A Distributed Graph Database System to Query Unstructured Data in Big Graph

Zihao Zhao\textsuperscript{1,2}, Zhihong Shen\textsuperscript{1}, Mingjie Tang\textsuperscript{1}, Chuan Hu\textsuperscript{1,2}, Yuanchun Zhou\textsuperscript{1,2}
\{zhaozihao,bluejoe\}@cnic.cn, tangrock@gmail.com,\{huchuan,zye\}@cnic.cn
\textsuperscript{1}Computer Network Information Center, Chinese Academy of Sciences
\textsuperscript{2}University of Chinese Academy of Sciences
Beijing, China

ABSTRACT
Unstructured data, such as images and videos, have grown significantly. The interconnected unstructured data can be viewed as the properties of nodes in graphs. End users usually query graph data and unstructured data together in different real-world applications. Some systems and techniques are proposed to meet such demands. However, most of the previous work executes various tasks in different systems and loses the possibility to optimize such queries in one engine. In this work, we build a native graph database, namely PandaDB, to support querying unstructured data in the graph. We first introduce CypherPlus, a query language to enable users to express complex graph queries to understand the semantic of unstructured data. Next, we develop a cost model and related query optimization techniques to speed up the unstructured data processing as well as the graph query processing. In addition, we optimize the data storage and index to speed up the query processing in a distributed setting. The PandaDB extends the graph database Neo4j implementation and provides the open-source version for commercial use in the cloud. The results show PandaDB can support a large scale of unstructured data query processing in a graph, e.g., more than a billion unstructured data items. We also like to share the best practices while deploying the system into real applications.

PVLDB Reference Format:
Zihao Zhao\textsuperscript{1,2}, Zhihong Shen\textsuperscript{1}, Mingjie Tang\textsuperscript{1}, Chuan Hu\textsuperscript{1,2}, Yuanchun Zhou\textsuperscript{1,2}. A Distributed Graph Database System to Query Unstructured Data in Big Graph. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/graphcco/pandadb-v0.1.

1 INTRODUCTION
Graphs are ubiquitous in various application domains, e.g., social networks, road networks, biological networks, and communication networks\cite{15}. The data of these applications can be viewed as graphs, where the nodes (a.k.a. vertexes) and the relationships (a.k.a. edges) have relational and non-relational properties (a.k.a. attributes)\cite{23, 33}. End users would prefer to issue queries for the graphs’ topology, as well as the data associated with the nodes and the relationships of the graph together.

Take the Figure 1 as an example, individual (e.g., Michael Jordan) and related context information (e.g., NBA Chicago Bulls) are represented as nodes in this graph. Then, the relationships between individuals (e.g., Michael Jordan works for Chicago Bulls) are viewed as the edges. In addition, the property of node (e.g., n\textsubscript{1}) in Figure 1 can be structured (birthday or name of Michael) or unstructured data (pictures, videos of Michael). End users usually initialize some queries to understand the data as following:

Example 1.1. Graph data related queries in Figure 1.

\begin{itemize}
  \item \textbf{Q1:} What is the color of Michael Jordan’s favorite cat?
  \item \textbf{Q2:} What jersey number did Michael Jordan’s teammates wear at Bulls?
  \item \textbf{Q3:} Whether Kerr (Michael Jordan’s former teammate) is the same person as the Gold State Warrior’s coach Steven Kerr?
\end{itemize}

To answer such queries (i.e., Q\textsubscript{2}), traditionally we at first find items with the name of Michael Jordan from the database. Then, we get the Michael’s teammates at Bulls via the teammate relationship in the database. Next, we fetch the corresponding teammates’ photos from the file system and gets the jersey numbers based on image information extraction models. Finally, we return the basketball jersey numbers for Michael Jordan’s teammate. As a result, developers often have to comprise multiple systems and runtime together. This gives rise to a number of issues such as managing the complexities of data representation, resource scheduling, and performance tuning across multiple systems. Therefore, an unified and native graph querying engine by considering unstructured and graph data together is urgent in real applications.

In addition, we are facing multiple scenarios related graph and unstructured data query processing as listed below.

(1) \textbf{Fraudulent cash-out detection:} Credit card cash-out is attractive for investments or business operations, which are considered unlawful if exceeding a certain amount. Specifically, some credit cardholders want to obtain cash through transactions, and the merchant receives the funds after transaction settlement by the acquires, then pays the funds back to the credit cardholder, charging the handling fee. In xxx company, we take the transaction among users as an edge, and each user as an vertex in the graph. Meanwhile, each transaction related user signature is stored for further analysis. Thus, we identify the possible cash-out groups from the built graph, when we find dense connected subgraph and share the similar signature.
Heath insurance evaluation: Heath insurance evaluation process need to consider the applicant and related family member’s health historic situation together. For example, the high pressure, heart disease history, and cancer of parents would influence the final insurance cost for individual greatly. In xx insurance company, we build a graph based on the lineage among insurance holder, then store these insurance payment claims documents as the unstructured data in graph. We then compute the insurance cost based on the applicant’s related family members’ insurance claim data.

In this work, we aim to build a native graph database to support unstructured data processing based on the following properties. (a) An extended query language to help users to query the unstructured data content in a graph. (b) A way to understand the semantic information of unstructured data with fast response time. (c) An efficient storage system to manage structured and unstructured data in a big graph with billions of nodes and TB of unstructured data.

The major contributions of this work are listed as below:

1. Data model and query semantic: We define the semantic and query operators for querying the content of unstructured data in a graph. A new query language called as CypherPlus is proposed. This facilitates the graph query language to meet the description and query requirements of unstructured data without significant syntax changes.

2. Query optimization: We construct a model to formalize the queries processing cost related to unstructured data in the graph and develop an optimizing algorithm to optimize the logical query plan. Then, we optimize the database execution runtime by designing a service protocol when the query involves an AI model.

3. Optimized data storage and indexing: We optimize the physical storage of graph databases for supporting unstructured data management and develop a new index to speed up the queries of unstructured data.

4. Distributed graph database system: Based on the design mentioned above, a distributed graph database system, PandaDB, is implemented and tested for large scale of data.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 formalizes the data model and gives the query language. Section 4 provides an system framework of PandaDB. Section 5 discusses the optimization of unstructured data queries. Section 6 gives more details about the data indexing and storage. The experiment results are presented in Section 7, and the conclusion is presented in Section 8.

2 RELATED WORK

Graph database and processing systems [42, 43] have developed rapidly, flourished in graph query and large-scale graph data management[3, 44, 52]. For example, Neo4j[39], and JanusGraph[26] are the widely adopted graph management systems for the cloud and on-premise usage, and focus on the querying and management of graph data[1, 2, 6].

