Lynx: A Graph Query Framework for Multiple Heterogeneous Data Sources

Zhihong Shen
Computer Network Information Center, Chinese Academy of Sciences
Beijing, China
bluejoe@cnic.cn

Chuan Hu
Computer Network Information Center, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
huchuan@cnic.cn

Zihao Zhao
Computer Network Information Center, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
zhaozihao@cnic.cn

ABSTRACT

Graph model are increasingly popular among modern applications for its ability to model complex relationships between entities. Users tend to query the data as a graph with graph operations (e.g., graph navigation and exploration). However, a large fraction of the data resides in relational databases or other storage systems. Challenges arise in uniformly querying multiple heterogeneous data sources as a graph. Traditional solutions are limited by time-consuming data integration, expensive development effort, and incomplete query requirements. Thus, we developed Lynx, a general graph query framework, to simplify querying graph data by converting complex statements into basic graph operations. Instead of connecting directly to the data sources, Lynx retrieves data through user-implemented interfaces for those graph operations. We demonstrate Lynx's capabilities through real-world scenarios, showcasing Lynx's ability to process graph queries on multiple heterogeneous data sources and also to be used as a generic graph query engine development framework.

PVLDB Reference Format:
Zhihong Shen, Chuan Hu, and Zihao Zhao. Lynx: A Graph Query Framework for Multiple Heterogeneous Data Sources. PVLDB, 16(12): 3926 - 3929, 2023.
doi:10.14778/3611540.3611587

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lynxworld/lynx.

1 INTRODUCTION

The graph can model complex relationships between a variety of entities. Interconnected data are usually modeled as graphs in many applications like social networks, smart cities, and knowledge graphs. In these applications, a significant fraction of the data resides in relational databases or other storage systems (e.g., key-value and column-oriented databases). Modern applications (e.g., large-scale knowledge graph [6]) need to query data from these heterogeneous data sources. Different data sources manage data using different models, so the data operations they support differ. For example, KV databases can only find values through keys, while relational databases are difficult to find paths. The heterogeneous data models make it hard to query the data as a graph uniformly.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611587

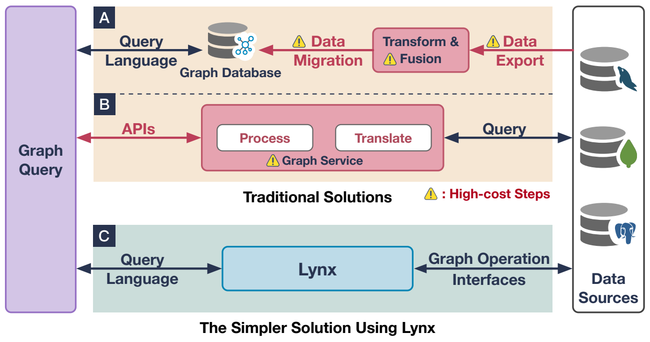


Figure 1: The diagram of various solutions proposed for multi-data-source graph query tasks, in which the red font refers to the high-cost steps.

There are two traditional solutions to address the problem, as shown in Figure 1. Solution A involves exporting the data from different sources, transforming it, fusing it, and importing it into the graph database. This approach is time-consuming due to data integration and migration. Furthermore, it cannot be easy to maintain data consistency between the graph database and multiple data sources in real-time. Solution B is to develop graph services that offer APIs for frequently used graph queries. However, developing such graph services can be expensive, requiring significant development effort. The flexibility, ease of use, and portability of APIs are not as good as those of query languages.

To mitigate the challenges mentioned earlier, we have developed Lynx, a general graph query framework. The purpose of Lynx is to decompose complex query statements into fundamental graph operations, thereby simplifying the process of querying graph data. Lynx does not manipulate the data source directly but provides the interfaces of these graph operations, such as data accessing, path-finding, index manager, etc.

