Property Graphs in Arachne

Oliver Alvarado Rodriguez, Fernando Vera Buschmann, Zhihui Du, David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, NJ, USA
{oaa9,fv54,zd4,bader}@njit.edu

***Abstract*—Analyzing large-scale graphs poses challenges due to their increasing size and the demand for interactive and user-friendly analytics tools. These graphs arise from various domains, including cybersecurity, social sciences, health sciences, and network sciences, where networks can represent interactions between humans, neurons in the brain, or malicious flows in a network. Exploring these large graphs is crucial for revealing hidden structures and computing metrics that are not easily computable without parallel hardware. Currently, Python users can leverage Arkouda to efficiently execute Pandas and NumPy-related tasks on thousands of cores. To address large-scale graph analysis, Arachne, an extension to Arkouda, enables easy transformation of Arkouda dataframes into graphs. This paper proposes and evaluates three distributable data structures for property graphs, implemented in Chapel, that will be integrated into Arachne. Enriching Arachne with support for property graphs will empower data scientists to extend their analysis to new problem domains. Property graphs present additional complexities, requiring efficient storage for extra information on vertices and edges, such as labels, relationships, and properties. Arachne is open-source and can be found on GitHub (https://github.com/Bears-R-Us/arkouda-njit).**

***Index Terms*—graph analytics, parallel algorithms, property graphs, distributed-memory**

I. Introduction

Property graphs are widely used in graph database systems to combine graph structures with attributes such as vertex labels, edge relationships, and properties. Data scientists often analyze networks that naturally store these attributes on vertices and edges. These attributes can enhance algorithms for tasks like breadth-first search on specific vertices or filtering subgraphs based on attribute matching, thereby enriching the data scientists' ability to analyze and understand the graph. It is essential to provide solutions for storing property graphs to enable data scientists to leverage the computational power of their systems effectively. These solutions should be integrated into proven libraries and frameworks designed for large-scale analysis, such as Arkouda [15].

Arkouda is an open-source framework initially developed as a scalable replacement for NumPy in Python. Powered by Chapel [5]–[7] at the backend and offering a Python interface, Arkouda has demonstrated its ability to handle datasets comprising over 500 million rows, making it an excellent choice for parallel analysis on large-scale datasets. With a user-friendly interface inspired by NumPy, Arkouda provides predefined operations for users to manipulate their datasets from Python scripts or Jupyter Notebooks. These operations primarily work with **p**arallel and **d**istributed array objects called **pd**arrays. Arkouda facilitates data preparation, exploration, and efficient parallel kernel invocation within a single session. Given that a significant amount of datasets can be structured as graphs, Arachne, built as an extension to Arkouda, facilitates graph analysis [17].

Arachne aims to be a highly productive and efficient graph framework for data scientists looking to extract information efficiently from large graph datasets. It introduces a distributable graph data structure called the Double-Index data structure (DI) [10]. Arachne includes implementations of graph kernels like breadth-first search and triangle counting, which can be executed on both shared-memory and distributed-memory systems. This work focuses on enhancing Arachne's analysis capabilities by introducing additional structure to enhance DI for property graphs.

The main contributions in this paper are as follows:

1) **DIP**, a data structure derived from the **DI** data structure, specifically designed to store **p**roperty graphs.
2) Various versions of DIP implemented in Chapel, exploring space and time-efficient variations: DIP-LIST, DIP-LISTD, and DIP-ARR, utilizing Chapel lists, doubly-linked lists, and two-dimensional array-based attribute storage.
3) Detailed discussions on how Chapel is employed to distribute the data structure and execute querying operations efficiently and in parallel.

II. The Property Graph Data Model

A property graph is a directed and labeled multigraph composed of a set of vertices V and edges E. Each vertex $v \in V$ and edge $(u, v) \in E$ can store property key-value pairs, usually in the form of a tuple $(key, value)$. Vertices can each store labels (or none at all) and edges may store relationships, where each edge between two vertices with a distinct relationship is considered its own entity [1]. If an edge has multiple relationships this means there is a multiedge, i.e., multiple copies of one edge.

