GraphTrek: Asynchronous Graph Traversal for Property Graph-Based Metadata Management

Dong Dai[1], Philip Carns[2], Robert B. Ross[2], John Jenkins[2], Kyle Blauer[1], and Yong Chen[1]

[1]Computer Science Department, Texas Tech University, {dong.dai, yong.chen, kyle.blauer}@ttu.edu
[2]Mathematics and Computer Science Division, Argonne National Laboratory, {carns, rross, jenkins}@mcs.anl.gov

***Abstract*—Property graphs are a promising data model for rich metadata management in high-performance computing (HPC) systems because of their ability to represent not only metadata attributes but also the relationships between them. A property graph can be used to record the relationships between users, jobs, and data, for example, with unique annotations for each entity. This high-volume, power-law distributed use case is a natural fit for an out-of-core distributed property graph database. Such a system must support live updates (to ingest production information in real time), low-latency point queries (for frequent metadata operations such as permission checking), and large-scale traversals (for provenance data mining).**

Large-scale property graph traversals are particularly challenging for distributed graph databases, however. Most existing graph databases implement a "level-synchronous" breadth-first search algorithm that relies on global synchronization in each traversal step. This traversal model performs well in many problem domains; but a rich metadata management system is characterized by imbalanced graphs, long traversal lengths, and concurrent workloads, each of which has the potential to introduce or exacerbate stragglers. We define stragglers as abnormally slow steps (or servers) in a graph traversal that lead to low overall throughput for synchronous traversal algorithms.

The straggler problem can be mitigated by the use of *asynchronous* traversal algorithms. Asynchronous traversal has been successfully demonstrated in graph processing frameworks, but such systems require the graph to be loaded into a separate batch-processing framework.

In this work, we propose GraphTrek, a general asynchronous graph traversal engine working with graph databases for processing rich metadata management in their native format. We also outline a traversal-aware query language and key optimizations (*traversal-affiliate caching* and *execution merging*) necessary for efficient performance. Our experiments show that the asynchronous graph traversal engine is more efficient than its synchronous counterpart in the case of HPC rich metadata processing, where more servers are involved and larger traversals are needed.

***Keywords*-Parallel File Systems; Rich Metadata Management; Property Graph; Graph Traversal; Property Graph Databases**

I. Introduction

High-performance computing (HPC) platforms can generate huge amounts of metadata about different entities including jobs, users, and files. Simple metadata, which describes the predefined attributes of these entities (e.g., file size, name, and permissions), has been well recorded and used in current systems. Rich metadata, which describes the detailed information about different entities and their relationships, extends simple metadata to an in-depth level. Rich metadata can contain arbitrary user-defined attributes and flexible relationships. A typical example of such rich metadata is provenance, which maintains a complete history of a dataset, including the processes that generated it; the user who started the processes; and even the environment variables, parameters, and configuration files used during execution [1]. Property graph is a promising data model for rich metadata management in HPC systems because of its ability to represent not only metadata attributes but also the relationships between them. A property graph can be used to record the relationships between users, jobs, and data.

In our research, we are developing a rich metadata management system based on the new concept of unifying metadata into one generic property graph [1]. Such a system must support large-scale traversals for different usage scenarios, such as provenance data mining, hierarchical data traversal, and user audit. In order to effectively manage property graphs, distributed property graph databases have been developed, such as Neo4j [2], DEX [3], OrientDB [4], G-Store [5], and Titan [6]. In addition to storing the property graphs, a major requirement of those property graph databases is to effectively answer graph traversal queries from user applications. Graph traversal usually serves as the basic building block for various algorithms and queries. In fact, it is so fundamental that traversal of simple graphs (defined as a set of nodes connected by weighted edges) has been used as a benchmark metric (Graph500) for measuring the performance of supercomputers [7], [8]. Traversal for property graphs is likewise critical and needs to be efficiently supported.

Typically, the core execution engine of graph traversal is implemented by following the general structure of the parallel "level-synchronous" breadth-first search (BFS) algorithm, dating back three decades [9], [10]. Given a graph G, level-synchronous BFS systematically explores G from a source vertex s level by level. The *level* is the distance or hops it travels. BFS implies that all the vertices at level k from vertex s should be "visited" before vertices at level $k+1$; hence, global synchronization is needed at the end of each traversal step. The "level-synchronous" breadth-first search structure has been adopted not only in graph databases but also in many distributed graph-processing frameworks, including Pregel [11], Giraph [12], and GraphX [13]. The

Bulk Synchronous Parallel (BSP) model is popular in this context because of its simplicity and performance benefits under balanced workload.

However, such global synchronization could cause serious performance problems in our property graph-based metadata management case. First, as an online database system, our system needs to support concurrent graph traversals. The interferences among traversals easily create stragglers [14], [15], which can cause poor resource utilization and significant idling during each global synchronization. Second, the imbalance of the graph partitions, along with the possible variations in attribute sizes among different vertices and edges, leads to highly uneven loads on different servers (an indication of stragglers) while traversing. The wide existence of small-world graphs in HPC metadata (e.g., degree of vertices follows the power-law distribution [16], [1]) makes this problem even worse. Third, for heterogeneous HPC metadata property graphs, possible graph traversal steps could be much larger than the graph diameter, which traditionally limits the maximal traversal steps in simple or homogeneous graphs, for example, the six degrees of separation theory in social networks [17]. In our use case, we might check different attributes or edges in different steps. Longer traversals introduce more synchronizations and lead to a higher chance of performance penalty caused by stragglers.

Previous work suggested that asynchronous approaches have potential to minimize the effects of load imbalance across different cores in single multicore machine [18]. GraphLab [19], PowerGraph [20] and other distributed frameworks [21], [22], also have investigated the use of asynchronous execution models, which could implement the traversal operations in general. However, these approaches are more suitable for the distributed, batch-oriented graph computation that runs on the entire graph instead of interactive traveling and querying on the partially interested graphs, which are common in our HPC rich metadata management system.

In this research, we explore how to integrate an asynchronous traversal engine directly into a graph database system. We propose optimizations, including *traversal-affiliate caching* and *execution merging*, to fully exploit the performance advantage of the asynchronous traversal engine. In addition, we summarize the typical graph traversal patterns for the property graph-based rich metadata management and propose a general traversal language to describe these diverse patterns. We show that the asynchronous engine can support such language with detailed progress report functionality matching that of a synchronous engine.