In Lynx, this conversion is a pipelined processing flow. Lynx parses the graph query statements into AST (i.e., Abstract Syntax Tree), then constructs logical and physical plans. The query plans are optimized by the optimizer in Lynx. The execution engine

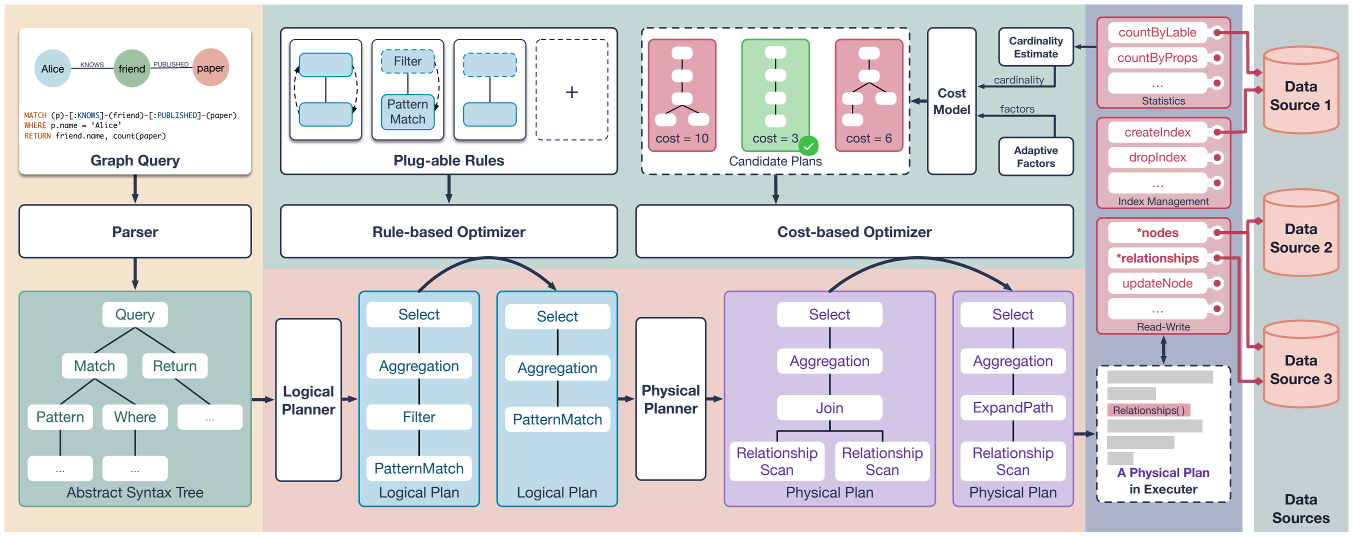


Figure 2: The Architecture and workflow of Lynx. The whole process is a pipeline and divided into the following parts: ①Query parsing (yellow background); ②Plan generation (red background); ③Plan optimization (cyan background); ④Plan execution (purple background); ⑤Graph Operation Interface (red box) and connection to data sources (red line).

regards the physical plan as a permutation of the fundamental graph operations and necessary algebra operations.

In this demo, we showcased Lynx through several real-world scenarios. We designed a multi-data-source scenario containing Mysql and Redis to showcase that Lynx can process graph queries on heterogeneous data sources by simply implementing interfaces. We also introduced how to develop a graph database by combining Lynx with a KV database and an index engine. This scenario demonstrates that Lynx can be used as a generic graph query engine development framework to simplify graph database development.

2 SYSTEM OVERVIEW

Lynx, a graph querying framework, follows a modular architecture that comprises several critical components, such as a parser, planner, and optimizer, which work collaboratively to ensure the efficient and effective execution of queries. Like other current query engines, Lynx leverages a pipeline-style processing methodology that facilitates the systematic handling of queries. Specifically, the graph query language is first parsed and converted into an abstract syntax tree, which the planner then transforms into a query plan. Subsequently, the optimizer reorganizes the query plan to improve execution efficiency before final execution in the execution engine to produce the expected results.

2.1 Query Plan Generation

A parser and a planner are needed to convert a query language into an executable query plan. The parser converts the query language into a structured abstract syntax tree. This tree represents this query's structure and enables further processing by the planner.