Different from structured data, users want to know the semantic information of unstructured data (e.g., text, photo, or video). For example, the plate number in the photo of a vehicle, the vehicle administration needs to find all cars with plate numbers starting with 123xxx. To the best of our knowledge, the primary commercial products do not support the querying of unstructured data in big graph[5, 25, 26, 36, 39]. In contrast to many existing systems that deal with batch-oriented iterative graph processing, such as Pregel[34], PowerGraph[55], GraphX[16], and Gemini[55], PandaDB preserves the well-formed data model of the existing graph database research, and the extended declarative language allows users to understand the semantic of unstructured data.

Multimedia retrieval systems support the querying and management of the content of unstructured data. However, most of works are usually designed for a single data type and a specific retrieval propose[8, 10, 21, 41, 47], such as face recognition[9, 48] or audio speech recognition[46]. In database community, the system
at first pre-process the multimedia data and then offers content-based searching in an offline manner. Multimedia retrieval system is widely used in different applications, but do not consider the graph data processing in most of cases.

Collaborative retrieval systems are usually built on the tools-chain-based system to support collaborative queries on structured data and unstructured data[50]. A collaborative query is decomposed into several sub-queries on different modules. Usually, a vector search engine is built for vector similarity search[12, 28, 53] and a database system is prepared for structured data management. In addition, the unstructured data analysis service is used to extract the feature vectors. Then a data pipeline is built to connect these components together. Because data and related computation are distributed in different systems, the consistency and correctness between unstructured data and the content will take many resources to be maintained. More importantly, the decoupled system framework loses the opportunity to optimize the workflow from beginning to end. Therefore, a mechanism and language for querying structured and unstructured data on the graph is needed.

3 DATA MODEL AND SEMANTICS

In this section, we formally define the property graph, then we introduce the extension to support the unstructured data processing in the property graph.

Table 1: Summary of notational conventions

| Concept                     | Notation | Set notation |
|-----------------------------|----------|--------------|
| Property keys               | k        | \( K \)      |
| Sub-property keys           | sk       | S\( K \)     |
| Relationship identifiers    | r        | \( R \)      |
| Node labels                 | l        | L            |
| Relationship types          | t        | \( T \)      |
| Property values             | v        | V            |
| Sub-property values         | sv       | S\( V \)     |
| (Semantic information)      | ud       | S\( ud \)    |
| Sub-property extraction function | \( \phi \) | S\( \phi \) |

3.1 Property Graph Data Model

In graph database community, data are typically represented as a property graph [3, 42, 43], in which nodes and relationships can have a set of properties. Every entity is represented as a node(a.k.a. vertex), identified by a unique identifier, having label(s) indicating its type or role. The attributes of the entity are called properties of the node. The relationship(a.k.a. edge) describes the association between entities. A graph includes infinite nodes, the nodes are connected by relationships. A relationship starts from a node(origin node), ends at a node(target node). The category of the entity is taken as the node’s label. A node could have more than one label. We give the formal specification of the property graph data model as [3]. Let \( L \) and \( T \) be countable sets of node labels and relationship types. A property graph is a tuple \( G = < N, R, K, src, tgt, \lambda, \tau > \) where:

- \( N \) is a finite subset of \( \mathcal{N} \), whose elements are referred to as the nodes of \( G \).
- \( R \) is a finite subset of \( \mathcal{R} \), whose elements are referred to as the relationships of \( G \).
- \( K \) is a finite subset of \( \mathcal{K} \), whose elements are referred to as the properties of \( N \) and \( R \).
- \( src: R \rightarrow N \) is a function that maps each relationship to its source node.
- \( tgt: N \rightarrow N \) is a function that maps each relationship to its target node.
- \( \nu: (N \cup R) \times \mathcal{K} \rightarrow V \) is a finite partial function that maps an identifier and a property key to a value.
- \( \lambda: N \rightarrow L \) is a function that maps each node id to a finite set of labels.
- \( \tau: R \rightarrow T \) is a function that maps each relationship identifier to a relationship type.

Take the Figure 1 as an example, it is formally represented model as a graph \( G = < N, R, K, src, tgt, \lambda, \tau > \):

- \( N = \{ n_1, ..., n_8 \} \);
- \( R = \{ r_1, ..., r_8 \} \);
- \( src = \{ r_1 \mapsto n_1, r_2 \mapsto n_1, r_3 \mapsto n_1, r_7 \mapsto n_2, r_8 \mapsto n_1 \} \);
- \( tgt = \{ r_1 \mapsto n_2, r_2 \mapsto n_3, r_3 \mapsto n_4, r_7 \mapsto n_7, r_8 \mapsto n_5 \} \);
- \( \lambda(n_1) = \lambda(n_2) = \lambda(n_3) = \lambda(n_4) = \lambda(n_5) = \lambda(n_6) = \lambda(n_7) = \lambda(n_8) = \{ \text{Person} \} \);
- \( \lambda(n_2) = \lambda(n_3) = \lambda(n_5) = \{ \text{Team} \} \);
- \( \lambda(n_2) = \lambda(n_3) = \{ \text{Organization} \} \);
- \( \lambda(n_4, \text{name}) = \text{Michael Jordan} \), ..., \( \lambda(n_8, \text{photo}) = < \text{image} > \);
- \( \tau(r_1) = \{ \text{workFor} \}, \tau(r_2) = \{ \text{hasPet} \}, \tau(r_3) = \{ \text{coachOf} \}, \tau(r_4) = \{ \text{teamMate} \}, \tau(r_5) = \{ \text{belongTo} \} \).

3.2 Graph Querying Language

Cypher[14] is a standard graph query language that allows high-level and declarative programming for various graph operations, including graph traversal, pattern matching, and sampling. The following query statements show how to create and query data via Cypher for Figure1.

```
-- Q1: Create two nodes and a relationship.
CREATE (jordan:Person{name: 'Michael Jordan'})
CREATE (scott:Person{name: 'Scott Pippen'})
CREATE (jordan)-[:teamMate]->(scott);

-- Q2: Query John's friend's name.
MATCH (jordan)-[:teamMate]->(n)
WHERE jordan.name='Michael Jordan'
RETURN n.name;
```

Q1 creates two nodes and builds a relationship, then two nodes are labeled with Person, with the name 'Michael Jordan' and 'Scott Pippen', respectively. Q2 retrieves the teamMate relationship starts from the node with name 'Micheal Jordan' and get the related nodes' name property.
3.3 PandaDB Extension

3.3.1 Unstructured Content Representation. The properties of nodes in a graph can be unstructured and structured data. In this work, we majorly focus on how to improve the query processing for the unstructured data since structured data processing is well developed in the current state-of-art system. At first, we deem the semantic information of data as the sub-property. For example, in terms of the node $n_1$ in Figure 1, the name and photo are the properties of $n_1$. The printed number of jersey is the sub-property of the photo. Thus, an unstructured data item can have multiple sub-properties. For example, the jersey number and human facial feature (e.g., color, hair and eyebrow) in $n_1$. photo are regarded as different sub-properties of Node $n_1$. We formalize the sub-property definition as following:

**Definition 3.1. Sub-property:** is the semantic information in unstructured data, that is

$$<\text{data item}> \rightarrow \text{subProperty} \equiv <\text{semantic information}>$$

**Example 3.1.** The semantic information of $n_1$’s photo in Figure 1 are represented as following ways:
- $n_1$.photo $\rightarrow$ jerseyNumber $=$ 23
- $n_1$.photo $\rightarrow$ face $=$ <feature_vector>

The list of sub-properties is pre-defined by the users, and it could be extended.