The main contributions of this work are threefold:

- Analysis and summary of the graph traversal patterns in property graph databases for HPC rich metadata management. Based on these patterns, we propose a graph traversal language to support them.
- Design and implementation of an asynchronous distributed traversal engine for property graph databases. We also propose optimizations specifically designed for the asynchronous traversal engine: *traversal-affiliate caching* and *execution merging* to improve the performance.

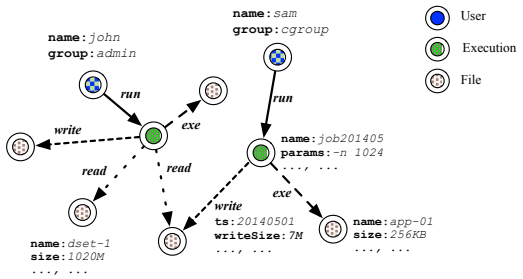

Fig. 1. An example metadata graph applied to HPC Systems.

- Evaluation and demonstration of the performance benefits compared with those of a synchronous traversal engine on both synthetic graphs and real-world graphs.

The rest of this paper is organized as follows. Section II summarizes the graph traversal pattern by analyzing HPC metadata applications. In Section III, we introduce the GraphTrek traversal language designed for these patterns, and we show how to use it to implement the given use cases. In Section IV, we describe the asynchronous traversal engine in detail, followed by several optimization strategies in Section V. In Section VI, we discuss the design choices in implementing such traversal frameworks. Section VII presents the evaluations, including comparisons with synchronous implementation and asynchronous without optimizations. In Section VIII, we present conclusions and discuss future work.

II. Traversal Patterns from Real Applications

In this section, we analyze use cases specific to HPC metadata management, which can be modeled by using a property graph. Through this analysis, we summarize the graph traversal patterns as the foundation of our proposed traversal language. A more complete discussion and analysis of these use cases can be found in our previous work [1].

A. Graphs in HPC Metadata Management

HPC metadata can be intuitively abstracted as a graphlike structure. For example, metadata—including users, executions of programs (jobs), data files accessed, or simply a directory—can be neatly mapped into different vertices in property graphs, as shown in Fig. 1. Between these entities, different interactions and relationships can be represented as different types of directed edges with properties attached. For example, the *run* edge indicates that the user started the corresponding execution instance, the *exe* edge denotes which executable file(s) an execution used, and the *read/write* edges indicate the types of operations performed on files from executions. Some entity properties are shown for vertices, such as UID/GID, file names, and parameters used by the execution. These properties are by no means exhaustive; and additional properties can easily be added, such as file permissions and creation time.

B. Use Cases

In this section we highlight examples of the types of queries that could be performed on rich metadata that is stored by using a property graph data model.

1) Data auditing: Data auditing is critical in large computing facilities where different users share the same cluster. For example, one type of audit query is the following: *Find the set of files read by a specific user during a given timeframe*. Based on the property graph abstraction, this query can be mapped to a graph traversal operation in two steps: (1) beginning at the given user, traverse the edges with the *run* property type to compute the set of executions the user has performed, filtering the results by the given time frame; and (2) traverse the *read* edges from the executions to the resulting files.

2) Provenance Support: Provenance has a wide usage in metadata management systems, including data sharing, reproducibility, and workflow management [23], [24], [25], [26]. After modeling rich metadata such as user/file system interactions as property graphs, we can answer provenance questions such as the following: *Find the execution whose model is A and input files have annotation as B* (here, *model* indicates the critical component of the execution, and *annotation* refers to the user-specified attributes on data files). This kind of query, which is a generalized version of a problem from the First Provenance Challenge [27], can be expressed as a graph traversal search that starts from *execution* vertices to *file* vertices while checking needed attributes during the traversal. In contrast to typical graph traversal operations, this query requires the source vertices (*executions*) as the returned value instead of the final *file* vertices.

C. Traversal Patterns

Many other use cases have needs similar to those of the cases discussed above. Based on these use cases, we summarize the typical traversal patterns as follows:

1) Like BFS, graph traversal starts from a set of vertices and travels in steps. In each step, since it travels through property graphs, it needs to filter these vertices according to attributes. It then selects given types of edges for the next set of vertices.
2) Unlike BFS, graph traversal may revisit the same vertex in different steps in order to check different attributes or edges. This kind of revisit is considered as cyclic or redundant in BFS, but it is acceptable in our use case. For example, the same file may first be visited as input data and then as executable again. But we do consider that revisiting on the *same* vertex in the *same* step is redundant.
3) Unlike BFS, graph traversal need not always return the destination vertices. As the provenance example shows, any vertices accessed during the traversal could be needed by users.

Based on these observations, we introduce the traversal language GraphTrek in the following section.

III. GraphTrek Traversal Language

A number of query languages either directly represent or can be used for property graphs [28]. These languages include SPARQL [29] for RDF data [30], GraphGrep for regular expression queries in graphs [31], Cypher for Neo4j graph databases [32], Gremlin from the Thinkerpop project [33], specific query language for provenance [34], Quasar for QMDS [35], and SQL (SemiJoin) for relational databases [36], [37], [38]. Among these languages, previous research [39] suggests that a low-level language like Gremlin provides better performance because it allows users to manually control each traversal step in detail. But, extremely low-level abstractions—for example, the *vertex-centric* or *edge-centric* graph programming primitives used in distributed graph-processing frameworks such as Giraph and Pregel—place too many implementation burdens on the users, who basically need to implement their own BFS algorithm using these primitives to finish a traversal. In this research, we propose a more restrictive traversal language that allows users to manually control each step while remaining simple, such that users can easily create such queries on the fly.