To achieve separation from the underlying storage system, Lynx employs two distinct types of query plans: logical plans and physical plans. The logical plan describes the query logic expressed in the query statement without specifying the actual execution details. In contrast, the physical plan is executable and directly involves physical data operations intimately tied to the underlying storage. Specifically, for instance, a sub-graph matching query is represented as a *PatternMatch* operator in the logical plan, which formalizes the query operation's specific details, such as relationship types and hop counts. In contrast, in the physical plan, this operation is transformed into a collection of individual operators such as *NodeScan*, *RelationshipScan*, or *IndexSearch*, depending on the underlying storage structure's design.

This separation of logical and physical plans provides two key advantages: firstly, users are empowered to expand physical plan operators based on the actual physical storage conditions and to modify the rules governing the transformation from logical to physical plans. Secondly, this design enables the development of parsers and planners for various graph languages, allowing for using a single set of logical plans to describe different graph query languages.

2.2 Plan Optimization

Lynx optimizes query plans using a rule-based optimizer and a cost-based optimizer, following the algorithm in Ref [7].

The rule-based optimizer rewrites queries to generate new query plans that produce equivalent results with less execution time. It rewrites queries based on algebra logic and an extensible rule set, including *filter/aggregation push-down* and *constant folding*. Lynx's embedded rule set includes optimization rules for common data models (e.g., RDBMS and KV), and developers can extend the rule set according to their storage backends' characteristics.

The cost-based optimizer estimates the cost of candidate query plans using an adaptive cardinality cost model and selects the one with the lowest expected cost. Traditional databases rely on a storage backend where the speed of processing certain operations for the same backend is constant.

The cost model in traditional databases uses fixed speed factors for each operation. However, these fixed speed factors do not work on different storage backends due to differences in their characteristics. For example, full-text retrieval is fast on ElasticSearch but time-consuming on MySQL. To address this, Lynx introduces an adaptive cardinality cost model that updates the speed factors on different storage backends by executing a set of sample queries.

2.3 Execution and Graph Operation Interfaces

Each operator in physical query plan is executable, and the entire physical plan tree is executed from the leaf node(s) to the root node. Operators can be categorized into two types. The first type includes operators such as *Join* and *Project*, which generally do not require access to external data sources and can be fully executed within the executor. The second type includes operators like *NodeScan* and *ExpandPath*. These operators involve data retrieval or updates and necessitate graph operations during execution. For example, as illustrated in the lower right corner of Figure 2, the *RelationshipScan* operator employs the `relationships` operation during its execution process. The graph operation interface in Lynx is crucial for implementing queries on heterogeneous data sources. In Lynx, we have designed about 40 fundamental graph operation interfaces, as shown in the upper right corner of Figure 2, which can be roughly divided into the following categories:

- **Read-Write**: The primary purpose of these interfaces is to retrieve and modify data from multiple sources, such as `nodes`, `updateNode`, `createElement`, etc. They also include path-finding interfaces like `expand` and `shortestPath`.
- **Statistics**: These interfaces provide statistics for the optimizers to estimate the cost of query plans.
- **Index Management**: This category encompasses graph operation interfaces for creating and dropping indexes.

Figure 2 illustrates how the graph data interface connects to data sources. As shown, `countByLabel` and `createIndex` are connected to *Data Source 1*, which could be a repository for managing metadata and indexes. The `nodes` interface links to two data sources, indicating that node data is distributed across different data sources in this example. The specific data source to access can be distinguished within the concrete implementation of the `nodes` interface (e.g., by label or property). In addressing the issue of data dispersion, Lynx adopts this flexible approach.

For the majority of operations, Lynx provides default implementations, except for the two data access interfaces: `nodes` and `relationships`. This means that for lightweight tasks that do not involve data writing, such as querying CSV files, only implementing Lynx's two data access interfaces is sufficient to complete the task. For interfaces with default implementations, users can also override them as needed, demonstrating Lynx's flexibility in data access.