3.3.2 Sub-property Acquisition and Filtering. For the acquisition of semantic information of unstructured property, we introduce the sub-property extraction function $\phi$:

**Definition 3.2. Sub-property extraction function $\phi$:** A finite partial function that maps a sub-property key to a sub-property value (semantic information) as following:

$$\phi : (N \cup R) \times K \times SK \rightarrow SV, Sem \subset SV$$

$$\forall so \in Sem, \exists ud \in Sud$$

$$\text{where } so = \phi(ud, sk)$$

(1)

Consider the nodes in Figure 1, the name and the photo are the properties, and the face, jerseyNumber and animal are the sub-property keys. The sub-property extraction in Figure 1 could be expressed as follow ways:

- $\phi(n_1, \text{photo, jerseyNumber}) = 23$
- $\phi(n_1, \text{photo, face}) = <$feature_vector$>$
- $\phi(n_3, \text{photo, animal}) = \text{cat}$
- $\phi(n_6, \text{photo, face}) = <$feature_vector$>$
- $\phi(n_8, \text{photo, face}) = <$feature_vector$>$

Overall, a property graph including unstructured data is a tuple

$$UG = <G, SK, \phi>$$

where:

- $G$ is a property graph, whose property could be unstructured data.
- $SK$ is a finite set, whose elements are referred to as the sub-property key of $UG$.
- $\phi$ is a function set, items of it are used to extract sub-property value from unstructured data

3.3.3 Query Language. Cypher is a declarative graph query language developed in Neo4j [14]. The three mostly used clauses in Cypher are MATCH, WHERE, RETURN. The MATCH expresses the graph pattern to match, WHERE adds constraints to a pattern, RETURN defines what to return in the query. The constraints to a pattern are usually the property value of nodes in the query graph pattern. In Figure 1, if we want to query the name of Michael Jordan’s teammate, the Cypher statement would be:

```cypher
MATCH (n: Person)-[: hasMate]-(m: Person)
WHERE n.name = 'Michael Jordan'
RETURN m.name;
```

To query unstructured data in the property graph, we develop CypherPlus to include new develop function: **Literal Function, Sub-property Extractor, and Logical Comparison Symbols**.

**Literal Functions** create the unstructured property in a graph from a specific source. For example, BLOB, fromURL(), BLOB, fromFile() and BLOB, fromBytes(), these functions are supplied by PandaDB.

**Sub-property Extractor** is the semantic symbol of sub-property extraction function. It obtains the specific sub-property value from the data item. The users define how to extract a specific sub-property from unstructured data. **Logical Comparison Symbol** offers a series of symbols as Table 2 to support logical comparison between sub-properties. According to predefined rules, these symbols are considered UDFs(User Defined Function) that compare logical relationships between specified semantic information. For example, when $\sim$ is used to compare face information, the similarity of two facial feature vectors is calculated.

**Table 2: Logical comparison symbols of unstructured data**

| Symbol | Description | Example |
|--------|-------------|---------|
| ::     | The similarity between x and y. | x:y = 0.7 |
| ~:     | Is x similar to y. | x:y = true |
| !:     | Is x not similar to y. | x:y = false |
| <:     | Is x contained in y. | x:<y = true |
| =>     | Is y contained in x. | x>y = false |

**Example 3.2.** We give the three graph queries for Figure 1 as following. Note that the native clauses of Cypher are in blue color, while the extensions of CypherPlus are in red color.

```cypher
-- Q1: What are the jersey numbers of Michael Jordan’s teammates?
MATCH (n: Person)-[: hasMate]-(m: Person)
WHERE n.name = 'Michael Jordan'
RETURN m.jerseyNumber;

-- Q2: Is Michael Jordan’s pet a Cat?
MATCH (n: Person)-[: hasPet]-(m: Pet)
WHERE n.name = 'Michael Jordan'
RETURN m.animal = 'cat';

-- Q3: Which Golden State Warrior’s coach Steven Kerr?
MATCH (n1: Person)-[: hasMate]-(n4: Person), (n7: Person)-[: isCoachOf]-(n6: Team)
WHERE n1.name = 'Michael Jordan'
AND n4.name = 'Kerr'
AND n6.name = 'Golden State Warriors'
AND n7.name = 'Steven Kerr'
RETURN n4.photo $\sim$ n7.photo $\sim$ face;
```
We adopt the native graph technology in this work as Figure 2. The query parser, an execution engine, and an optimization algorithm are introduced at first, followed by the data storage and index to support efficiently querying structured and unstructured data. Finally, the AI server is proposed to the execution runtime to understand the semantic information of unstructured data.

4.1 Query Plan Optimization

We modify the parser of Cypher to understand and parse the semantic of CypherPlus. In general, the execution plan of PandaDB is executed linearly one by one following a conventional model outlined by the Volcano Optimizer Generator[19]. The query plan optimization applies standard rule-based optimizations, and includes constant folding, predicate pushdown, projection pruning, and other rules. For example, to support query the properties of graph nodes, predicates of the property filtering operations are pushed down to the storage layer [32]. This makes full use of the index in the storage layer.

As we know, the unstructured data semantic understanding always involves AI model inference and computation, and this is time-consuming in the real application. Therefore, PandaDB estimates the cost of unstructured data operations based on the proposed model and develop an optimizing algorithm to optimize the corresponding query plan. More details are introduced in Section 5.

4.2 Execution Operator

PandaDB adopts the execution engine from Neo4j†. A query is decomposed into different operators, and these operators are combined into a tree-like structure called execution plan. In this work, we introduce a series of new operators as Table 3 to create data source, extract the sub-property and compare the similarity.

