Snapshot Isolation for Neo4j

Marta Patiño, Diego Sancho
Universidad Politecnica de Madrid
Madrid, Spain
mpatino@fi.upm.es,
diego.bursan@gmail.com

Ricardo Jiménez-Peris
LeanXcale
Madrid, Spain
rjimenez@leanxcale.com

Iván Brondino, Valerio Vianello, Rohit Damane
Madrid, Spain
Madrid, Spain
{ibrondino, vvianello}@fi.upm.es
rohit.dhamane@gmail.com

ABSTRACT

NoSQL data stores are becoming more and more popular. Graph databases are one of this kind of data stores. In this paper we present an overview of the implementation of snapshot isolation for Neo4j, a very popular graph database.

Keywords

NoSQL, transaction processing, graph databases.

1. INTRODUCTION

Graph databases such as Neo4J [1], Titan [2] and Sparksee [3] are being adopted to represent data that is more naturally captured as a graph than with structured or semi-structured data models such as the relational model, key-value models or document-oriented models. Graph databases also provide either query languages or APIs that enable for traversing graphs, running the whole query on the query engine, therefore, resulting in an efficient traversal of the graph. The use of other data management technology for representing and traversing graphs them becomes very inefficient because it implies executing many iterative queries to extract the adjacent nodes to a given one, what results in a huge number of return trips between the client and database side.

Some of these graph databases provide transactions. This is the case of Neo4J. Neo4J implements the most basic isolation level: read committed. Unfortunately, read committed suffers from many anomalies including unrepeatable reads and phantom reads. Unrepeatable reads allows that a transaction observes different values for a given data item in the same transaction. In the case of graph it means that a path that has been traversed, might not exist when trying to go through it later in the same transaction (e.g. due to a two-step graph algorithm). Phantom reads affect to the selection of items with a predicate. This affects a transaction that performs a predicate selection multiple times, since it might observe a different result set each time, resulting in inconsistent behavior. A higher isolation level avoiding these two anomalies is highly recommended.

Snapshot isolation (SI) [4] is an isolation level that has become very popular since it provides an isolation very close to the one provided by serializability while avoiding read-write conflicts. Snapshot isolation provides a snapshot of the committed state to transactions. Basically, SI splits the atomicity of a transaction in two points. The start of the transaction where logically all reads happen and the commit of the transaction where logically all writes happen. Snapshot isolation only can suffer from an anomaly avoided by serializability know as write skew. The anomaly is not exhibited by all applications, for instance, TPC-C benchmark never observes an anomaly when running on an SI database.

This paper presents how we have designed and implemented a multi-version concurrency control for Neo4J that provides snapshot isolation, avoiding the unrepeatable and phantom reads phenomena that currently affect Neo4J. This work has been performed in the context of the European project CoherentPaaS [5] that provides transactional behavior to NoSQL data stores and global transactions and queries across NoSQL and SQL data stores.

2. Neo4j ARCHITECTURE

Neo4j is a graph database, as such the entities it handles are nodes and relationships (edges in graph jargon) among them. It also allows to define properties and labels. Labels are kind of used to associate a “role” to a node. Properties can be associated to both nodes and relationships.

Neo4j architecture is similar to a traditional database architecture in the overall architecture, although it differs quite a bit in the details (Figure 1). Overall, the architecture has an object cache and a persistent store as a traditional database. However, the internal representation is optimized for graph representation.

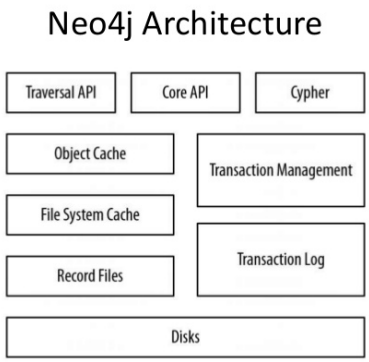

Figure 1: Neo4j Architecture

Nodes are kept in a file whose position is determined by the node identifier. That position in the file contains the ID of the first relationship of the node and the ID of its first property. Relationships are stored in a different file. The source node of the

relationship and the destination node are stored. Properties of nodes and relationships are stored in a different file.

Neo4J also uses indexes to optimize some of the accesses. It has two indexes for nodes, one for labels and another one for properties that map them to the set of nodes associated to them. It also maintains one index for relationships, mapping properties to nodes holding those properties.