Implementation. Lynx is developed in Scala, offering compatibility with Java and allowing seamless integration into a wide range of projects. To utilize Lynx, users can import it as a JAR package, ensuring effortless implementation and usage. Currently, Lynx supports OpenCypher [5], a widely-adopted and expressive graph query language.

3 DEMONSTRATION

In our demo, we show how developers can use Lynx to solve graph query problems. The demo includes a base scenario: using public datasets and common databases, we show how developers can support graph queries on non-graph database data sources by implementing interfaces designed to showcase Lynx's multi-source graph query capabilities and ease of use. In addition, an extension scenario is set up to showcase Lynx's flexibility and capability in graph database development.

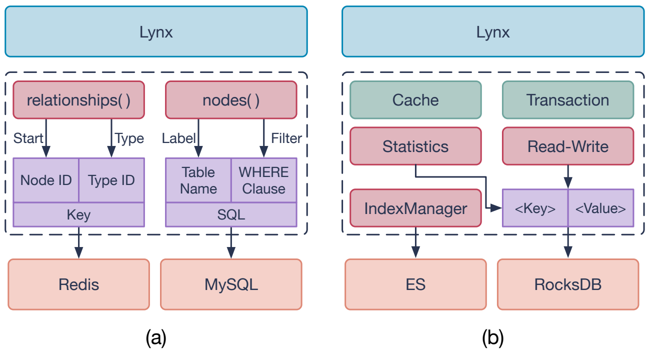


Figure 3: The diagram of the use of Lynx in two scenarios. The dashed boxes in the diagram are related to the graph operation interface implementation. (a) In the implementation of the two data access interfaces, nodes and relationships, the filter information is converted into query language or key to query data from MySQL and Redis. (b) A graph database built with a storage, an indexing engine, and Lynx. Lynx is used as a query engine development framework in database development.

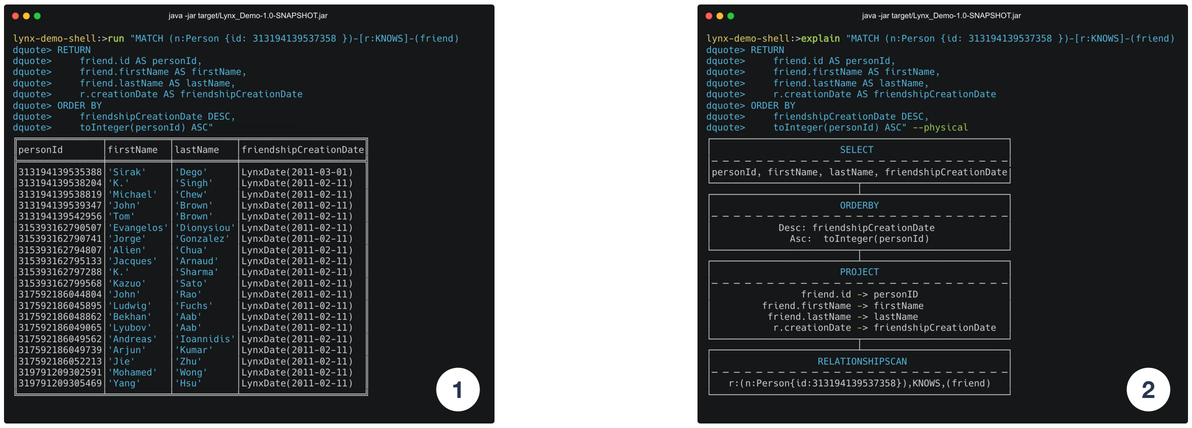
3.1 Scenario 1: Graph Query across Data Sources

Imagine that we are faced with a scenario where we possess a large amount of data in multiple databases that use different data models, including relational and key-value models. Some data within these databases are related; now, we need to do graph queries on these data but don't want to migrate them.

We use the dataset from LDBC-SNB [2], one of the most popular property graph benchmarks. The nodes are stored in different MySQL data tables according to their labels, and the node id is set as the primary key, which is consistent with the scenario described in the scenario; the relationships among nodes exist in different Redis according to their types; the graph queries used for testing are also from LDBC-SNB, in the form of Cypher.