In addition, we provide the user define function (UDF) for end users to specific their own way to understand the semantic of unstructured data. Thus, the UDF can be any format of AI-model. For example, users define a sub-property named as face. This represents the facial features of the individual photo. Next, our system can ingest the UDF (e.g., a face recognition model) to extract the facial features from the corresponding photos. AI models often have a strict requirement of the running environment, a high-performance GPU, a specific version of the dependent package. It is essential to deploy AI models away from the kernel of a database to make their running environments not affect each other. We presents a general interactive protocol (namely AIPM) between database kernel and AI models. Once a query obtains the semantic information from the AI model, the query engine sends an AIPM-request to get the extracted information. The server receives the request and extracts the computable pattern using the model corresponding to the service asynchronously. When the database query engine receives the extracted information, it caches the result and returns it to the user.

4.3 Data Storage

Graph storage is classified as non-native and native graph storage in the database community. For the non-native store, the graph storage comes from an outside source, such as a relational or NoSQL database. These databases store nodes and relationships of the graph without considering the topological, which may end up far apart in actual storage.

In this work, we opt for the native graph storage⁷. The data is kept in store files for the native graph engine. Each file contains data for a specific part of the graph, such as nodes, relationships, node-related labels, and properties. Therefore, a graph database has native processing capabilities when it uses an index-free adjacency list, and each node directly references its adjacent node, acting as a micro-index for all nearby node. Index-free adjacency is cheaper and more efficient because query times are proportional to the amount of the graph visited. We store relationship data as first-class entities. More details are presented in Section 6.

5 LOGICAL PLAN OPTIMIZATION

This section introduces the procedure to generate the plan for the graph query processing, then formalizes a new approach to improve the query execution performance based on the newly proposed algorithm.

---

†https://neo4j.com/docs/cypher-manual/current/execution-plans/

⁷https://neo4j.com/developer/kb/understanding-data-on-disk/
5.1 Query Plan Generation

As introduced before, the design of CypherPlus is motivated by Cypher[14], XPath[30] and SPARQL[22]. Given a query statement, the plan-generator generates the query plan based on the following steps: (a) Parses the query statement into an AST(Abstract Syntax Tree), checks the semantics, collects together different path matches and predicates. (b) Builds a query graph representation of the query statement. (c) Deals with the clauses and finds the optimal operator order. (d) Translates the optimal plan into the database API methods for data access. Therefore, a query is decomposed into a series of operators, each of which implements a specific piece of work.

In general, the query planning in PandaDB is optimized based on the IDP algorithm[an improved dynamic algorithm][38, 45] based on the corresponding cost model[20]. In this work, we extend this cost model and related algorithm to support the unstructured data processing.

These operators are combined into a tree-like structure (namely query plan tree, QPT). Each operator in the execution plan is represented as a node in the QPT. The execution starts at leaf nodes (usually AllNodeScan or NodeScanByLabel), and ends at the root node (usually Projection). The details of the basic query operator based on neo4j can be found in link ‡. The query optimization in this work focuses on step (c) as mentioned above. It re-organizes the operators to find an optimal plan with less computation cost. For an operator, its execution time depends on the data size of its input and its own characteristics. Most of existing optimization methods mainly focus on graph structure matching and structured property filters.

Consider the query statement in Figure 3, it queries the Michael Jordan’s pet cat’s name. The parsed operators include a structured property filter (Prop Filter1, filtering the data by the condition ‘Michael Jordan’), an unstructured property filter (Prop Filter2, making sure the pet is a cat), then an expanding operator to find relevant nodes by node’s relationships (to make sure the relationship between the two nodes). Also, there are some necessary related algebra operations like Projection and Join. Figure 3 shows three possible query plans to get the same queried results. The difference between the three plans lies in the relative order between the operators. The plan (a) executes the two filters and joins the results, then expands on the result. The plan (b) expands based on Property Filter1 and joins with the results of Property Filter2. The plan (c) executes the sub-property filter at last. However, the query execution time would differ.

For plan (a) and plan (b), the Prop Filter2 filters the photos of all nodes in the database. However, in plan (c) it filters the output of the Join operator. When the Prop Filter2 is much slower than other operators, plan (c) will have the shortest execution time than others.

### Table 3: Details about the unstructured data operators

| Operator                 | Arguments                                      | Description                                      |
|--------------------------|------------------------------------------------|-------------------------------------------------|
| createFromSource()       | URL or file path or binary content              | Create a BLOB from the source.                  |
| extract()                | BLOB item & sub-property name                  | Extract the sub-property (semantic information)  |
| compareAsSet()           | Two sets of semantic information               | Compare the similarity of the semantic information in the sets. |

Suppose there are 100 units of data, the Prop Filter1 takes 1 second to filter a unit of data, while the Prop Filter2 takes 100 seconds. Then the plan (a) and (b) takes 10000s to execute the Prop Filter2, while plan (c) takes only 100s. Because in plan (a) and (b), Prop Filter2 needs to filter all the data in the dataset, while in plan (c) it only needs to filter the output of Join. The fewer data the Prop Filter2 to filter, the less time the whole plan takes, if the Prop Filter2 is slower than other operators. While in real-world applications, it is difficult to judge the semantic filters’ speed from experience, the query plan would be more complex. Our system needs to optimize the query plan to obtain a fast execution plan by consider processing unstructured data in the graph database.

5.2 Logical Plan Optimization For Unstructured Data Querying

Traditional technologies apply cost-based methods to optimize the query. While they focus on graph structure filters and structured property filters, failing to estimate the cost of unstructured property filters. Compared with structured property filters, index and cache have a greater impact on the performance of unstructured property filters. So it would be not efficient to apply traditional cost model to estimate the cost of unstructured data filters. PandaDB applies the cost-based optimization (CBO) to speed up the query processing. It introduces a new method to calculate the expected speed of an unstructured property filter, then optimize the query plan by a greedy strategy.

The system updates the record of speed of an unstructured property filter (i.e. \( \sigma_p \)) after it is invoked. The speed of \( \sigma_p \) after it is invoked for the \( i \)th time could be calculated by the following formula:

\[
\nu_i(\sigma_p) = \begin{cases} 
\frac{\text{cost} + \frac{|T|}{k}}{1 - \nu_{i-1}(\sigma_p) + k \cdot \text{cost} / |T|} & \text{if } i = 1 \\
\nu_i(\sigma_p) & \text{if } i > 1
\end{cases}
\]

The \( \nu_i \) is taken as the expected speed of this filter when it was invoked for the next time, namely:

\[
\mathbb{E}(\nu_{i+1}(\sigma_p)|\nu_i(\sigma_p)) = \nu_i(\sigma_p)
\]

The cost is the consuming time for it to process the data in this query, and \(|T|\) is the size of the input. The \( k \) is a factor to adjust the model, the greater the \( k \), the more sensitive the model. It means the model tends to estimate the speed of a filter according to the latter performance. In the applications where unstructured data change fast, the \( k \) should be greater. Instead, the \( k \) should be smaller.