Property graphs can be either static or dynamic. In static property graphs, edges and/or vertices cannot be added into the graph over time, whereas dynamic graphs allow for the addition of edges and vertices overtime. For this paper, we only target static property graphs built from datasets that can be viewed as dataframes where vertex labels, edge relationships, and properties can all be inferred from the columns of a tabular

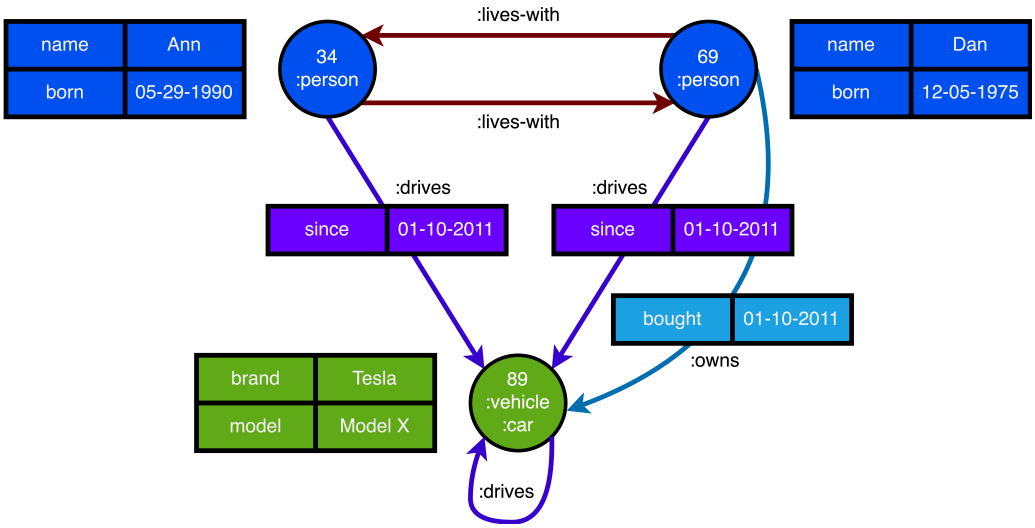

Fig. 1: Example of a property graph with three vertices and five edges (the two edges between vertices with values 69 and 89 are structurally maintained as one but can conceptually be considered two distinct edges). The tables show the properties that are defined on each vertex as well as some of the edges. The label, relationship, and property sets can be empty as is the case with `lives-with`.

dataset. An example of a property graph can be seen in Figure 1.

Given two vertices $u, v \in V$ and an edge $e \in E$ where $e = (u, v)$, then it is said that the source vertex is u and a destination vertex is v where there is a direction specified as $u \rightarrow v$. Data can be extracted from the property graph data model when given some vertex u or edge $e = (u, v)$. These operations can be thought of as queries on the data structure where the information stored at these locations is returned back to the user upon completion.

III. DI Fundamentals

DI was first introduced into Arachne by Du *et al.* [10] to allow for easy distribution of edges across a compute cluster. In this section, we will highlight the fundamentals of DI for directed graphs. DI is composed of four arrays: source (SRC), destination (DST), number of neighbors (NEI), and the starting indices (STR) into SRC and DST. The grouping of the SRC and DST arrays are referred to as the edge index arrays, whereas the grouping of STR and NEI are referred to as the vertex index arrays. The indices of the edge arrays are in the range $[0, m-1]$ and the indices of the vertex arrays are in the range $[0, n-1]$ where $m = |E|$ is the number of edges and $n = |V|$ is the number of vertices. Given an edge $e = (u, v)$, the vertices are stored in the edge arrays where $SRC[e] = u$ and $DST[e] = v$ and e is the index into the edge arrays. All the edges in SRC and DST are sorted based off the vertex values where SRC is sorted first, and then for every vertex, we sort its corresponding adjacency list in DST. The vertex index arrays are created based off the sorted edge arrays. Lastly, all the original vertex names are normalized to the range $[0, n-1]$ during construction. Storing these arrays takes up space $2(\Theta(m) + \Theta(n))$.

