com/developer/neo4j-fabric-sharding/

[50] Nepture 2023. *Amazon Neptune - Fully Managed Graph Database - AWS*. Retrieved October 11, 2023 from http://aws.amazon.com/neptune

[51] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19 (2010), 91–113.

[52] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. 2019. Industry-scale Knowledge Graphs: Lessons and Challenges: Five diverse technology companies show how it's done. *Queue* 17, 2 (2019), 48–75.

[53] Ontotext GraphDB 2023. *Ontotext*. Retrieved October 11, 2023 from http://www.ontotext.com/products/graphdb

[54] OpenSPG 2023. . Retrieved October 30, 2023 from http://github.com/OpenSPG/openspg

[55] Alisdair Owens, Andy Seaborne, Nick Gibbins, et al. 2008. Clustered TDB: A clustered triple store for Jena. (2008).

[56] Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. 2023. Unifying Large Language Models and Knowledge Graphs: A Roadmap. *arXiv preprint arXiv:2306.08302* (2023).

[57] Zhenxuan Pan, Tao Wu, Qingwen Zhao, Qiang Zhou, Zhiwei Peng, Jiefeng Li, Qi Zhang, Guanyu Feng, and Xiaowei Zhu. 2023. GeaFlow: A Graph Extended and Accelerated Dataflow System. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[58] Shipeng Qi, Heng Lin, Zhihui Guo, Gábor Szárnyas, Bing Tong, Yan Zhou, Bin Yang, Jiansong Zhang, Zheng Wang, Youren Shen, et al. 2023. The LDBC Financial Benchmark. *arXiv preprint arXiv:2306.15975* (2023).

[59] QLExpress 2023. *alibaba/QLExpress.* Retrieved October 11, 2023 from http://github.com/alibaba/QLExpress

[60] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.".

[61] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages.* 1–10.

[62] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.* 505–516.

[63] SOFAStack 2023. *Scalable Open Financial Architecture Stack-Alibaba Cloud.* Retrieved October 25, 2023 from http://www.alibabacloud.com/product/sofastack

[64] Stardog 2023. *Stardog: The Enterprise Knowledge Graph Platform.* Retrieved October 11, 2023 from http://stardog.com

[65] Patrick Stünkel, Ole von Bargen, Adrian Rutle, and Yngve Lamo. 2020. GraphQL federation: A model-based approach. (2020).

[66] Tugraph 2023. *TuGraph the high-performance graph database.* Retrieved October 11, 2023 from http://tugraph.antgroup.com

[67] TypeQL 2023. *vaticle/typeql - the polymorphic query language of TypeDB.* Retrieved October 20, 2023 from https://github.com/vaticle/typeql

[68] Jacopo Urbani and Ceriel Jacobs. 2020. Adaptive low-level storage of very large knowledge graphs. In *Proceedings of The Web Conference 2020.* 1761–1772.

[69] Min Wu, Xinglu Yi, Hui Yu, Yu Liu, and Yujue Wang. 2022. Nebula Graph: An open source distributed graph database. *arXiv preprint arXiv:2206.07278* (2022).

[70] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a fast and compact system for large scale RDF data. *Proceedings of the VLDB Endowment* 6, 7 (2013), 517–528.

[71] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2019. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *arXiv preprint arXiv:1910.05773* (2019).