Based on the designs mentioned above, Definition 5.1 formalizes the cost model as following.

**Definition 5.1.** Given the the input table \( T \), the cost of an unstructured property operator could be estimated as follow:

\[
\mathbb{E}(\text{cost}(\sigma_p)) = \mathbb{E}(\nu_{i+1}(\sigma_p)|\nu_i(\sigma_p)) \cdot \mathbb{E}(|T|)
\]
Running example. For giving an example for the Algorithm 1, Figure 4 give a query statement and its query graph. The figure shows the PlanTable, Cand, and $T_{best}$ step-by-step.

Step1: The table is initialized with the plans that offer the fastest node access. This query does not specify the label of nodes, so the table could only obtain the nodes by plain \texttt{AllNodeScan}. The filter operations and projection are added into \texttt{Cand}. There are only two possible path to expand: $n1 \rightarrow n3$ and $n3 \leftarrow n1$. The former means to start from $n1$, expand by the out-relationship, the latter means to start from $n3$, expand by the in-relationship. They are added into the \texttt{Cand}. Supposed that the filter by name is the best candidate in \texttt{Cand}, it is inserted into the \texttt{PlanTable}. This operation covers the \texttt{AllNodeScan} of $n1$, so the \texttt{AllNodeScan} is removed.

Step2: The two expand operations could be joined with the filter operation. Suppose the first in \texttt{Cand} is the best candidate, insert it into the \texttt{PlanTable}. The $(n1 \rightarrow n3)$ is represented as $T_{n1 \rightarrow n3}$. The result covers the plain \texttt{AllNodeScan} of $n3$, so it is removed from the \texttt{PlanTable}. Then goes to Step3, the only candidate left is
the projection, insert it into the PlanTable. The final query plan is shown in the PlanTable of Step 4. It is the algebra representation of the query plan shown in Figure 3 (b).

**Complexity analysis.** The greedy procedure (lines 6-8) starts with n plans and removes at least one plan at every step. So it is repeated at most n times, where n is the count of nodes in the query graph Q. The complexity of estimating the cost of an unstructured property filter is \( O(1) \). Then, assuming that canJoin utilizes the Union-Find data structure for disjoint sets, the complexity of the entire algorithm becomes \( O(n^2) \).

6 DATA STORAGE AND INDEXING

In this section, we first introduce how the graph structure data and property data (including structured data and unstructured data) are stored in PandaDB. Then, we motivate the newly developed indexing to speed up the query processing for unstructured data in a graph.

6.1 Support Unstructured Data Storage In Graph

PandaDB modifies the storage of Neo4j [39] to support the unstructured data management in the graph. Neo4j stores the nodes and relationships of a graph in files, that is, the Nodestore, Relationshipstore, Propertystore and Labelstore. They keep the node, relationship, key/value properties of nodes and label of graph, respectively. Figure 5 lists the related data storage format. Nodestore uses the nextRelId and nextPropId to store the physical address of relationship and property for the corresponding node. Similarly, Relationshipstore stores the address of startNodeld and endNodeld, where startNodeld and endNodeld are the related nodes of this relationship. Therefore, we can get the relationship of nodes based on the stored address and vice versa. More details can be found in the link3.

Properties are stored as a double-linked list of property records, each holding a key and value and pointing to the next property. For example, propBlock is used to store the content of the property in binary format. Originally, users store the unstructured data in the

block propBlock as a byte-array. However, this way can not support streaming data reading and writing. Then the related IO performance degrades significantly when storing unstructured objects here. In this work, we modify the format of property and introduce the binary large object (BLOB) as a new datatype to store the unstructured data. From the bottom of Figure 5, the metadata (i.e., length, mime type, and id) of BLOB are stored in the last 28.3 bytes. For those BLOBs under 10kB, the binary content is stored in another file, like a long string and array storage. For those over 10kB, storing it into a native file will influence the performance, because the BLOBs will be fully loaded into the memory. Besides, when there are too many BLOBs stored in native files, the meta data would take much space. So we adopt HBase to manage the BLOBs.

Overall, PandaDB stores unstructured data in the following ways: (1) Treat the unstructured property as a BLOB. (2) Store the metadata and literal content of the BLOB, respectively. (3) The metadata (including length, mime type, and id) of BLOB are kept in the property store file, as shown in Figure 5. (4) For those BLOB whose literal value is less than 10kB, store it in the same method as long strings. (5) For those exceeds 10kB, store them in the BLOBValueManager based on HBase. The BLOBValueManager organizes and manages BLOB in a BLOB-table, which has n columns. In a row of the BLOB-table, each column stores a BLOB literal value. The location of a BLOB could be calculated by its Id by the following formula, where |column| means the count of the columns in HBase:

\[
\text{row}_\text{key}(\text{BLOB}) = \text{id}(\text{BLOB}) / |\text{column}|
\]

\[
\text{column}_\text{key}(\text{BLOB}) = \text{id}(\text{BLOB}) \% |\text{column}|
\]

The BLOBValueManager could quickly locate a BLOB by its id, as shown in Figure 5. Besides, the transmission of BLOB between BLOBValueManager and Query Engine is streaming.

6.2 Semantic Information Cache and Indexing

We observed that on a 56 CPU cores server, the average time for extracting facial features from a face image by OpenCV is approximately 0.3s. It is expensive to extract the semantic information repeatedly. Thus, PandaDB caches and indexes the semantic information to accelerate the acquisition and retrieval of semantic information. Intuitively, different features compose different semantic spaces. So we partition the semantic information of objects by their semantic spaces, then build indexes on different semantic spaces, respectively.

6.2.1 Semantic Information Extracting and Caching. PandaDB extracts the semantic information and stores it via the Key-Value format, where the key is composed by the id of the unstructured
data item and the serial number of AI model. The AI model extracts the semantic information. And the value is the semantic information. For each query including semantic information, the system first tries to query the cache.

Figure 6 shows the cache mechanism. Naturally, one AI model indicates one semantic one space (one-to-one mapping). When the admin updates the AI model, the new model would have an updated serial number. A cache is valid when the serial number in the cache’s key equals the latest model. For example, suppose that the AI model with serial number 002 is in use, then the fourth cache is out of date. Because the serial number of it is 001.