Given a vertex identifier u, the neighborhood of that vertex u can be found by doing the following slice $DST[STR[u]..STR[u] + NEI[u] - 1]$. The arrays of DI are distributed as blocks to the compute nodes that are allocated for the job. In Chapel, the result of an array slice is a reference to the subset of the array elements specified from the slicing index set. No new memory is ever allocated, making this memory efficient. An example showing the slicing can be found in Fig. 2. Here, we are looking for the edges of a vertex with label 50 whose neighborhood we find by taking the slice specified by the STR and NEI arrays. The edges returned would be $(50, 20), (50, 30), (50, 45), (50, 60)$. Enumerating the adjacency list of a vertex using this method takes time proportional to the number of neighbors. If a vertex u has k neighbors then the time to iterate over the adjacency list is $\Theta(k)$ and finding this list takes constant time $O(1)$.

IV. DIP Design and Development

DIP is powered by the DI data structure that currently drives graph storage in Arachne. It employs the same edge-centric view of graphs that allows for easy load-balancing across cluster (multilocale) systems in Chapel. Since DIP is designed to be written in Chapel, we will discuss operations in terms of how they are implemented in Chapel.

A. Notes On The DIP Design

Everything listed in Sec. III is applicable to DIP with the added complexity of storing multiple vertex labels, edge relationships, and properties. While designing DIP and its variations, we approached the problem in a memory-efficient manner to ensure we also matched the compactness of DI. We implemented three different methods to store property graphs based off of two-dimensional byte arrays (DIP-ARR),

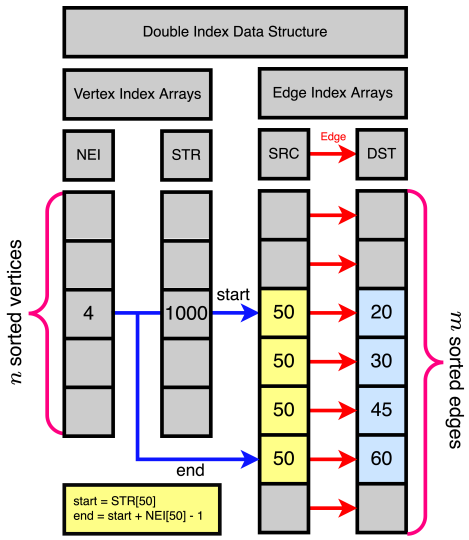

Fig. 2: Example of neighborhood slicing in DI. To get the neighborhood of the vertex with index 50, the slice is taken of $DST[STR[u]..STR[u] + NEI[u] - 1] = DST[1000..1003]$. The NEI and STR domain set is the range $[0, n-1]$ and the domain set range of SRC and DST is $[0, m-1]$. The domain map specified by those domain sets makes up the indices of those arrays.

Chapel lists (DIP-LIST), and doubly-linked lists (DIP-LISTD). In short, attributes are either represented by a two-dimensional byte array that flags whether a particular edge or vertex contains it, or by lists that maintain a copy of every attribute for each vertex and edge. For simplicity, we will focus our discussions on storing and querying edge relationships. Everything written can also be expanded to vertex labels, vertex properties, and edge properties.

B. DIP-LIST(D)

Storing attributes can also be done in an attribute-centric matter where we store each attribute for a vertex or edge explicitly, and for the case of DIP-LISTD we maintain inherent pointers to the "next" and "previous" attribute to easily extract all the vertices and/or edges that make up that attribute. This is the typical method used in many graph databases where objects represent each vertex and store all the data held by that vertex. This choice, however, is not very memory efficient as many copies must be stored for each attribute at each entity and can quickly get out of hand. More on this in Sec. IV-D.