In Figure 3 (a), two fundamental interfaces have been implemented to retrieve data after importing Lynx, as depicted by the dotted line.

- The node interface enables querying data from MySQL using a SQL statement consisting of two components. The table name locates the relational table where the nodes to be queried are stored, which is determined by the node's Label. If no label is specified, all relational tables are queried. Moreover, the `nodes` interface may include property key-value


java -jar target/Lynx_Demo-1.0-SNAPSHOT.jar
1
java -jar target/Lynx_Demo-1.0-SNAPSHOT.jar
2

```
lynx-demo-shell:>run "MATCH (n:Person {id: 313194139537358 })-[r:KNOWS]-(friend)
dquote> RETURN
dquote>     friend.id AS personId,
dquote>     friend.firstName AS firstName,
dquote>     friend.lastName AS lastName,
dquote>     r.creationDate AS friendshipCreationDate
dquote> ORDER BY
dquote>     friendshipCreationDate DESC,
dquote>     toInteger(personId) ASC"
╔═══════════════╤═══════════╤═══════════╤══════════════════════╗
║personId       │firstName  │lastName   │friendshipCreationDate║
╠═══════════════╪═══════════╪═══════════╪══════════════════════╣
║313194139535388│'Sirak'    │'Dego'     │LynxDate(2011-03-01)  ║
║313194139538204│'K.'       │'Singh'    │LynxDate(2011-02-11)  ║
║313194139538819│'Michael'  │'Chew'     │LynxDate(2011-02-11)  ║
║313194139539347│'John'     │'Brown'    │LynxDate(2011-02-11)  ║
║313194139542956│'Tom'      │'Brown'    │LynxDate(2011-02-11)  ║
║315393162790507│'Evangelos'│'Dionysiou'│LynxDate(2011-02-11)  ║
║315393162790741│'Jorge'    │'Gonzalez' │LynxDate(2011-02-11)  ║
║315393162794807│'Alien'    │'Chua'     │LynxDate(2011-02-11)  ║
║315393162795133│'Jacques'  │'Arnaud'   │LynxDate(2011-02-11)  ║
║315393162797288│'K.'       │'Sharma'   │LynxDate(2011-02-11)  ║
║315393162799568│'Kazuo'    │'Sato'     │LynxDate(2011-02-11)  ║
║317592186044804│'John'     │'Rao'      │LynxDate(2011-02-11)  ║
║317592186045895│'Ludwig'   │'Fuchs'    │LynxDate(2011-02-11)  ║
║317592186048862│'Bekhan'   │'Aab'      │LynxDate(2011-02-11)  ║
║317592186049065│'Lyubov'   │'Aab'      │LynxDate(2011-02-11)  ║
║317592186049562│'Andreas'  │'Ioannidis'│LynxDate(2011-02-11)  ║
║317592186049739│'Arjun'    │'Kumar'    │LynxDate(2011-02-11)  ║
║317592186052213│'Jie'      │'Zhu'      │LynxDate(2011-02-11)  ║
║319791209302591│'Mohamed'  │'Wong'     │LynxDate(2011-02-11)  ║
║319791209305469│'Yang'     │'Hsu'      │LynxDate(2011-02-11)  ║
╚═══════════════╧═══════════╧═══════════╧══════════════════════╝
```

```
lynx-demo-shell:>explain "MATCH (n:Person {id: 313194139537358 })-[r:KNOWS]-(friend)
dquote> RETURN
dquote>     friend.id AS personId,
dquote>     friend.firstName AS firstName,
dquote>     friend.lastName AS lastName,
dquote>     r.creationDate AS friendshipCreationDate
dquote> ORDER BY
dquote>     friendshipCreationDate DESC,
dquote>     toInteger(personId) ASC" --physical
┌───────────────────────────────────────────────────┐
│                      SELECT                       │
│ - - - - - - - - - - - - - - - - - - - - - - - - - │
│personId, firstName, lastName, friendshipCreationDate│
└─────────────────────────┬─────────────────────────┘
┌─────────────────────────┴─────────────────────────┐
│                      ORDERBY                      │
│ - - - - - - - - - - - - - - - - - - - - - - - - - │
│           Desc: friendshipCreationDate            │
│             Asc:  toInteger(personId)             │
└─────────────────────────┬─────────────────────────┘
┌─────────────────────────┴─────────────────────────┐
│                      PROJECT                      │
│ - - - - - - - - - - - - - - - - - - - - - - - - - │
│               friend.id -> personID               │
│        friend.firstName -> firstName              │
│         friend.lastName -> lastName               │
│          r.creationDate -> friendshipCreationDate │
└─────────────────────────┬─────────────────────────┘
┌─────────────────────────┴─────────────────────────┐
│                 RELATIONSHIPSCAN                  │
│ - - - - - - - - - - - - - - - - - - - - - - - - - │
│   r:(n:Person{id:313194139537358}),KNOWS,(friend) │
└───────────────────────────────────────────────────┘
```