GraphTrek defines an iterative query-building language to represent property graph traversal operations. Currently, the language is implemented in Java. The primary class defined is called GTravel, whose methods return the caller GTravel instance to allow call chaining. Several core methods are defined in GTravel (because of space limits, we omit more functions such as *progress report*):

- *Vertex/Edge selector: v(), e()*
  The vertex selector method *v()* represents an entry point for a graph. These IDs can be initially retrieved with searching or indexing mechanisms provided by any underlying graph storage. The edge selector method *e()* selects specific edges from the working set of vertices (*frontier*) by its label argument, at the point the method call is placed in the call chain.
- *Property filters va() and ea()*
  Property filters take property key, type of filter, and comparison property values as arguments to filter out vertices and edges. The filter types currently include *EQ*, *IN*, and *RANGE*, which indicate that the given properties of vertices or edges must be *equal to* the value, *within* a set of values, or *in between* the given ranges, respectively. Note that multiple property filters can be applied in one step to filter more entities by using the *AND* operation. *OR* is not explicitly supported in the current version, but users can issue different traversals and combine their results for this purpose.
- *Return indicator rtn()*
  A return method tells the graph traversal engine that the working set of vertices at the point of the call should be returned to the user, but only for those vertices whose resulting traversals reach the end of the call chain. Normally, graph traversals return the final destination vertices or all visited vertices. But, as our HPC provenance example shows, it is useful to be able to return the intermediate results, such as executions whose connected vertices satisfy given conditions. For such traversals, we simply add *rtn()* to the call chain after the traversal step of interest, in order to return the needed vertices.

A. Traversal Commands Applied to Use Cases

Given the graph traversal language, we can easily describe the traversal operations for the use cases discussed in the preceding section.

1) Data Auditing: The query *Find all files ending in .txt read by "userA" within a timeframe* can be expressed as follows. First, a vertex selector is used to choose the right user vertex (i.e., userA). Then, the traversal follows the "run" and "read" edges with property filters applied. As *rtn()* suggests, this command will return the *file* vertices encountered.

```
GTravel.v(userA).e('run')
      .ea('start_ts', RANGE, [t_s, t_e])
      .e('read')
      .va{'type', EQ, 'text'}.rtn()
```

2) Provenance Support: The example provenance request *Find the execution whose model is A and inputs have annotation as B* is shown below. In this command, we first select all the vertices with given type (i.e., execution) and then denote that these execution vertices are the return vertices using *rtn()*. In this way, the paths that satisfy all these constraints will return their source *execution* vertices to users.

```
GTravel.v().va('type', EQ, 'Execution').rtn()
      .va('model', EQ, 'A')
      .e('read')
      .va('annotation', EQ, 'B')
```

IV. Asynchronous Traversal Execution

A graph traversal begins from the moment users submit their GTravel instances and ends when users receive all returned vertices. In this section, we introduce the proposed asynchronous traversal execution in detail.

A. Traversal Submission

The graph traversal instance (GTravel) encapsulates multiple steps into a single batch. To start a graph traversal, users build the GTravel instance by chaining different operations sequentially and then submit it to GraphTrek.

Most existing graph databases (e.g., OrientDB and Titan) simply split the multistep traversal into multiple queries. Clients issue one query each time and aggregate results together to build the next query, as Fig. 2(a) shows. We consider this as a *client-side* traversal since the client plays a central controller role during the traversal. This design usually leads to performance problems because the clients need all the intermediate results transferred from servers through the busy client-server network. Furthermore, compared with servers, the client is error-prone, thus significantly affecting the system stability.

GraphTrek relies on a different strategy, namely, *server-side* traversal. As Fig. 2b shows, the client sends the GTravel instance to one selected backend server ($s3$ in this example) to start a graph traversal. This selected server ($s3$) will serve as the *coordinator* for this traversal. The traversal is executed among backend servers and returns the status and results to the *coordinator*. In this way, server-side traversal reduces unnecessary client-server communications and takes advantage

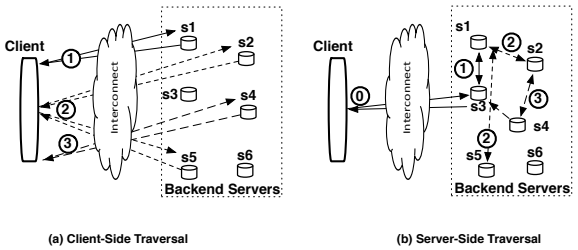

Fig. 2. Comparison of client-side traversal and server-side traversal.

of data locality and fast network transmission between backend servers. This *server-side* traversal is similar to job submission in graph-processing frameworks such as Giraph, Pregel, and GraphX.

B. Asynchronous Execution

The server-side traversal is scheduled upon the coordinator's receipt of the client's GTravel instance. The coordinator develops a multistep execution plan from the traversal command execution plan and executes it asynchronously, as shown in Fig. 3. Because of space limits, we omit the property filters on vertices and edges, which in this case are applied locally on each server.

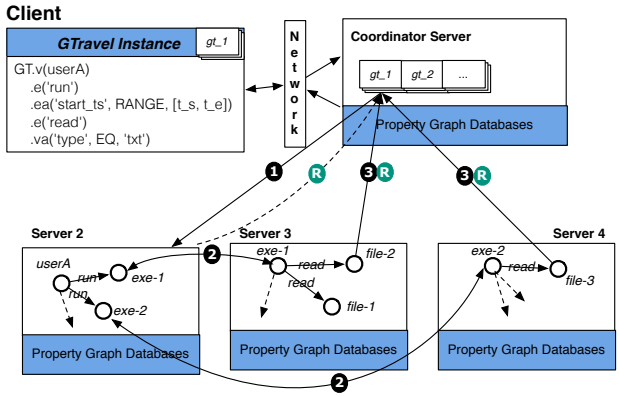

Fig. 3. Asynchronous execution of a traversal command. Numbered circles represent the order of operations, black circles represent data (vertex/edge) transfers, and green circles represent traversal status updates.

In this example, the two-step GTravel instance (gt_1) begins from *userA*. The coordinator server first learns that *userA* is stored in $server_2$ from the underlying graph databases and then sends the request to $server_2$ with an extra parameter specifying the current step *1*. Upon receiving the request, $server_2$ will iterate all the "run" edges from the given vertices and filter them based on specified filter functions. Optimizations can be applied in underlying storage for iterating edges: since we usually iterate edges by type, storing all the edges of one vertex together based on their type will provide better performance for such behavior. With the storage optimizations, the edge iteration on $server_2$ would become sequential, which could obtain the best performance on block-based storage devices.

