Toward Lazy Evaluation in a Graph Database

Jeffrey Eymer
SUNY Binghamton
jeymer1@binghamton.edu

Philip Dexter
SUNY Binghamton
pdexter1@binghamton.edu

Yu David Liu
SUNY Binghamton
davidl@binghamton.edu

Abstract

For graph databases deployed in a server setting, high throughput is a critical design goal in performance optimization. With larger data, individual queries incur longer latency, which in turn lowers throughput. This paper explores a novel throughput optimization called *in-graph batching* empowered by *lazy evaluation* in data processing. By delaying queries and allowing them to be propagated in the graph, multiple queries can be batched together during propagation, effectively sharing the latency cost. We implement our idea in Neo4j, one of the most widely used graph databases. For graphs consisting of 100 million nodes, our preliminary implementation shows an average increase in throughput of about 282% compared to the default data processing of Neo4j. More encouragingly, the throughput improvement appears to scale with the data size.

1. Introduction

Graph databases are widely deployed in cloud-computing servers and data centers. Neo4j (neo 2010) is the most popular graph database (dbe 2019) with real-world deployments, e.g., Walmart and Ebay (neo 2019). In this paper, we explore a novel notion of batching — inspired by lazy evaluation in programming languages — to support throughput optimization in Neo4j.

Batching is a classic and simple form of optimization in databases. When multiple database operations such as queries or updates are issued, processing them individually would lead to multiple "round trips" of database access, each of which at worst leads to the traversal of the entire graph. Existing systems may allow multiple operations to be combined together so that only one "round trip" is needed. This form of batching is usually applied at the client-database boundary, i.e., at the root of the graph. In practice, this form of "all-or-nothing" batching results in a fundamental tension in design between how large the batch size is and how long it takes to form a batch; even though it is desirable to have a larger batch to reduce the number of "round trips", the system may waste time in waiting at the root of the graph.

In this paper, we resolve the tension by introducing *fine-grained in-graph batching*. Instead of considering batching only at the root of the graph, we allow delayed operations to be distributedly propagated through the graph and batching may happen at any node and any time when systems have more resources.

Our design takes inspiration from lazy evaluation such as call-by-name and call-by-need reduction in the λ calculus, where the lazily applied evaluation may be propagated through the expression structure in a fine-grained manner. In essence, *lazy evaluation meets graph databases* in this paper.

2. In-Graph Batching for Neo4j

In Figure 1, we illustrate a Neo4j graph with six nodes, whose traversal order follows 1, 2, 3, 4, 5, 6. In our design, we associate the traversal sequence with a *shadow list*, each node of which conceptually serves as a decorator (Gamma et al. 1995) of the node in the Neo4j graph. Intuitively, each node in the shadow list represents a delayed operation. For example, the shadow node `find 2` means that an operation has been issued to the graph database looking for a node with key 2. Conceptually, our goal is to have such delayed operations gradually propagate through the graph following the traversal sequence. However, with the shadow list design, the same propagation can be mimicked through updating the decoration references. For example, `find 2` operation is currently propagated to node 1. If it is subsequently propagated to node 2, we update the decorator reference for `find 2` to node 2. One attractive benefit of the shadow list design is that it completely decouples our run-time support of lazy data processing from the object graph formed by the Neo4j graph database.

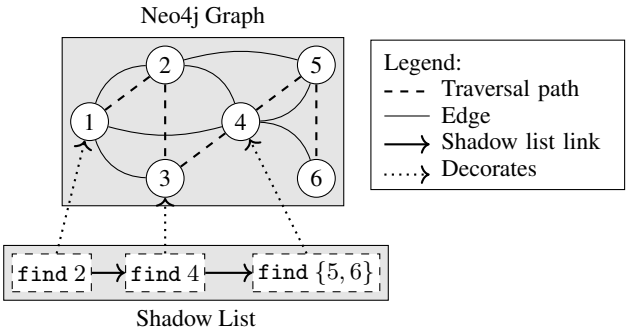
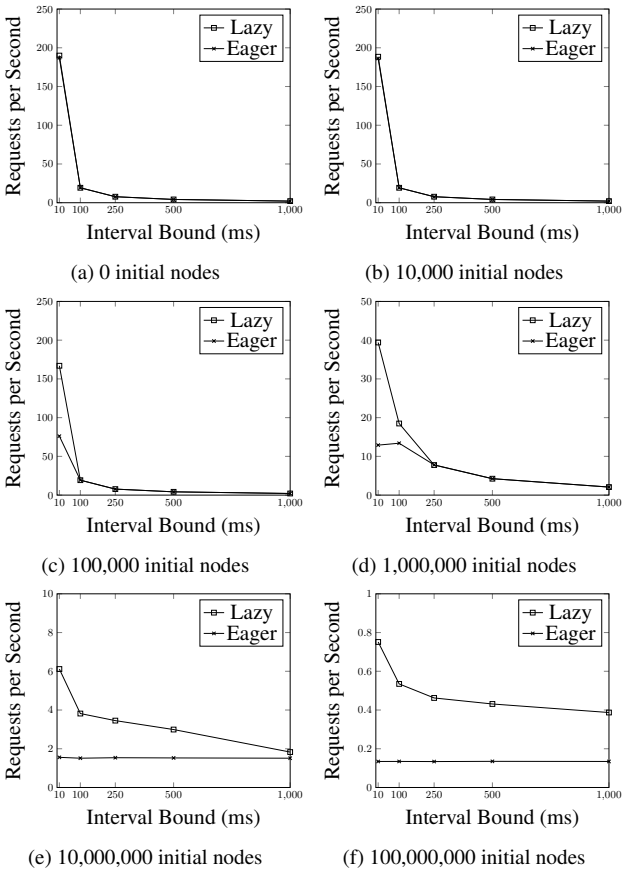


