Poster Paper

Snapshot Isolation for Neo4j[1]

Marta Patiño-Martínez, Diego Burgos-Sancho
Universidad Politécnica de Madrid
Madrid, Spain
{mpatino,diego.burgos}@fi.upm.es

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

Iván Brondino, Valerio Vianello, Rohit Dhamane
Universidad Politécnica de Madrid
Madrid, Spain
{ibrondino, vvianello,rdhamane}@fi.upm.es

ABSTRACT

NoSQL data stores are becoming more and more popular. Graph databases are one of this kind of data stores. Neo4j is a very popular graph database. In Neo4j all operations that access a graph must be performed in a transaction. Transactions in Neo4j use read-committed isolation level. Higher isolation levels are not available. In this paper we present an overview of the implementation of snapshot isolation (SI) for Neo4j. SI provides stronger guarantees that read-committed and provides more concurrency than serializability.

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 or key-value models. Graph databases also provide either query languages or APIs that enable for traversing graphs, running the whole query on the data store, 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 overhead.

Some of these graph databases provide transactions, like Neo4j. Neo4j implements a 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 graphs, it means that a path that has been traversed might not exist when trying to go through it later in the same transaction. 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. 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. SI only can suffer from an anomaly avoided by serializability known as write skew. The anomaly is not exhibited by all applications, for instance, the 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

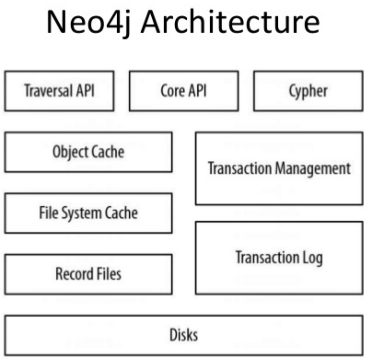

Figure 1: Neo4j Architecture

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

[1] This research has been partially funded by the European Commission under projects CoherentPaaS and LeanBigData (grants FP7-611068, FP7-619606), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC-2894), and the Spanish Research Agency MICIN project BigDataPaaS (grant TIN2013-46883).

10.5441/002/edbt.2016.94

Neo4j architecture is similar to the one of a traditional database, 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 graphs.

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 typically implemented as a multi-version concurrency control (MVCC). It requires keeping track of multiple versions per entity. This means that updating in place is not especially convenient and a mechanism is needed to maintain multiple versions of each data item, either physically or logically.

SI 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 start. The write rule states that no two concurrent transactions can update the same data item.

SI requires a way to remove obsolete versions of the data items that will never be read by active transactions. Another important issue to take into account is that versions of uncommitted data items should be kept private, but they should read by the transaction that wrote them to guarantee that a transaction reads its own writes.

4. SNAPSHOT ISOLATION FOR Neo4j

Transactions are assigned a start timestamp that corresponds to the most recent committed state. The commit timestamp is given to a transaction when it commits. This commit timestamp is used to tag each item (version) the transaction has updated. We have versioned both nodes and relationships. Versions are implemented as an additional property for both of them. This property stores 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 isolation 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 visible or not for a particular transaction. 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. 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.

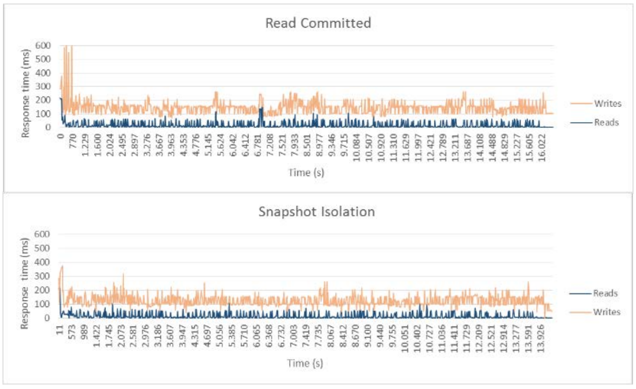

Figure 2: Performance Results

We have performed a preliminary performance evaluation comparing the original implementation of transactions in Neo4j with our SI implementation. The database has 12.3MB and stores movies (from http://neo4j.com/ developer/example-data/). The workload executes 50% updates and 50% reads. Read transactions read a random node. Update transactions read a random node and modify a random property of the node. In Fig. 2 the results of the micro-benchmark are shown. The response time is similar for both implementations, updates last 100-200 ms and reads below 50 ms.

5. REFERENCES

[1] I. Robinson, et al. Graph Databases. O'Reilly Media. 2015.

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

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

[4] H. Berenson, et al. A Critique of ANSI SQL Isolation Levels. SIGMOD 1995.

[5] http://coherentpaas.eu