Figure 4: A shell application provided by Lynx. The figure shows the results of executing a test query statement on the LDBC-SNB dataset. ① shows the query results. ② shows the query plan.

pairs that act as filter conditions. These filter conditions are transformed into WHERE clauses in SQL and then executed in MySQL.

- The `relationships` interface is accountable for retrieving all relationships for a specific start node and relationship type. The key to retrieving Redis involves concatenating the starting node ID and the relationship type ID.

Lynx offers a shell interaction application with a run command and an explain command to execute Cypher queries and view query plans. Figure 4(a) demonstrates the result of executing this statement, whereas Figure 4(b) exhibits the corresponding physical plan.

3.2 Scenario 2: Graph Database Development

PandaDB[1] is a scalable, high-performance graph database based on Key-Value storage. It maintains property graph data as key-value, and the flexible KV storage makes it easy to scale out for large datasets. PandaDB supports full-text index of properties. It has been evaluated on a dataset with billions of nodes and tens of billions of relationships. PandaDB performs better than Neo4j on property filtering and simple graph queries. PandaDB adopts Lynx to develop its query engine, as shown in Figure 3(b). Developers only need to implement interfaces shown in Figure 2, without paying attention to the implementation of query parsing, plan generation, and optimization. This allows developers to focus more on the design and polishing of the storage engine. The **customizable framework** also allows developers to implement query engines based on actual scenario requirements. As far as we know, several teams are developing their experimental graph database with Lynx.

4 RELATED WORK

There are several frameworks (e.g., Cytosm [8] and GraphGen [9]) support graph query over non-graph databases by mapping query languages. However, they only support single data model (usually RDBMS) and are unsuitable for heterogeneous data sources. JanusGraph [1], a graph database, supports multiple data sources, but its graph data storage is restricted to BigTable [4] databases like Apache Cassandra and Apache HBase.

Apache Calcite [3] is a query framework that enables processing multiple heterogeneous data sources but does not support graph queries.

[1]https://github.com/grapheco/pandadb-v0.3

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China(Grant No.2021YFF0704200) and Informatization Plan of Chinese Academy of Sciences(Grant No.CAS-WX2022GC-02).

REFERENCES

[1] 2023. *JanusGraph*. https://janusgraph.org/
[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, et al. 2020. The LDBC social network benchmark. *arXiv preprint arXiv:2001.02299* (2020).
[3] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.
[4] 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.
[5] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying.. In *EDBT*. 520–523.
[6] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4, Article 71 (jul 2021), 37 pages. https://doi.org/10.1145/3447772
[7] Donald Kossmann and Konrad Stocker. 2000. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Trans. Database Syst.* 25, 1 (mar 2000), 43–82. https://doi.org/10.1145/352958.352982
[8] Benjamin A Steer, Alhamza Alnaimi, Marco ABFG Lotz, Felix Cuadrado, Luis M Vaquero, and Joan Varvenne. 2017. Cytosm: Declarative property graph queries without data migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. 1–6.
[9] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. 2017. Graphgen: Adaptive graph processing using relational databases. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. 1–7.