An example of both DIP-LIST and DIP-LISTD can be seen in Fig. 3. For the case of DIP-LIST the list stored for a particular entity just contains a Chapel `string`. For the case of DIP-DLL there is also a list stored for each entity. However, it is a list of a doubly-linked list of `Node` objects that contains variables to store the data, vertex or edge it belongs to, and pointers to the next and previous elements of the doubly linked list. Since this is a distributable data structure, the previous and next pointers point to objects that can live on the different locales allocated for a job. Objects in Chapel are pointers to heap-allocated memory. Using the Chapel memory management technique called `shared`, an object can be initialized and allocated at runtime and remain in scope fully until all variables that reference that object are also in scope. Changes to the data and vertex are allowed and done through the memory management technique `borrowed` which does not delete the object when the borrowing variable goes out of scope.

Paralellizing the construction of the doubly-linked list is not straightforward, and currently items cannot be added into the list in parallel. A global `Map` called `last_entity_tracker` is maintained to track the most recently added `Node` into the list. This is done to facilitate searching the doubly-linked list when querying for all the entities that make up one attribute. During the search, different parallel tasks are spawned for each chunk of linked lists where each task can begin some search of the list, but that search is performed sequentially.

The addition of a `Node` during property graph construction requires updating the `next` pointer of the previous node and the `prev` pointer of the node being inserted. This is done by extracting the last added `Node` from `last_entity_tracker`. Once that is finished, `last_entity_tracker` is updated to include the `Node` that was just added. For this map, the key is the name of the attribute and the value is the `Node` that was just added. The addition requires calling a lock on both the map and list until the insertion operation finishes. This is done by encapsulating the code running the operation with a mutex lock created by using `sync` variables in Chapel. Currently, this is not as efficient as it could be, and more performance could be extracted explicitly creating a `Lock` class with C locks in Chapel for DIP-LISTD.

C. DIP-ARR

Unlike the list versions of DIP that were implemented, we also implemented an array-based data structure that makes indexing and slicing for data very efficient and avoids complicated class structures to represent the data and vertex stored. Further, traversing arrays is relatively cheap since they are stored contiguously in memory. In this section, we explore our array-based method of storing attributes. Simply put, for each attribute there exists a boolean array of size n or m depending on whether it is storing vertex or edge information. Then, storing that specific attribute is just storing `true` if it exists for an element (vertex or edge) and `false` otherwise. An example of this can be seen in Fig. 4.

The two-dimensional boolean byte array is broken up into chunks using the array type `domain(2) dmapped Block(0..<k, 0..<x)` in Chapel. This operation creates a blocked array with two dimensions with k rows and n columns. It is chunked in such a way by Chapel that if there were four locales then the array would be split into four quadrants, one for each locale. This would mean that no one entire attribute list for an element or element list for

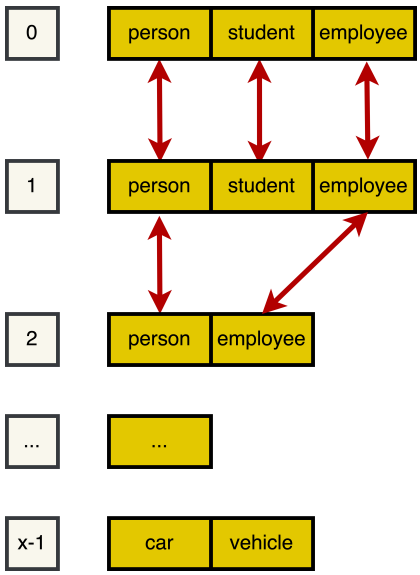

Fig. 3: Example of storing attributes in lists for each element. In this case, there are x elements where x can be either n or m depending on whether vertices or edges are being stored. In the arrows, we show the doubly-linked list for DIP-LISTD that simplifies searching for all the entities that make up a given attribute.

an attribute would be stored on the same machine. However, this should not impact performance much during querying processes since each locale keeps the index sets of its sub-elements and processes its sub-elements independently.