After iterating edges, we can get a new set of vertices, which will be the starting vertices in the second step, just like *userA* in the first step. Without any synchronization, $server_2$ will concurrently send the GTravel instance (gt_1) to all servers that the vertices are stored at ($server_3$ and $server_4$ in this example). Similarly, the extra parameter that denotes the second step is also attached with the request. Upon receiving this data, $server_3$ and $server_4$ will perform the similar edge iterations to get a new set of vertices and will dispatch requests to more servers. If the current execution is the last step in the traversal command, instead of dispatching the traversal to a further step, the server will return the vertices to the coordinator server, shown as step 3 in Fig. 3. (Returning non-"end" vertices will be discussed in Section IV-D.) Once all the vertices are fully returned, the coordinator server starts to reply the client, and the whole graph traversal finishes. A buffered pipeline can be created to transfer results from the coordinator to the clients if the return dataset is too large. We left this optimization as future work.

Graph traversal in GraphTrek follows the breadth-first structure: each vertex iterates all its neighbors in parallel, and there is no accessing order among these neighbors. However, different from traditional BFS implementation, which needs synchronization among different steps, our proposed design and implementation allow each server to start the next step without explicit synchronizing with other servers. Hence, it can coordinate the unnecessary waiting for the slow servers and provides a better overall performance.

C. Status and Progress Tracing

In asynchronous graph traversal, each server independently executes its actions and spreads the traversal to more servers if needed. No global traversal status can be obtained. This introduces *correctness concern* like silent failure, which means that if the asynchronous execution fails, the system may not be informed. Because of the existence of such failures, the coordinator server will not be able to decide whether the entire traversal has finished correctly or not. In GraphTrek, we introduce a status-tracing mechanism to identify failures to help guarantee the traversal correctness. We leave the implementation of full fault tolerance features like restarting traversal from where it failed as the next step.

Consider one backend server as an example. During the traversal, it repeats the same operation: it receives the GTravel instance from another server, performs the needed vertex and edge filtering to get a new set of vertices, then concurrently sends the traversal instance to more servers according to the new set of vertices. We consider this whole procedure on a specific server as one *traversal execution*. A asynchronous graph traversal consists of many such concurrent *traversal executions*. Intuitively, tracing the status of each execution will give us a global view of the traversal. To trace each execution, in GraphTrek, we log the creation and termination events of executions in the *coordinator* server. If any execution was logged as created but did not terminate (as the result of a timeout or similar reasons), we consider that the server failed. Since we currently do not implement a full fault tolerance feature, this failure will simply cause the traversal to be restarted. We leave the fine-grained failure recovery as future work.

In Fig. 3, the green circles show the example tracing reports from the backend servers to the coordinator during graph traversal. Whenever one server successfully sends the GTravel instance to other servers to start the next step, it will report an *execution creation* event to the coordinator telling it that the new execution is created in the target servers. In addition, after the GTravel instances have been successfully sent, the server will report the *execution termination* event denoting its own termination. An execution will not be considered finished in the coordinator unless it has registered all its downstream executions in the coordinator server and has reported its own termination. Similarly, a graph traversal does not finish unless all the executions created are marked as terminated in the coordinator server.

The status reports of the *traversal executions* from the backend servers also help track the traversal progress. Although it is not feasible to have the exact current step of the traversal as it is executed in an asynchronous way, the count of current unfinished *traversal executions* in each step can still help users estimate the remaining work and time.

D. Traversal Return

Traversal stops when it reaches the last step of GTravel instance. Typically, these final executions will transmit the final vertices to the coordinator and to the clients. But as *rtn()* suggests, GraphTrek also allows users to return the intermediate or even the source vertices. In order to support such functionality, each time a backend server starts a traversal execution, it will check whether the generated vertices are marked as returned by users. If yes, the server will change the report destination of all the downstream traversal executions to help return the needed vertices.

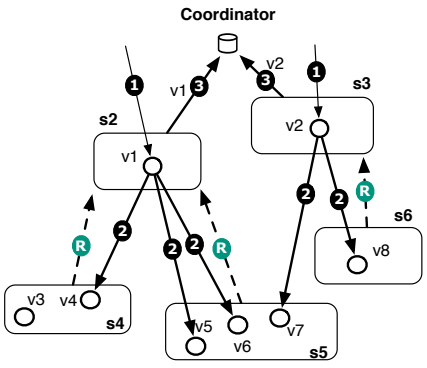

Fig. 4. Example of a graph traversal that returns the intermediate vertices.

As Fig. 4 shows, changing the "reporting destination" causes the graph traversal to execute in a slightly different way. Assume that the vertices (v_1 and v_2) are generated in a certain step of the traversal, which is marked with *rtn()*. Then, the servers actually storing v_1 and v_2 will force the

downstream servers to change their "reporting destination" from *coordinator* to themselves. The servers that execute the last step will send their final results to these "reporting destinations." For example, as step *R* shows, after one more step traversal, the backend servers return to $s2$ and $s3$ instead of the coordinator server. When replies arrive, servers ($s2$ and $s3$) will know the status of downstream executions and will send the vertices to the coordinator server accordingly. In this way, we can return vertices in arbitrary steps in the graph traversal.

V. Asynchronous Traversal Optimizations

To achieve the best performance for asynchronous graph traversal, we introduce several critical optimization.

A. Traversal-Affiliate Caching

One potential drawback of asynchronous traversal is the redundant vertex visit. Unlike the repeated vertex visit introduced in Section II, these redundant vertex visits are from the same step on the same vertex and are triggered by asynchronous execution. In Fig. 5, we show an example of such a scenario: three different paths arrive at v_H in the same step starting from v_a. These three paths (i.e., $a \rightarrow c \rightarrow H$, $a \rightarrow d \rightarrow H$, and $a \rightarrow e \rightarrow H$) go through two servers. Because of the asynchronous execution model, they may arrive at three different times and cause redundant disk I/Os. This drawback wastes precious I/O bandwidth, leading to a performance problem.

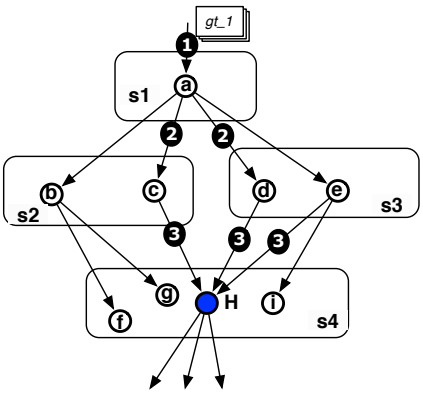

Fig. 5. Example of a redundant vertex visit in an asynchronous graph traversal: *a, b, ...* are vertices. *S1, S2, ...,* are backend servers.