6.2.2 Semantic Information Index. When there is a large amount of semantic information, it is essential to build an index. Each kind of semantic information has its own meaning. For example, facial features are vectors, text contents of audios are in the string format, etc. Different methods should be applied to index these different data. In this work, PandaDB adopts different index methods for a different type of semantic information. For the numerical data, the semantic index is based on B-Tree[11, 18], inverted index[51, 56] is adopted for semantic information under the format of strings and texts. For high dimensional vectors data, we adopt inverted vectors search[4]. Note, once we build the index, the query plan generator would push down the related semantic information operator into the related index and speed up the data query processing. In addition, PandaDB applies two strategies for building indexes, batch building and dynamic building. The former applies to a semantic space that is not indexed before or the corresponding AI model is updated. The latter is adopted when there is a new semantic information item(ie., a newly added unstructured data item in the database). More details are given in the appendix of the tech report [54].

7 EXPERIMENT AND IMPLEMENTATION

7.1 Implementation

PandaDB extends Neo4j[39] to support the unstructured data processing, in addition, we choose HBase [7] to store unstructured data. We implement the semantic information index engine adopting Milvus[49], an open-source C++-based vector database for vector search. In addition, PandaDB adopts ElasticSearch[17, 31] as the index for structured property data, thus, a node in PandaDB is mapped to a document in ElasticSearch, then the property name to the field name of the document, and the property value to the document content. When the PandaDB deals with a query, the property filter is pushed down to be executed on the ElasticSearch.

In order to support more extensive scale data, PandaDB distributes the data to multiple nodes. Considering the characteristics of different data, the distribution methods of these data among nodes are also different. The data volume of graph structure data is relatively small, and the correlation between data is vital. If the system partitions the graph structure data into different nodes in the cluster, it will produce significant cross-node communication overhead. Therefore, a copy of graph structure data is saved on each node, and the raft protocol is used to maintain the consistency of graph structure data. The property data, including structured and unstructured properties, are stored on the cluster in a partitioned manner because of its relatively large amount of volume. Besides, cross node distribution will not lead to significant cross-node communication. In the implementation, structured properties are stored in ElasticSearch, and unstructured properties are stored in HBase. When a new physical driver connects to a cluster, the queries it sends are divided into reading-query and writing-query. Thus, the reading-query only reads the data, while writing-query also modifies the data. Reading-query is randomly distributed to any available machine, and writing-query is forwarded to the leader for execution. The leader node initiates data synchronization within the cluster. When the leader node executes a writing-query, it records its corresponding query statements and assigns a version number to each writing-query in ascending order. The version number and query statement are recorded in the log. This log is synchronized to other nodes in the cluster. When a node goes online, it first compares whether the local log version is consistent with the log version of the leader in the current cluster. If consistent, the node can join the cluster. If the local log version is lower than the cluster log version, execute query statements in the local log until the version is consistent.

In total, the project includes about 50,000 lines of source code. All the source codes could be accessed at the link\footnote{https://github.com/grapheco/pandadb-v0.2}.

7.2 Cases Studies

7.2.1 Academic graph disambiguation and mining. NSFC (National Natural Science Foundation of China) is responsible for managing national scientific research funded projects. It stores and manages data about scholars, published papers, academic affiliation and scientific research funds details. Millions of users visit the service monthly. There are many implicit relationships in these data, such as authorOf between scholars and papers, workFor between scholars and organizations. Figure7 shows the data overview in NSFC, there are about 1.5TB data in total, with 2 million scholars. Three example queries are shown in Figure7, all of them involves unstructured semantic information, about sixty different types of queries similar to these three are carried on the system. When managing such large amount of data, we run into several graph queries scenarios: (a) Lack of relationships, including the creative relationship between scholars and papers, the subordinate relationship between scholars and organizations, etc. (b) Entity ambiguity. Some scholars have multiple names (i.e. Wang Wei, Wei Wang, WW, Wei W, Wang W may refer to the same scholar), and some different scholars have the same name. According to the graph structure and property information, the traditional solution can only confirm the entity direction.

PandaDB is widely used in the NSFC for the user name disambiguation. We use OCR technology to extract the author and scientific research organization information from the PDF file of the paper, then construct the corresponding association relationship between authors and their corresponding universities. This affiliation is used to build the connection among two nodes of graph.\footnote{https://github.com/grapheco/pandadb-2019}
Figure 7: Academic graph disambiguation and mining in NSFC.

Then, the similarity of facial photos between nodes is analyzed by the predefined UDF function. Nodes with similar facial features are considered to point to the same scholar, then two authors with same name can be identified based on the graph queries. The accuracy rate exceeds methods based on graph neural network[40]. In addition, PandaDB also provides classical graph query and graph mining, such as relationship query and community discovery. Nowadays, PandaDB is providing the graph query processing for more than 100k researchers in China monthly and enable users to find the related scientific documents more easily.

7.2.2 **DoubanMovie system**. When watching TV programs, viewers often look at an actor and cannot remember his name or what programs the actor has played. PandaDB is deployed to help user to find the super star in **DoubanMovie**, the biggest movie comments and review website in China. **DoubanMovie** contains more than 10 million movies and 1 million super star. We built a graph containing actors, movies, and participation relationships. When the user submits a photo, PandaDB can find the superstar share the similar photo as the facial information of the input photo, then find the film in which the actor has played from the graph. This system is deployed and used in the production environment, and one demo video is in the link.

### 7.3 Experimental Setup

We combine a graph benchmark dataset and a face recognition dataset to obtain a property graph including unstructured data. For property graph data, we adopt Linked Data Benchmark Council Social Network Benchmark[LDBC-SN][13]. It is a scalable simulated social network dataset organized as a graph. For unstructured data, we use Labeled Faces in the Wild(LFW)[24], it is a public benchmark for face verification, including 13233 photos of the face in the wild environment. We attach the photos in LFW to person nodes in LDBC-SN, each node a photo. For recording the mapping between node and photo, the photo's id is set as a property of the node. We use different scale of dataset to evaluate the performance of PandaDB, the datasets are detailed in Table 4, where SF is short for scale factor, an argument detailed in Table 4, where SF is short for scale factor, an argument considered about the limited pages, we detail four of the eight queries.

| Name  | #Node  | #Relationship | #BLOB  | Total Space |
|-------|--------|---------------|--------|-------------|
| SF1   | 3115527| 17236389      | 9916   | 2.0GB       |
| SF3   | 8879918| 50728269      | 24292  | 5.6GB       |
| SF10  | 28125740| 166346601     | 65675  | 18.0GB      |
| SF30  | 83298531| 507720806     | 165643 | 57.2GB      |

#### 7.3.1 **Testbed**. The experiments are conducted with a cluster including five physical machines. Each node has 52 logical cores, 384GB RAM, 2TB SSD, and 215TB HDD. These machines are all equipped with two Intel Xeon Gold 6230R CPUs (2.10GHz). Machines are connected via a 10Gbps Ethernet network.