Figure 1: Delayed Operations in the Shadow List

One important observation is that delayed operations are maintained in a fine-grained manner: delayed operation `find 4` currently decorates node 3, indicating that it has been propagated "halfway" through the graph. This is in contrast to "all or nothing" batching systems where operations may only be delayed at the root of the graph. Our fine-grained nature of lazy propagation is more aligned with lazy evaluation in programming languages. Whereas lazy programming languages propagate evaluation over the structure of the program, our design propagates delayed operations through the graph.

A natural consequence of lazy in-graph propagation is in-graph batching. When multiple operations decorate the same node, our system may batch them together for their propagation in the remaining traversal sequence. For example, `find 5,6` in Figure 1 means that a batched operation is formed to find both node 5 and 6. A batched operation may lead to savings in propagation. For example, to further propagate `find 5,6` from node 4 to node 5 only requires one "hop" in the traversal instead of two.



(a) 0 initial nodes (b) 10,000 initial nodes

(c) 100,000 initial nodes (d) 1,000,000 initial nodes

(e) 10,000,000 initial nodes (f) 100,000,000 initial nodes

Figure 2: Lazy/Eager Processing (The x-axis represents the interval bound for the arrival of the requests. A number x represents that requests arrive at a random interval which forms a uniform distribution between [0,x]. The y-axis represents the throughput of query results.)

3. Preliminary Evaluation

We have implemented our shadow list design on Neo4j. In this section, we report a preliminary evaluation on the effectiveness of our approach, with a focus on the impact of lazy processing on throughput and its scalability.

We construct a set of experiments that first initialize a graph of a certain size and then submit a stream of 100 operations to this graph. The operation stream is randomly generated, consisting of 60% queries, 20% node insertions, and 20% node deletions. When propagation occurs, we move the delayed operation across a number of nodes equal to 0.1% of the graph size, or 1 in the case of the empty initial graph (we call this the *stride* of propagation). To compare the effectiveness of our approach, we construct a baseline that submits the same operation stream to an unmodified Neo4j runtime. From now on, we call this latter setting *eager* processing.

The results are shown in Figure 2 where each subfigure represents a different initial graph size. All results were produced on an AMD Opteron 6378 processor with 32 cores at 1.4 GHz with 66 GB of memory. The Java version used to compile and run Neo4j is Java 8 Update 211 and the Neo4j version used is version 3.5.6. We run each setting for 10 cold trials.

Our preliminary experiments show that lazy processing may significantly outperform eager processing, especially when data size is large. For example, in Figure 2f the result throughput is improved by about 459%, 297%, 245%, 219%, and 188%, respectively, for various input intervals on a graph consisting of 100 million nodes. The relatively stable nature of throughput regardless of input interval — both for lazy processing and eager processing — should not come as a surprise. When the graph size is large, the time for graph traversal becomes the dominating factor for processing operations. As a result, the arrival interval does not play an important role in impacting throughput.

What is perhaps more encouraging is how our approach scales. When data size is small — see Figure 2a and Figure 2b — there is no advantage between the two modes of graph processing. When data size grows — see Figure 2c, Figure 2d, Figure 2e, and Figure 2f — lazy processing starts to show its benefit, and its relative effectiveness appears to be greater when the data size grows. This intuitively makes sense: when the graph size increases the time needed to serve each request also increases; as a result, the time saved from batching becomes evident, as reflected by the throughput improvement.

4. Related Work

We now summarize some closely related work. Sloth (Cheung et al. 2016) is a run-time system based on lazy evaluation. Our in-graph batching design is complementary to theirs, as Sloth dynamically batches database queries at the client–database boundary. DeltaGraph (Dexter et al. 2016) is a Haskell library that delays graph updates in-data, allowing fusion and batching optimizations. They introduce a new form of graph representation based on datatypes whereas our system uses shadow lists to decouple the graph from the run-time support for lazy evaluation.

5. Conclusion

In this paper, we explore the connection between lazy evaluation and database optimization. Our in-graph batching design allows delayed operations to be propagated and batched within the graph. Our shadow list implementation decouples the run-time laziness support from the graph database itself. Our preliminary implementation based on Neo4j shows that this approach has the potential to lead to significant throughput improvement and scales well with data size.

In the future, we would like to expand our experimental space by studying the impact of different combinations of operations (queries, insertions, and deletions), the impact of different batch sizes, and the stride of propagation.

References

Neo4j graph database. `http://www.neo4j.org`, 2010. Accessed: 2019-05-18.

Db-engines ranking. `https://db-engines.com/en/ranking`, 2019. Accessed: 2019-05-18.

Neo4j customers. `https://neo4j.com/customers/`, 2019. Accessed: 2019-05-18.

A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). *ACM Trans. Database Syst.*, June 2016.

P. Dexter, Y. D. Liu, and K. Chiu. Lazy graph processing in haskell. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016. ACM, 2016.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.