To avoid this problem, we introduce a traversal-affiliate cache. In each backend server, a preallocated cache is created once the servers start. During the graph traversal, the server caches the current execution into this buffer with the identification of a {travel-id, current-step, vertex-id} triple. While serving a new request, the server first checks whether it has been served before by querying the cache. If there is a cache hit, then the server can safely abandon the request. By doing so, we also avoid spreading graph traversal multiple times.

Although the traversal-affiliate caching buffers only three-element triples, complex and concurrent graph traversal requests can still fill it up. To substitute the cached elements, we use the time-based replacement strategy: for each traversal instance, the triples with the smallest step Ids are substituted. The rationale comes from the fact that the existence of a larger step Id indicates that the oldest steps are already finished. This is still true even under the asynchronous execution model. The distance between the largest step and the smallest step is controlled by using an optimized execution scheduling and merging strategy introduced in next subsection.

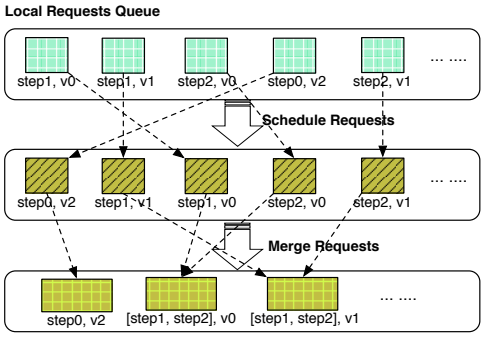

Fig. 6. Request queue in local server under scheduling and merging.

B. Execution Scheduling and Merging

In the proposed asynchronous graph traversal, each server receives a traversal instance and current step from its ancestor servers. It puts the received requests into a local queue and replies to the ancestor servers before processing these requests. In this way, the ancestor servers can finish asynchronously, and the local server will have a number of buffered requests. A pool of *worker* threads is waiting on this queue for new requests. This leads to a dynamic queue size in each server: if a server is slower or with heavier loads, its internal queue is longer, and more requests are buffered. This presents an opportunity to improve performance via scheduling and merging. Otherwise, the queue is shorter, and the server responds quickly for new requests.

A *worker* thread takes one queued request at a time for processing. The upper queue in Fig. 6 shows an initial status after receiving a number of requests from ancestor servers. We show only a single graph traversal in this figure. During scheduling, the *worker* thread always chooses the request with the smallest step Id in the queue. In this way, all the requests are ordered by their step Ids, as the middle queue in Fig. 6 shows. Using this execution scheduling strategy, we can process the slow steps with higher priority in order to help them catch up. This approach also helps control the maximal step difference between the fastest one and the slowest one in the traversal, thus reducing the traversal-affiliate cache usage.

In addition to execution scheduling, we introduce execution merging. As Fig. 6 shows, we consolidate different steps on the same vertex; for example, traversal executions on $v2$ for step 1 and step 2 will be combined as one disk request. In this way, we need only to retrieve the vertex attributes or to scan its edges once locally. This optimization significantly reduces the amount of disk I/O.

The scheduling and merging on the buffered queue provide

an automatic load-balancing mechanism among asynchronous executions inside the same graph traversal. If executions are slower because of stragglers, more requests will be buffered in the queue, providing an opportunity to schedule and merge executions more efficiently. These optimizations significantly improve the execution of asynchronous graph traversals.

VI. Discussion of Implementation and Evaluation

We have designed GraphTrek as a graph traversal engine working as a standalone component along with the backend storage systems (i.e., the property graph databases). As Fig. 3 shows, the asynchronous graph traversal engine runs on each backend server based on the graph databases instances. These traversal engine components communicate with each other through RPC calls and retrieve data and information from the local daemon of the underlying graph storage systems. The information that the graph storages provide mainly includes the location of a given vertex and edges.

Although numerous distributed property graph databases have been proposed and developed, they are all general graph databases without considering the requirements of our HPC metadata management use case. For example, Titan stores the property graphs on general column-based NoSQL storage systems such as HBase [40] or Cassandra [41], where all vertices are mapped as different rows; edges and attributes are mapped as separate columns in the same row; key-value storages were also used to implement graph functionalities [42]. Systems such as Noe4j store the graph structure and attributes separately in order to gain performance on queries of graph structure. In this paper, we evaluated GraphTrek based on our own concise but complete graph storage system developed for HPC metadata management [43].

Specifically, we map the attributes and the connected edges of a vertex into different key-value pairs that are sequentially stored for better scan performance. Also, the same type of edges are stored together; different types of vertices are mapped into key-value pairs in separate namespaces. In each server, we deploy RocksDB [44] to store these key-value pairs persistently. The graph traversal engine communicates with each other through RPC calls, which are implemented by ZeroMQ [44] as a high-speed network transmission protocol.

In addition to different storage layouts, graph partition strategies can make a huge difference in the performance on the graph traversal. From the commonly used *edge-cut* partition, which places the vertices across different servers by their hash values, to the *vertex-cut* partitioning strategies, which places the edges to different servers with a considerable amount of variations [45], [20], [46], many design choices have to be made based on the use cases. However, due to inevitable interferences, even with the best load-balanced strategy, stragglers will still exist and asynchronous execution is still a key to minimize the affects of such stragglers. To simplify, in this study, we focus on the *edge-cut* partition, as most graph databases do.

For comparison, we also implemented synchronous graph traversal in our framework for use as a baseline. In a synchronous graph traversal, a control server typically is used to synchronize each step of the traversal. This control server can be a client or a selected backend server. Using a client as a controller makes traversal is more vulnerable to failures and has worse performance because of the slow client-server communication. Thus, in our synchronous graph traversal implementation, we follow the same *server-side* traversal design to obtain a fair comparison. The execution of the synchronous traversal is straightforward. Each time, the controller makes sure that all previous executions have finished and then starts the next step. In order to obtain the best performance, the data flows are transferred between involved backend servers without going through the controller. Each server waits for the signal from the controller to start the next step, in order to realize global synchronization between sequential steps.

VII. Evaluation

In this section, we evaluate the performance of GraphTrek on synthetic graphs and on a real-world HPC rich metadata management use case. We implement the synchronous graph traversal (denoted Syn-GT), plain asynchronous traversal without any optimizations (denoted Asyn-GT), and GraphTrek (denoted GraphTrek) for comparison.

