CEUR-WS.org/Vol-2456/paper90.pdf

Transactional Guarantees for SPARQL Query Execution with Amazon Neptune

Brad Bebee[1], Ankesh Khandelwal[1], Sainath Mallidi[1], Bruce McGaughy[1], Simone Rondelli[1], Michael Schmidt[1], and Bryan Thompson[1]

[1] Amazon Web Services, Seattle, WA 98101, USA

Abstract. Amazon Neptune is a fast, reliable, and fully managed graph database service, designed to efficiently store and query highly connected data. Neptune supports both the Apache Tinkerpop Gremlin stack as well as the RDF/SPARQL W3C standards. Designed to support highly concurrent OLTP workloads over data graphs, ACID support is a key feature of Neptune. While the W3C SPARQL standard deliberately does not define transaction semantics for SPARQL query processing, we learned from our customers that transactional guarantees are crucial in the context of concurrent read-write workloads against the data graph. In this presentation, we sketch use cases in which transactional guarantees are essential to avoid data anomalies, highlight aspects in which graph data transaction requirements differ from transactional systems in classical relational databases, sketch the transactional guarantees provided by Amazon Neptune, and discuss implementation aspects of Neptune's transaction system.

Keywords: RDF, SPARQL, OLTP, Transactions, Amazon Neptune

1 Transactional Guarantees for SPARQL

The SPARQL standard, unlike SQL, does not define any transaction semantics. What we learned in discussions with our customers, however, is that transactional guarantees (and, evenly important, knowledge about which guarantees do hold) are crucial to build reliable applications on top of graph databases. While transactional guarantees for triple stores are not new, in this presentation we discuss the challenges, experiences, and conceptual issues encountered when designing Neptune's transaction system.

A typical use case we're seeing for RDF is data integration, where data is assembled from different sources. Data extractors may run and update the same entity in parallel. As a motivating example, assume we have two triples describing an entity `:Person1` with `rdf:type :Person` and `:name "John Doe"`. Now assume two concurrent transactions, one adding a triple to the person, the other deleting all its triples:

```
tx1: INSERT { :Person1 :age 23 }      tx2: DELETE { :Person1 ?p ?o }
```

Depending on the execution order and transactional guarantees, different outcomes are possible: if `tx2` runs strictly after `tx1`, it may see the change made by `tx1` and delete all triples for the person; otherwise, we may end up with a "dangling" triple `:Person1 :age 23`, where dangling refers to the fact that the entity is somewhat incomplete now (in particular, untyped). This may harm the correctness of applications that do (want to) rely on the fact that all subjects in the database are typed.

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

What makes this example different from transactions in the relational world, where it would naturally translate into a conflicting row update vs. delete over a `Person` table, is that RDF decomposes data into triples. Most notably, when executed individually, `tx1` and `tx2` operate over *totally disjoint* sets of triples. Hence, no lock *over a single triple* would ever prevent the transactions to perform concurrent modifications – yet there is a logical connection in the sense that both transactions modify the same entity. Applications built on top of RDF tend to have such an "entity centric" view of the data, so there are often strong requirements to lock the entity as such.

We see use cases for transactional guarantees across all industries, in particular in scenarios where RDF is used to integrate data from different source systems. Examples include ontology based product catalogs that are periodically aggregated from different sources, IT-level assets aggregated from different inventory and monitoring systems, or the challenge of capturing change in dynamic social network applications.

Neptune isolation levels. Neptune supports snapshot isolation for read-only queries and read-committed-without-phantoms isolation for update queries. Reads for both read-only and update queries avoid all three phenomena of dirty, nonrepeatable, and phantom read. Nonrepeatable read is satisfied because triples cannot be updated, they have to be deleted and newly inserted. Read-only queries never see changes from concurrent transactions. Update queries see committed changes, yet locking (cf. next paragraph) prevents inserts and deletes into index ranges after they have been read, thus guaranteeing repeatable read. Hence, whenever an update query sees a change from a concurrent query, it is as if it started executing after the latter completed.

SPARQL update queries have an implicit or explicit WHERE clause that is executed prior to its INSERT or DELETE part. Evaluating this WHERE clause takes shared locks on all the triples visited during the evaluation, to prevent any deletions. In order to facilitate entity level locking, we employ so-called prefix locks. For example, while evaluating the (implicit) WHERE clause for `tx2`, the entire prefix spanned by `:Person1` is locked, preventing *any* triples with this subject being inserted or deleted by concurrent transaction. If Neptune encounters any uncommitted matching triple from a concurrently executing query, the evaluation of the WHERE clause is blocked until that query completes. This helps detecting conflicts between reads and writes. We wait a few seconds for conflicts to resolve or else rollback the query.

Based on these guarantees, we can safely rewrite our example by changing `tx1` to:

```
tx1: INSERT { :Person1 :age 23 } WHERE { :Person1 rdf:type :Person }
```

A dangling triple could now no longer arise: if the transactions run concurrently, `tx2` does lock the *prefix range* for `:Person1`, preventing `tx1` from concurrent insertion; if `tx1` runs after `tx2`, it would no longer see the `rdf:type` triple and become a no-op.

In the presentation we will discuss customer use cases that require strong transactional guarantees, describe Neptune's transaction system, and sketch recipes for building applications that leverage its transaction guarantees to overcome the conceptual gap between a triple-based data model and an entity centric view over the data.