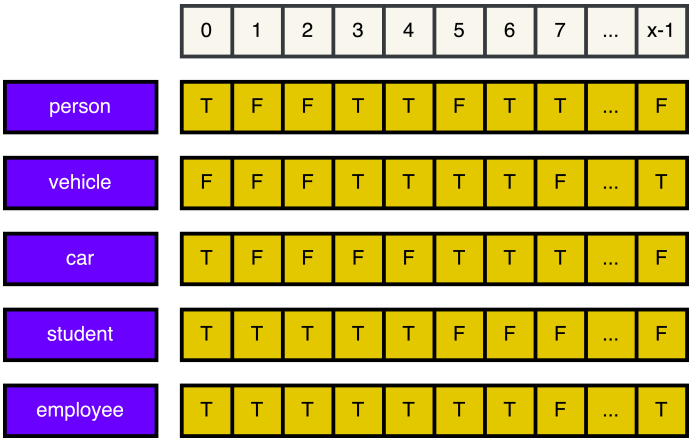

Fig. 4: Example of storing attributes as a two-dimensional byte boolean array. The number of columns is of size x which is either n or m dependent if we are storing information for the vertices or the edges. The number of attributes can be of any size k, in this case $k = 5$.

D. Space and Time Complexity Trade-offs

Each of the proposed variations of DIP supports the same fundamental operations of insertion and querying. The insertion operations are specified for inserting vertex labels, edge relationships, vertex properties, and edge properties. The querying operations are specified for returning all the attributes specified for a specific vertex or edge, or accepting a list of vertex labels, edge relationships, vertex properties, and/or edge properties and returning all the entities that contain any of them. The returned values can be further studied to find the intersections of the returned vertex and edge arrays to create a subgraph. For simplicity, we will refer to the vertices and edges as entities, and labels, relationships, or properties as attributes. The complexity analysis all comes down to be the same notation for all the attributes stored no matter the entity. We will use N to refer to the size of the entity set and K to refer to the size of the attribute set. We will use $k \leq K$ to denote the size of an attribute set for any given entity.

1) Space Complexity: DIP-LIST stores a list for each entity of size k that varies for each entity. In the worst case, each entity will contain every attribute to make the size of DIP-LIST to be $O(NK)$. DIP-LISTD stores the same list with extra data that has constant – but not neglible – size. This constant c makes the storage of DIP-LISTD to be, in the worst case, $O(NKc)$. This c will be made up of 32B for data, 16B for each vertex name integer which is doubled for edges, 8B for the previous pointer, and 8B for the next pointer. This creates a total of 80B for edge attributes and 64B for vertex attributes. Lastly, DIP-ARR stores a two-dimensional array of size $N \times K$ making its space complexity $\Theta(NK)$.

2) Space as the Number of Entities and Attributes Increase: Say we are storing a property graph with one trillion vertices and one hundred labels. Also, say we encounter the worst case specified for DIP-SLL and DIP-DLL in Sec. IV-D1. Each `Node` for entity-attribute pair would take up 64B, and there will be one hundred of them for each vertex. This will take up space $64 * 10^{12} * 10^3$ bytes which is equal to 6400 terabytes, or 6.4 petabytes. Now say we have one hundred byte vectors and the same one trillion vertices. This means we would store $10^3 * 10^{12}$ bytes which comes out to be 100 terabytes of storage, which is much less when compared to 6.4 petabytes.