On the hardware side, all evaluations were conducted on the Fusion cluster at Argonne National Laboratory. It contains 320 nodes, and we used 2 to 32 nodes as backend servers in these evaluations. Each node has a dual-socket, quad-core 2.53 GHz Intel Xeon CPU with 36 GB memory and 250 GB local hard disk. All nodes are connected by high-speed network interconnection (InfiniBand QDR 4 GB/s per link, per direction). The global parallel file system includes a 90 TB GPFS file system and a 320 TB PVFS file system.

On the software side, GraphTrek servers are currently able to run using either local storage or parallel file systems by changing the locations of RocksDB database files. They can be placed on local disks for better performance or on parallel file systems (e.g., GPFS in the Fusion environment) for fault tolerance against server failures. In the following evaluations, unless explicitly pointed out, we evaluated GraphTrek atop GPFS as an initial step toward a fault tolerance strategy. In fact, in production, the HPC rich metadata should be kept in a resilient manner, so the performance on GPFS gives a more reasonable performance estimation than does using local disks (around 10% performance benefits of local disks for both synchronous and asynchronous traversal is detected during our evaluation).

For experiments with synthetic graphs, we used scale-free graphs generated by the RMAT graph generator [47]. The RMAT graph generator uses a "recursive matrix" model to create graphs that model real-world graphs as social network graphs. We generated directed property graphs with 2^{20} vertices and an average out-degree of 16. The vertices and edges in these synthetic graphs are the same type, with randomly generated attributes attached (the attribute size is 128 bytes). The graph (denoted RMAT-1 graph) was generated with parameters $a = 0.45, b = 0.15, c = 0.15$, and $d = 0.25$,

which create a power-law graph with moderate out-degree skewness. In addition to this parameter set, we tried different RMAT configurations during the evaluation. They all generated largely similar results, so we included only this single set because of space limitations.

We execute 2-step, 4-step, and 8-step graph traversal examples starting from the same randomly selected vertex on 2 to 32 backend servers. No extra workloads are generated on any backend servers during the experiments; any workload imbalance is due to the properties of the graph itself. All evaluations are carried out from a cold start in order to force disk access in the traversal engine. Note that the graph size is held constant as we vary the number of servers. In larger-scale experiments each individual server therefore stores fewer vertices and edges.

A. Sensitivity to Asynchronous Traversal Optimizations

We begin our evaluation by investigating the impact of the asynchronous traversal optimizations described in Section V. GraphTrek introduces two optimizations, traversal-affiliate caching and execution scheduling/merging, to improve the performance of asynchronous graph traversal. To verify the benefits of these optimizations, we evaluate the performance analysis of Sync-GT, ASync-GT, and GraphTrek in a specific case, an 8-step graph traversal on the RMAT-1 graph, as shown in Table I. Other examples are omitted since they show similar results.

TABLE I
PERFORMANCE COMPARISON ON RMAT-1 GRAPH

No. Servers	Sync-GT	Async-GT	GraphTrek
2	47.8 s	63.7 s	45.2 s
4	28.5 s	33.1 s	22.5 s
8	17.1 s	20.6 s	13.4 s
16	10.3 s	12.1 s	8.3 s
32	7.2 s	7.4 s	5.6 s

In this series of tests, the Async-GT is the plain asynchronous engine without optimizations. As the results indicate, it has worse performance than both the GraphTrek and Sync-GT. Since the Async-GT and GraphTrek are implemented in a similar manner, the main performance difference comes from the proposed optimizations. To further understand the benefits of these optimizations, we placed instruments inside the GraphTrek engine to collect the statistics during the execution. In each server, we collected three statistics: (1) *redundant visits*, which indicates the number of repeated vertex requests detected by the traversal-affiliate caching; (2) *combined visits*, which counts the number of vertex requests that can be combined together by the execution merging; and (3) *real I/O visits*, which counts the real vertex accesses to backend storage systems. The sum of these three numbers equals the total vertex requests received in one server during the traversal. Figure 7 shows the results of a typical run of an 8-step graph traversal on 32 servers (servers are reordered for better presentation).

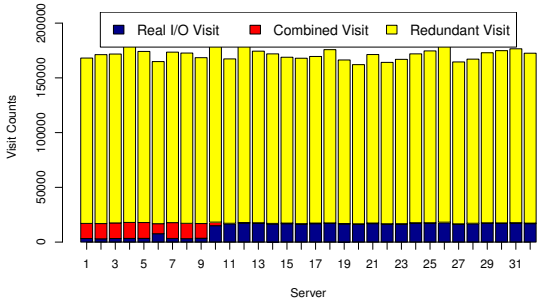

Fig. 7. Statistics collected from an 8-step traversal on 32 servers.

These statistics show a significant reduction from the received requests to real I/O visits as a result of the proposed optimizations. The redundant vertex visits actually dominate the majority of received requests. Traversal-affiliate caching can effectively remove them in each server, thereby boosting performance. On the other hand, another optimization—execution merging—has different impacts on the different servers: the *combined visit* is much more obvious in the first 10 servers than in the other servers. From an in-depth analysis of these 10 servers, we found that they actually stored more high-degree vertices. Because of this imbalance of graph structure, they were much slower than the other servers while serving the same number of vertex requests. Thus, in an asynchronous engine, the local queue can buffer more requests, and the execution merging was able to merge more vertices together. So, as Fig. 7 shows, these servers end up with fewer real vertex requests and hence can catch up with other servers. This optimization significantly reduces the actual disk I/Os and helps improve the overall performance.

Based on these results, we omit the Async-GT evaluation from all subsequent experiments and focus on the comparison between asynchronous traversal with optimizations (GraphTrek) and synchronous traversal (Sync-GT).

B. Synthetic Workloads

In this section we use a wider sampling of synthetic workloads. The results obtained by using RMAT-1 are shown in Fig. 8 to Fig. 10. The number of servers is shown on the x-axis, while the y-axis shows the elapsed time (ms) for graph traversal requests.