7.3.2 **Dataset**. We combine two public datasets to obtain a property graph with unstructured data. For property graph data, we adopt Linked Data Benchmark Council Social Network Benchmark[LDBC-SN][13]. It is a scalable simulated social network dataset organized as a graph. For unstructured data, we use Labeled Faces in the Wild(LFW)[24], it is a public benchmark for face verification, including 13233 photos of the face in the wild environment. We attach the photos in LFW to person nodes in LDBC-SN, each node a photo. For recording the mapping between node and photo, the photo's id is set as a property of the node. We use different scale of dataset to evaluate the performance of PandaDB, the datasets are detailed in Table 4, where SF is short for scale factor, an argument to describe the scale of dataset.

#### 7.3.3 **Query**. The experiment designs eight queries to simulate the queries in real-world applications. Actually, we carefully evaluate the performance of PandaDB over these eight queries. While in consideration about the limited pages, we detail four of the eight queries, the others share the same conclusion of these four queries. The query statements and their meanings are listed as below. Note that the symbol $\sim$ is defined to judge whether two faces similar, by comparing the similarity between the facial features.

```sql
-- Q1: Query a node by name and photo.
MATCH (n:Person) WHERE n.photo = Blob.fromURL('http://example.com/\$name') AND n.firstName = '$name1' RETURN n;

-- Q2: Query the shortest path between two nodes.
MATCH (n:Person),(m:Person) WHERE m.photo = Blob.fromURL('http://example.com/\$name2') AND n.firstName = '$name1' RETURN shortestPath((n)-[*1..3]-(m));

-- Q3: Whether two nodes refer to the same person.
MATCH (n:Person),(m:Person) WHERE n.firstName = '$name1' AND m.firstName = '$name2' RETURN n.photo = m.photo;

-- Q4: Whether the two friends looks similar.
MATCH p = (n:Person)-[:friendOf]->(m:Person) WHERE n.photo = m.photo RETURN p;
```

7.3.4 **Native solution implementation**. We implemented native solution as the baseline in the experiment. In the native solution, we use neo4j to maintain the graph data, take local file system to store the photos, use AIPM to extract the facial features in photos, and calculate the similarity by outer scripts. The query process is detailed as below:
(1) Q1: Find the photos whose facial features are similar to those of the specific BLOB. Next, retrieve the corresponding nodes of the photos, then filter the nodes by the firstName.
(2) Q2: Find the nodes whose photo similar to the specific BLOB and the nodes whose firstName meet the argument. Then retrieve the shortest path between the nodes in neo4j.
(3) Q3: First retrieve the nodes whose firstName meet the arguments in query statement, then calculate the similarity of the facial features.
(4) Q4: First retrieve the nodes corresponds to the path, then calculate the similarity of the facial features.

7.4 Throughput and Response Time
In order to test the throughput of PandaDB and its ability to handle concurrent requests, we use Apache JMeter\(^\text{b}\) to simulate concurrent requests in a real applications. The response time of a single query keeps at about 20ms, and the throughput increases with the increase of the number of requests sent per second until it reaches 5300 times per second.

7.5 PandaDB vs Native Solution
In this section, we execute the four queries detailed in Tabel 4 to compare the overview performance of PandaDB and native solution. The results are shown in Figure 8 and Figure 9. Each sub-figure represents the result of a query. The x-axis means the scale of dataset, the details about the scale are introduced in Section 7.3. The y-axis means the execution time; we take logarithm of the execution time in the figures because of the significant performance gap.

Each line in the figures represent a solution. The PandaDB-NoOP stands for a PandaDB without optimization for unstructured data queries. And the PandaDB-OP is optimized for unstructured data queries by the method introduced in Section 5.2. The differences between them lay on the query plan, where unstructured data filter could be executed more front or back. That would lead to a different workload for the unstructured data filter.

We set the upper limit of query time to 24 hours. When the execution time of a query exceeds 24 hours, we regard the query times out and it will not show the result in the figure. For example, the native solution times out on Q4 over all the datasets, when the semantic information is not cached(i.e. Figure 8 (d)).

Because of the different features of the four queries, the performance improvements of PandaDB differ over them. The performance of PandaDB and native solution differs litter on Q2 when the semantic information is not cached(i.e. Figure 8(b)). Because in this case, the execution process of PandaDB is the quite similar to that of the native solution. In all the other situations, PandaDB performs better than the native solution overall four queries because there is a less overhead cost in PandaDB.

When the semantic information is not cached, in Q1, PandaDB has about 3 orders of magnitude advantages over the native solution; in Q3, PandaDB is faster 10x than the native solution on average. Compared with Q1, Q3 and Q4, PandaDB has less performance advantage in Q2. The query optimization allows PandaDB to execute the query with fewer extraction operations. Actually, according to the optimization detailed in Section 5, PandaDB filters the data according to the structured data and then filters the result by semantic information. But the native has to filter all the semantic information. While in Q2, both PandaDB and native solution need to extract semantic information of all the unstructured data in the database. So the performance differs little in Q2.

After pre-extraction and caching of the semantic information, we re-evaluate the overview performance. The results are shown in Figure9. Over the four queries, PandaDB performs 100x to 1000x faster than the pipeline system. As introduced before, extracting semantic information takes most of the time. While, in this case, the semantic information is pre-extracted and cached, it takes nearly no time to be obtained. So, the overhead cost influences the query time more. In the native solution implementation, data flow from a component to another costs much, especially when the data is large(unstructured data is also larger than structured data). While PandaDB executes the whole inner database, so PandaDB performs much better than the native solution.

7.6 Unstructured Data Storage Performance Evaluation
In neo4j, unstructured data can be stored as ByteArrayInputStream. Some applications store unstructured data in key-value database, in which the ID of unstructured data is used as key and the binary content is used as value. PandaDB stores unstructured data as BLOB. We use different methods to store unstructured data, and compare the reading and writing efficiency of these methods. We conducted a read-write test on unstructured data content from 1KB to 10MB. Considering the streaming reading requirements in some scenarios, in the reading test, we tested the time required to read the first byte, middle byte and end byte of unstructured data. The results are shown in Figure 10. BLOB performs better on all the reading tasks. Because both the neo4j and RocksDB solution needs to load the whole unstructured item from the disk to get even the first byte of the unstructured data.