3. SNAPSHOT ISOLATION

Snapshot isolation is a multi-version concurrency control. It requires to keep track of multiple versions per entity. This means that updating in place is not feasible and a mechanism is needed to maintain multiple versions of each data item, either physically or logically.

Snapshot isolation can be implemented by enforcing two rules. The read rule states that a transaction should observe the most recent committed version of each data item at the time the transaction started. The write rule states that no two concurrent transactions can update the same data item.

The most common way to enforce the read rule of snapshot isolation is to associate a commit timestamp to versions. The commit timestamp can be seen as a kind of serialization order of a transaction. Snapshot isolation also needs a mechanism to read the right snapshot for a transaction. This mechanism given a start timestamp should enable to observe the most recent committed state that has a commit timestamp equal or lower than the start timestamp.

The write rule of snapshot isolation requires the ability to detect write-write conflicts among concurrent transactions. There are two ways to deal with write-write conflicts, first-updater-wins that rollbacks the transaction that is not the first to update the data item and first-committer-wins that rollbacks the conflicting transaction that does not commit first.

Snapshot isolation also requires a way to remove obsolete versions of the data items (garbage collection), that will never be read by active transactions. This situation happens when there are versions older than the oldest version than the oldest active transaction can read. For instance, if the oldest transaction has start timestamp 100 and a data item has versions with commit timestamps 40, 56 and 90, the first two will never be read by any active transaction.

Another important issue to take into account is that versions of uncommitted data items should be kept private and not accessible to other transactions, but they should read by the transaction that wrote them to guarantee that a transaction reads its own writes.

4. SNAPSHOT ISOLATION FOR Neo4j

We have versioned both nodes and relationships. We have added an additional property to both of them for keeping the commit timestamp. Another property has been added to indicate if a data item has been deleted. A deleted data item has to be kept till no previous version can be read by an active transaction. This mechanism is also called tombstone versions. Versions are kept in the Object Cache of Neo4j. In particular, each object representing a node or relationship stores a list of versions. In that way, when a transaction reads a node, the right version for the reading transaction can be obtained by traversing the list of versions.

Neo4j uses an iterator to traverse the persistent state when needed to answer queries. We have enriched this iterator to take into account the versions kept in the cache in order to guarantee read-your-own writes behavior.

Neo4j implements read committed with a traditional locking mechanism with short read locks and long write locks. We have removed the short read locks since they are not needed for snapshot isolation. The implementation of long write locks has been modified to perform write-write conflicts implementing a first-updater wins strategy.

Multi-versioning has also been applied to indexes. Properties and labels are never deleted in Neo4j even if no node/relationship is using them. We version them to know whether they should be considered or not. When a property or label has been created by a transaction with a higher timestamp than the start timestamp of the reader transaction, it can simply discard them. If the timestamp is equal or lower than the start timestamp of the reading transaction then the list of associated nodes/relationships is traversed. The nodes/relationships are tagged with the commit timestamp of the transaction that associated the label/property to the node/relationship. In this way, it is possible to discard those nodes/relationships that do not correspond to the snapshot to be observed by the transaction (those with a higher commit timestamp than the start timestamp of the reading transaction).

The most difficult question to provide snapshot isolation in Neo4J is how to implement multi-versioning in an efficient way. One of the most common inefficiencies introduced by multi-versioning is the version garbage collection process. For instance, in PosgreSQL this process, called vacuum process, stops the processing for a few seconds periodically. This happens because it traverses all the pages in the persistent storage and rewrites them after removing the obsolete versions.

The approach we have adopted avoids this issue by only writing to the persistent data store the most recent committed version of each data item. The other versions are kept in memory. In order to make the version garbage collection efficient, they are threaded with a double linked list sorted by timestamp to enable to perform the garbage collection just traversing those versions that must be garbage collected. In this way, the cost of garbage collection is reduced to the minimum.

5. ACKNOWLEDGMENTS

This research has been partially funded by the European Commission under project CoherentPaaS (grant agreement FP7-611068), the Madrid Regional Council (CAM), FSE and FEDER under project Cloud4BigData (grant S2013TIC-2894), and the Spanish Research Agency MICIN under project BigDataPaaS (grant TIN2013-46883).

6. REFERENCES

[1] Ian Robinson, Jim Webber, and Emil Eifrém. Graph Databases Publisher: O'Reilly Media. 2015.

[2] http://thinkaurelius.github.io/titan/

[3] http://sparsity-technologies.com/

[4] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. ACM SIGMOD Conference 1995. pp 1-10.

[5] http://coherentpaas.eu