From these figures we see that for graph traversals with smaller steps and fewer servers, the synchronous implementation actually performs better than does the GraphTrek, as Fig. 8 shows. The reason is that the short traversal does not provide enough optimization opportunities for asynchronous executions. GraphTrek's relative performance improves when more servers are involved in the traversal and the potential for stragglers increases. Figure 9 and Figure 10 also illustrate that GraphTrek performs well with more traversal steps. For example, in Fig. 10, with an 8-step graph traversal, the performance improvement over 32 servers was around 24%, compared with the 5% improvement over 2 servers. The increased number of traversal steps (as would be common in HPC metadata management use cases) also significantly increases the potential for straggler servers to affect performance. With

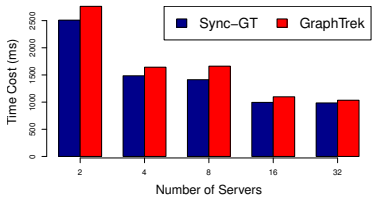

Fig. 8. 2-step graph traversal on RMAT-1.

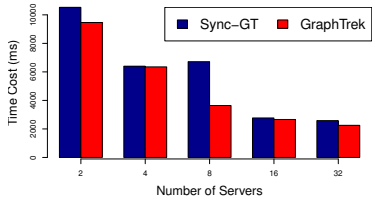

Fig. 9. 4-step graph traversal on RMAT-1.

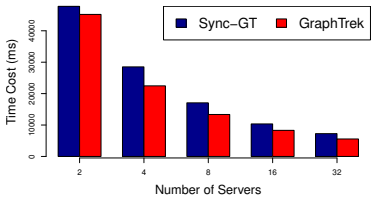

Fig. 10. 8-step graph traversal on RMAT-1.

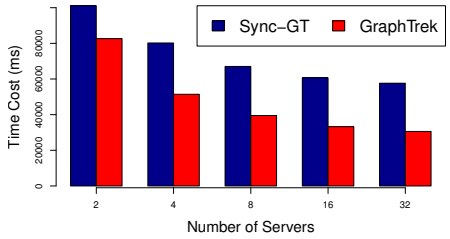

Fig. 11. Performance comparison with simulated external stragglers. Each bar shows an average of three runs.

different RMAT graphs, a similar performance pattern can be observed.

C. Synthetic Workloads with External Interference

Servers may experience transient straggling behavior because of concurrent I/O activity from other traversals or external applications. To investigate the performance impact on the traversal engine, we emulated this phenomenon by inserting fixed (50 ms) delay into individual vertex data accesses. Each time, multiple delays (500 times, indicating 500 slow vertices accesses) were created to emulate a straggler that lasts a certain period of time. By creating fixed delays, we can make sure that the two traversal engines are facing the same amount of external delays. During the whole traversal, three stragglers were created in three selected servers at chosen steps (step 1, 3, and 7). Specifically, we created one straggler chosen by round-robin from these three selected servers in each step. Figure 11 shows how this affected performance for an 8-step RMAT-1 graph traversal.

The results suggest an obvious performance advantage of GraphTrek (2x with 32-server) compared with synchronous solutions. The asynchronous traversal can make productive traversal progress despite the presence of external interference because it does not require synchronization after each step of the traversal. The execution scheduling and merging optimizations also allow straggling servers to more quickly catch up with the other servers.

D. HPC Metadata Management Workloads

We also evaluated the impact of the proposed asynchronous graph traversal and the GraphTrek framework for real HPC rich metadata management use cases. To build the heterogeneous property graph, we imported one year of Darshan traces (2013) from the Intrepid supercomputer at Argonne National Laboratory into a property graph for the evaluation [48], [49]. This collection of Darshan logs characterizes the I/O activity of approximately 42% of all core-hours consumed on the Intrepid over the course of a year. Statistics for the generated graph are listed in Table II. Our previous work shows that this rich metadata graph is also a small-world graph with a power-law distribution [1].

TABLE II
STATISTICS OF RICH METADATA GRAPH

No. of Users	Jobs	Executions	Files	Edges
177	47600	123.4 Millions	34.6 Millions	239.8 Millions

Because of page limit constraints, we show only one example data auditing query and its performance. This query is used for analyzing the influence of a suspicious user on the system. It lists all files that were written by executions whose input files are suspicious. The graph traversal can be expressed as follows.

```
GTravel.v(suspectUser).e('run')
    .ea('ts', RANGE, [ts, te])  //select jobs
    .e('hasExecutions')  //select executions
    .e('write')        //select outputs
    .e('readBy')       //select executions
    .e('write').rtn(); //outputs of executions
```

Running this request for a randomized user on 32 servers with different graph traversal implementations has different performance results, as reported in Table III. Similar to synthetic graphs, the GraphTrek clearly outperforms the synchronous design and approach.

TABLE III
PERFORMANCE COMPARISON ON DARSHAN GRAPH

No. Servers	Sync-GT	Async-GT	GraphTrek
32	3575 ms	4159 ms	2839 ms

VIII. CONCLUSION AND FUTURE WORK

Motivated by the needs of graph-based HPC rich metadata management use cases, we have proposed a graph traversal language to help describe complex queries. We also introduced an asynchronous graph traversal engine, GraphTrek, in order to avoid the performance bottleneck caused by stragglers while traveling through property graphs. To achieve better asynchronous traversal performance, we proposed two critical optimizations, traversal-affiliate caching and execution merging, for the asynchronous traversal design. A performance

comparison of the synchronous and asynchronous traversal engines on both synthetic datasets and real-world workloads confirms that, for larger systems with deeper traversals, the proposed asynchronous engine achieves better performance than does the traditional synchronous approach.

For future work, we will focus on the fine-grained fault tolerance capability in order to make the traversal capable of restarting from where it fails and also explore possible optimizations including automatic load balancing to further improve the performance.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Defense; by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357; and by the National Science Foundation under grant CCF-1409946 and CNS-1338078.

REFERENCES

[1] D. Dai, R. B. Ross, P. Carns, D. Kimpe, and Y. Chen, "Using Property Graphs for Rich Metadata Management in HPC Systems," in *Parallel Data Storage Workshop (PDSW), 2014 9th*. IEEE, 2014, pp. 7–12.

[2] J. Webber, "A Programmatic Introduction to Neo4j," in *Proceedings of the 3rd annual conference on Systems, Programming, and Applications: Software for Humanity*. ACM, 2012, pp. 217–218.

[3] "DEX," http://www.sparsity-technologies.com/.