7.7 Query Optimization and Index Performance Evaluation
This experiment evaluates the efficiency of optimization work on an optimized PandaDB and a PandaDB that treats the semantic information filter as an ordinary structured property filter(namely, Not optimized). We also evaluate the effectiveness of PandaIndex on the SIFT-1M\(^\text{[27]}\), and SIFT-100M\(^{1/10}\) of the SIFT1B\(^{[29]}\)) build the index for the dataset, then execute kNN search, evaluate the recall and performance. Experimental results show that query optimization improves the average query performance by an order of magnitude. Figure 11 and Figure 12 give the results. With the PandaIndex, PandaDB could query unstructured data in milliseconds.

More details are presented in the appendix of tech report [54].

8 CONCLUSION
In this work, we introduced the cost of unstructured data operators(a.k.a extractor, computation and filter) into the cost model and optimize the query plan with a greedy-based optimization. The optimized data storage and indexing of semantic information significantly improve the efficiency of queries. The proposed AIPM
enables the database with unstructured data analysis service. Users are able to execute complex queries involving unstructured data on property graph, by driving a single system.

Figure 8: Overview performance comparison (semantic information not cached)

Figure 9: Overview performance comparison (semantic information cached)

Figure 10: Unstructured data storage performance evaluation

Figure 11: Index recall evaluation on kNN search.

9 APPENDIX

9.1 Build Index for Semantic Data

Algorithm 2 shows how PandaDB builds index for semantic space composed by vectors. For high dimensional vectors, we divide the space into m buckets. Each bucket has a core vector, and vectors are assigned to this bucket based on the closet distance. Suppose a
kNN search task where \( k = 1 \), the system first calculates distances of the vector to each core-vector, then selects the corresponding bucket of the nearest core-vector. Next, execute a linear search in this bucket, find the nearest vector. For datasets with a larger scale, we also offer the implementation of HNSW\[35\] and IVF_SQ8\[37\]. These two index algorithms perform better on larger datasets of vectors, and HNSW even supports dynamic insert. The inverted vector search is an ANNS(Approximate Nearest Neighbour Search).

When the semantic information is pre-extracted and cached, the optimization performs better in Q2. In this case, semantic information filter is slower than structured property filter, so putting semantic information filtering behind can reduce the overhead. In the case without cache, there is also this optimization logic. However, when there is no cache, the extraction of semantic information takes much time, so the effect of this optimization is not apparent.

9.3 Index Performance Evaluation

We brought kNN search on the datasets(respectively, with \( k = 1, 10, 100, 500 \)). For each \( k \) value, the experiment is repeated 500 times, recording the max, min, and average of the query accuracy. The result is shown in Figure11. The average accuracy is stable above 0.95. When the \( K \) value is small, there are very few cases of low accuracy.

In order to evaluate the query speed of the index, we carried out experiments from the perspectives of single vector retrieval and batch vector retrieval. For single vector retrieval, kNN retrieval is performed on one vector at a time, and the query time is recorded. For batch vector retrieval, ten vectors are searched by kNN each time, and the query time is recorded. Among them, the value of \( K \) is 1, 10, 100, and 500, respectively. For each \( K \) value, repeat 500 times and record the average value. The results are shown in Figure 12, where the \( #v \) means the number of vectors included in a query. Figure 12(a) records the average time spent per query in 500 repeated experiments under different conditions. Figure 12(b) records the average time spent per vector in a query, for queries with \( #v = 1 \). The average time of each vector is the time of the query. For \( #v = 10 \) queries, the average time per vector is 1 / 10 of the query time. On the same dataset, the total time consumption of single query and batch query is very close and does not change significantly with the change of \( K \) value. The average time consumption does not increase significantly with the increase of \( K \) value, which also enlightens us that we can reduce the average time consumption of each vector query by submitting batch query tasks.

9.2 Optimization Comparison

The results are shown in Figure8 and Figure9. The features differs from one query to another, so the optimization efficiency differs. There are two filters in Q1, one for structured data(filter by name), the other for semantic information(filter by face feature). The input of the first filter is all the property data in the database, while the input for the second filter is the output of the first one. Obviously, executing the filter for the name would make the semantic information filter extract fewer data than executing the name filter later. While in Q2 and Q3, the number of semantic information to be extracted could not be narrowed down, so the optimization does not perform well.

Algorithm 2: Semantic Information Indexing Algorithm

| Step | Description |
|------|-------------|
| 1    | Function PickBucket(vec, B): |
| 2    | D ← ∅ |
| 3    | foreach bucket ∈ B do |
| 4    | d ← distance(vec, bucket.core) |
| 5    | D.insert(d, bucket) |
| 6    | bucket ← minByDis(D) |
| 7    | return bucket |
| 8    | Function BatchIndexing(S): |
| 9    | if S is ∅ then |
| 10   | S ← GetSemSpace(D, Schema, subPty) |
| 11   | m ← count(S) |
| 12   | B ← ∅ |
| 13   | while size(B) < m/100000 do |
| 14   | bucket.core ← randomSelect(S) |
| 15   | S.remove(bucket.core) |
| 16   | foreach vec ∈ Space do |
| 17   | bucket ← PickBucket(vec, B) |
| 18   | bucket.insert(vec) |
| 19   | return B |
| 20   | Function DynamicIndexing(d): |
| 21   | i ← ExtractSemInfo(d, subPty, Schema) |
| 22   | Space.insert(i) |
| 23   | bucket ← PickBucket(i, B) |
| 24   | bucket.insert(i) |
| 25   | return B |

REFERENCES

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindalaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In Proceedings of the 2018 International Conference on Management of Data. 1421–1432.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domenico Vegliò. 2017. Foundations of modern query languages for graph databases. ACM Computing Surveys (CSUR) 50, 5 (2017), 1–40.

[3] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. ACM Computing Surveys (CSUR) 40, 1 (2008), 1–39.

[4] Chris Buckley and Alan F. Liu. 1985. Optimization of inverted vector search. In Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval. 97–110.

[5] Baidu 2021. HugeGraph. Baidu. Retrieved June 21, 2021 from https://github.com/hugegraph/hugegraph.

[6] Vito Giovanni Castellana, Alessandro Morari, Jesse Weaver, Antonino Tumeo, David Haglin, Oreste Villa, and John Feo. 2015. In-memory graph databases for web-scale data. Computer 48, 3 (2015), 24–35.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26, 2 (2008), 1–26.

[8] Gal Chechik, Eugene Ie, Martin Rehn, Samy Bengio, and Dick Lyon. 2008. Large-scale content-based audio retrieval from text queries. In Proceedings of the 1st ACM international conference on Multimedia information retrieval. 105–112.

[9] Li-Fen Chen, Hong-Yuan Mark Liao, Ming-Tat Ko, Ja-Chen Lin, and Gwo-Jong Yu. 2000. A new LDA-based face recognition system which can solve the small