3) Building Time Complexity: DIP-LIST and DIP-LISTD insert data sequentially led by parallel chunks of work. This means that we can populate two vertices u and v that live on separate chunks simultaneously, but changes to the lists for u and v must be done sequentially to avoid race conditions. This comes out to a time of $O(\frac{NKc}{P})$ where N is the number of entities, c is the overhead of inserting into a Chapel List and/or doubly-linked list, K is the number of attributes being inserted, and P is the number of processors. In the case of DIP-ARR, we don't care about race conditions since we are only flipping a switch on if we encounter that attribute for an entity. This time complexity comes out to be only $O(\frac{N}{P})$, where P is the number of parallel processing units in the system, since turning a boolean from false to true is a constant time operation.

V. Querying Data

The property graph data model allows us to search for vertices, edges, or attributes that match a particular query. These queries specified on property graphs can follow different

formats [12], but all queries boil down to simple searches on the graph data structure. Creating a data structure that allows fast and easy searching with parallel reads will increase performance as you increase the number of processors that the system runs on. Fast querying makes data analysis more interactive and improves data science workflow uptime. We will follow the same notation and worst-case scenarios as specified in Sec. IV-D.

A. DIP-LIST

Given an attribute, finding all the entities that contain it takes time $O(\frac{KN}{P})$ since every single attribute list for every entity must be traversed. The fraction $\frac{N}{P}$ breaks the data up into blocks where each search is done sequentially by the task spawned to tackle that block. This can be reduced to time $O(BK)$ since lists of size K will have to be traversed for each entity.

B. DIP-LISTD

Given an attribute, finding all the entities that contain it takes time $O(K)$ since we traverse starting from the last `Node` added into `last_entity_tracker` (see Sec. IV-B). This traversal involves parsing through previous and next pointers in the distributed memory doubly-linked list. Since Chapel objects are just pointers to a distributed heap-allocated space, jumping to an object stored on a different locale requires spawning a thread on the remote locale to process that object.

C. DIP-ARR

Given an attribute, finding all the entities that contain it takes time $O(\frac{N}{P})$ since we traverse the row for the given attribute to see which elements are `true`. This method is the nicest to parallelize since Chapel tasks run concurrently on the locale that owns a slice of the array.

VI. Related Work

The work by McColl *et al.* [14] provides a performance evaluation of open-source graph databases, where most store their data using the property graph data model. The simplest way to store graph-based data models is via a labeled property graph, which is a set of triples. The work by Angles *et al.* [2] provides a new way of viewing graph-based data called multilayer graphs that extends directed labeled graphs with edge ids.

Knowledge graphs and property graphs are graph models used to represent and organize network data. While knowledge graphs focus on representing knowledge and semantic relationships [4], property graphs concentrate on the properties and relationships of vertices and edges [18]. Both pure models have their own characteristics and are used in different contexts based on specific project requirements or application domains. There are other mixed models such as semantic property graphs [16] that we will return to in future works.

Both knowledge graphs and property graphs share similarities. Firstly, both models utilize a graphical representation, where vertices represent entities and edges represent the relationships between them. This graphical approach provides a visual representation of the data structure. Secondly, both knowledge graphs and property graphs allow for representing connections between entities and the properties associated with vertices and edges [3], [11]. This capability enables the storage and querying of detailed information about the entities and their relationships within the graph. Lastly, both knowledge graphs and property graphs are employed for data analysis and discovering hidden knowledge [13], [18]. These models support advanced queries and analysis, facilitating the extraction of meaningful information from the graph.

While there are several similarities between knowledge graphs and property graphs, it is also important to highlight their differences. Firstly, knowledge graphs are designed to represent knowledge and relationships between real-world entities [11], whereas property graphs focus on representing the properties and relationships of vertices and edges in a graph [3]. Secondly, in terms of data modeling, knowledge graphs utilize an ontology or semantic schema to define the classes of entities and the allowed relationships between them [11], while property graphs model data using vertices, edges, and properties without the need for a predefined schema [3]. Thirdly, property graphs offer more flexibility in terms of adding new properties or relationships between vertices and edges, allowing for quicker adaptation to changes in data requirements [19]. Conversely, knowledge graphs tend to be more structured and require a more rigorous definition of entity classes and relationships [8]. Lastly, knowledge graphs are often queried using standard-based query languages specifically designed for working with semantic data [8]. Property graphs, on the other hand, employ database-specific query languages such as Cypher in the case of Neo4j [9].