[4] "OrientDB," http://www.orientechnologies.com/orient-db.htm.

[5] R. Steinhaus, D. Olteanu, and T. Furche, "G-Store: A Storage Manager for Graph Data," Ph.D. dissertation, Citeseer, 2010.

[6] "Titan," http://thinkaurelius.github.io/titan/.

[7] D. Bader and K. Madduri, "Design and Implementation of The HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," *High Performance Computing–HiPC 2005*, pp. 465–476, 2005.

[8] "The Graph 500 List," http://www.graph500.org/results.

[9] E. Reghbati and D. G. Corneil, "Parallel Computations in Graph Theory," *SIAM Journal on Computing*, vol. 7, no. 2, pp. 230–237, 1978.

[10] M. J. Quinn and N. Deo, "Parallel Graph Algorithms," *ACM Computing Surveys (CSUR)*, vol. 16, no. 3, pp. 319–348, 1984.

[11] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a System for Large-Scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[12] A. Ching, "Giraph: Production-Grade Graph Processing Infrastructure for Trillion Edge Graphs," in *ATPESC*, ser. ATPESC '14, 2014.

[13] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*, 2013.

[14] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in The IO Performance of Petascale Storage Systems," in *Proceedings of the 2010 ACM/IEEE SC*. IEEE Computer Society, 2010, pp. 1–12.

[15] D. Dai, Y. Chen, D. Kimpe, and R. Ross, "Two-choice Randomized Dynamic I/O Scheduler for Object Storage Systems," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 635–646.

[16] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On Power-Law Relationships of the Internet Topology," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[17] "Six Degrees of Separation," http://en.wikipedia.org/wiki/Six-degrees-of-separation.

[18] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance computing, networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "Graphlab: A New Framework for Parallel Machine Learning," *arXiv preprint arXiv:1408.2041*, 2014.

[20] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Power-Graph: Distributed Graph-Parallel Computation on Natural Graphs," in *OSDI*, 2012.

[21] D. Dai, Y. Chen, D. Kimpe, R. Ross, and X. Zhou, "Domino: an incremental computing framework in cloud with eventual synchronization," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014, pp. 291–294.

[22] X. Shi, J. Liang, S. Di, B. He, H. Jin, L. Lu, Z. Wang, X. Luo, and J. Zhong, "Optimization of Asynchronous Graph Processing on GPU with Hybrid Coloring Model," in *Proceedings of the 20th ACM PPoPP'15*.

[23] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and Where: A Characterization of Data Provenance," in *Database Theory ICDT 2001*.

[24] D. Dai, Y. Chen, D. Kimpe, and R. Ross, "Provenance-based Object Storage Prediction Scheme for Scientific Big Data Applications," in *Big Data, 2014 IEEE International Conference on*. IEEE, 2014.

[25] Y. L. Simmhan, B. Plale, and D. Gannon, "A Survey of Data Provenance in e-science," *ACM Sigmod Record*, vol. 34, no. 3, pp. 31–36, 2005.

[26] C. T. Silva, J. Freire, and S. P. Callahan, "Provenance for Visualizations: Reproducibility and Beyond," *Computing in Science & Engineering*, vol. 9, no. 5, pp. 82–89, 2007.

[27] "Provenance Challenges," http://twiki.ipaw.info/bin/view/Challenge/.

[28] R. Angles and C. Gutierrez, "Survey of Graph Database Models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, p. 1, 2008.

[29] E. PrudHommeaux, A. Seaborne *et al.*, "SPARQL Query Language for RDF," *W3C recommendation*, vol. 15, 2008.

[30] F. Manola, E. Miller, B. McBride *et al.*, "RDF Primer," *W3C recommendation*, vol. 10, no. 1-107, p. 6, 2004.

[31] R. Giugno and D. Shasha, "Graphgrep: A Fast and Universal Method for Querying Graphs," 2002. [Online]. Available: http://dx.doi.org/10.1109/ICPR.2002.1048250

[32] "Cypher," http://neo4j.com/developer/cypher-query-language/.

[33] "Gremlin," https://gremlindocs.com/.

[34] D. A. Holland, U. J. Braun, D. Maclean, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Choosing a Data Model and Query Language for Provenance," 2008.

[35] S. Ames, M. Gokhale, and C. Maltzahn, "QMDS: A File System Metadata Management Service Supporting a Graph Data Model-based Query Language," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 28, no. 2, pp. 159–183, 2013.

[36] N. Martinez-Bazan and D. Dominguez-Sal, "Using Semijoin Programs to Solve Traversal Queries in Graph Databases," in *Proceedings of Workshop on Graph Data management Experiences and Systems*, 2014.

[37] L. Zou, L. Chen, and M. Ozsu, "Distance-join: Pattern Match Query in A Large Graph Database," 2009. [Online]. Available: http://dx.doi.org/10.14778/1687627.1687727

[38] H. He and A. K. Singh, "Graphs-at-a-time: Query Language and Access Methods for Graph Databases," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[39] F. Holzschuher and R. Peinl, "Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4j," 2013. [Online]. Available: http://dx.doi.org/10.1145/2457317.2457351

[40] A. Khetrapal and V. Ganesh, "HBase and Hypertable for Large Scale Distributed Storage Systems," *Dept. of Computer Science, Purdue*, 2006.

[41] A. Lakshman and P. Malik, "Cassandra: a Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, 2010.

[42] D. Dai, X. Li, C. Wang, M. Sun, and X. Zhou, "Sedna: A Memory Based Key-Value Storage System for Realtime Processing in Cloud," in *IEEE Cluster Workshops*, 2012.

[43] D. Dai, "GraphMeta Prototype," http://discl.cs.ttu.edu/gitlab/dongdai/graphfs.

[44] "RocksDB," http://rocksdb.org/.

[45] M. Kim and K. S. Candan, "SBV-Cut: Vertex-cut Based Graph Partitioning Using Structural Balance Vertices," *Data & Knowledge Engineering*, vol. 72, pp. 285–303, 2012.

[46] A. Abou-Rjeili and G. Karypis, "Multilevel Algorithms for Partitioning Power-Law Graphs," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*. IEEE, 2006, p. 10.

[47] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," vol. 4, pp. 442–446, 2004.

[48] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[49] "Darshan data," ftp://ftp.mcs.anl.gov/pub/darshan/data/.