VII. Conclusion

Designing data structures for property graphs involves not only efficiently storing the vertices and edges of a graph, but more importantly, the attributes that are also stored with them. Oftentimes, property graph database developers want to tightly couple data with the entity, but as we show with DIP-ARR, viewing the data as a two-dimensional structure can possibly make querying faster and more memory-efficient. Further work involves large-scale benchmarking of these methods and full integration of them into Arachne.

Acknowledgment

We thank the Chapel and Arkouda communities for their guidance. This research is supported in part by the NSF grant CCF-2109988.

References

[1] Renzo Angles. The Property Graph Database Model. 2018.

[2] Renzo Angles, Aidan Hogan, Ora Lassila, Carlos Rojas, Daniel Schwabe, Pedro Szekely, and Domagoj Vrgoč. Multilayer graphs: a unified data model for graph databases. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '22, pages 1–6, New York, NY, USA, June 2022. Association for Computing Machinery.

[3] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. RDF and Property Graphs Interoperability: Status and Issues. 2019.

[4] Michael E. Bales and Stephen B. Johnson. Graph theoretic modeling of large-scale semantic networks. volume 39, pages 451–464, 2006.

[5] Bradford L Chamberlain. Chapel (Cray Inc. HPCS Language)., 2011.

[6] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[7] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, et al. Chapel comes of age: Making scalable programming productive. *Cray User Group*, 2018.

[8] David Chaves-Fraga, Kemele M. Endris, Enrique Iglesias, Oscar Corcho, and Maria-Esther Vidal. What are the parameters that affect the construction of a knowledge graph? In *On the Move to Meaningful Internet Systems: OTM 2019 Conferences*, volume 11877 of *Lecture Notes in Computer Science*, pages 11–20. Springer, October 2019.

[9] Isabelle Comyn-Wattiau and Jacky Akoka. Model driven reverse engineering of NoSQL property graph databases: The case of Neo4j. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 453–458, 2017.

[10] Zhihui Du, Oliver Alvarado Rodriguez, Joseph Patchett, and David A Bader. Interactive graph stream analytics in Arkouda. *Algorithms*, 14(8):221, 2021.

[11] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. In Michael Martin, Martí Cuquet, and Erwin Folmer, editors, *SEMANTiCS (Posters, Demos, SuCCESS)*, volume 1695 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[12] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445, Houston TX USA, May 2018. ACM.

[13] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. Querying knowledge graphs by example entity tuples. *IEEE Transactions on Knowledge and Data Engineering*, 27(10):2797–2811, 2015.

[14] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications*, PPAA '14, pages 11–18, New York, NY, USA, February 2014. Association for Computing Machinery.

[15] Michael Merrill, William Reus, and Timothy Neumann. Arkouda: interactive data exploration backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, pages 28–28, 2019.

[16] Sumit Purohit, Nhuy Van, and George Chin. Semantic property graph for scalable knowledge graph analytics. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 2672–2677, 2021.

[17] Oliver Alvarado Rodriguez, Zhihui Du, Joseph T. Patchett, Fuhuan Li, and David A. Bader. Arachne: An Arkouda Package for Large-Scale Graph Analytics. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, 2022.

[18] Sudharshan S. Vazhkudai, John Harney, Raghul Gunasekaran, Dale Stansberry, Seung-Hwan Lim, Tom Barron, Andrew Nash, and Arvind Ramanathan. Constellation: A science graph network for scalable data and knowledge discovery in extreme-scale scientific collaborations. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3052–3061, 2016.

[19] Gongsheng Yuan, Jiaheng Lu, Zhengtong Yan, and Sai Wu. A survey on mapping semi-structured data and graph data to relational data. *ACM Comput. Surv.*, 55(10), 2023.