Enabling an Enterprise Data Management Ecosystem using Change Data Capture with Amazon Neptune

Brad Bebee[1], Rahul Chander[1], Ankit Gupta[1], Ankesh Khandelwal[1], Sainath Mallidi[1], Michael Schmidt[1], Ronak Sharda[1], Bryan Thompson[1], and Prashant Upadhyay[1]

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

Abstract. Given their flexibility in interlinking heterogenous data, graph databases are often used as a central hub within the enterprise data management ecosystem. While the data graph as such can be queried as an integrated data corpus using existing graph query languages (in Amazon Neptune, we support both SPARQL as a query language over RDF as well as Gremlin over the property graph data model), one key requirement of our customers is to integrate the data graph with external, purpose-built data management systems. In this demonstration, we will present Amazon Neptune's approach to synchronize graph data to external systems using Neptune's Change Data Capture (CDC) mechanism. We discuss the design and properties of the CDC feature and show how it can be used to synchronize graph data to external systems. Exemplified by a movie graph database which is periodically updated with new movies, we will showcase a fault-tolerant cloud architecture leveraging CDC to periodically propagate updates made to the graph database into a backing Elasticsearch search index, in order to provide efficient full-text search over the graph data. On top of this stack, we demonstrate Neptune's approach to integrated querying across the data graph and the keyword search cluster.

Keywords: CDC, Full-text Search, Graph Databases, RDF, SPARQL, Amazon Neptune, Apache TinkerPop

1 Change Data Capture in Amazon Neptune

While Relational Database systems have been dominating the database market for many decades, in recent years we have witnessed an ongoing diversification. Addressing the broad range of today's data processing use cases, the Amazon Web Service (AWS) ecosystem follows a purpose-built data management paradigm, offering a variety of services tailored to and optimized for the specific needs of our customers' use cases. Beyond the classical relational data management systems offered by Amazon RDS, AWS services include Amazon Redshift for analytics, Amazon DynamoDB as a NoSQL key-value store, Amazon DocumentDB for managing JSON documents, Amazon ElastiCache for optimized in-memory processing, Amazon Timestream as a solution for time series data, as well as Amazon Neptune, a fully managed graph database service supporting both the W3C's RDF/SPARQL [1] and Apache Tinkerpop's Property Graph/Gremlin [2] stack. These core database services are complemented by other purpose-built data processing systems such as the Amazon Elasticsearch Service providing efficient full-text search functionality, large-scale data analytics services like Amazon Elastic MapReduce (EMR), and services such as Amazon SageMaker supporting machine learning workflows and algorithms on top of the data.

Within this growing universe of data management and processing approaches, graph databases often take a distinguished role: in particular in the context of RDF, which has its strengths in data integration from different sources, across different domains, we commonly see our customers use graph database as a central hub in their data ecosystem. In such scenarios, it becomes crucial to provide mechanisms that make it easy to synchronize graph data (or subsets thereof) with other systems.

Use Cases. In discussions with our customers, we have learned about use cases for integrating graph with virtually each of the data management and analytics system in the AWS ecosystem, and beyond. A customer in the financial services industry would like to provide configurable subscriptions triggered by changes in RDF triple patterns. Other customers want to synchronize a secondary RDF (or property graph) store based on the primary Amazon Neptune instance, update caches such as ElastiCache to provide high-performance access to frequently requested portions of the data, dump graph data changes to S3 to facilitate batch processing of graph data via EMR using MapReduce or Apache Spark, or keep datasets analyzed in SageMaker using machine learning toolchains synchronized with their graph data. Use cases for CDC beyond external system synchronization include triggers based on detected updates to entities, e.g. rule re-computation or the re-validation of SHACL [3] constraints after resource updates.

Feature Description. Addressing these use cases, we have developed a graph Change Data Capture (CDC) [4,5,6,7] mechanism for Amazon Neptune, which exposes changes made to the data via a change log API. The change log is represented as a sequence of deltas (either addition to or removal from the data graph), where entries have a unique, monotonically increasing event id. Access to the changes is paginated, i.e. readers provide the desired starting event id and the max number of changes, and get a sequence of changes (wrapped into a JSON document) back as a response.

When using Neptune over RDF data, changes are reported in N-Quads format [8]; over Property Graphs, they are reported at a higher abstraction level, using an edge/vertex centric view. While, from a logical perspective, the information content is identical in both cases, the different abstraction levels adhere to the abstraction that the users of the two stacks are familiar with and facilitates the reuse of client-side tooling for the two stacks, respectively (e.g., the reuse of RDF libraries for parsing change sets).

CDC in Neptune has been implemented as an opt-in feature. The change logs can be fetched using a REST endpoint from either the master instance or any read replica in the cluster. This makes it possible to spin up a read replica dedicated for CDC access, thus querying change logs without interfering with operational workloads. To keep storage cost contained, change logs older than seven days are automatically purged.

Neptune's CDC mechanism is tightly integrated with its transaction system, i.e. the change log is maintained as an integral part of each transaction. This makes it possible to derive strong guarantees regarding representation and timeliness of changes:

- The change log is **complete and correct**. There is **no look-ahead required**, i.e. a change written to the log would never be invalidated by a future log entry.

- Changes are **free of redundancy**: for instance, if a SPARQL query attempts to delete the same quad twice, the change log will show the deletion only once.
- Changes are **ordered by their effective commit order**: this is crucial to guarantee that a sequential replay of the change log (e.g. against another RDF store) will result in exactly the same database state.
- Changes are **visible immediately**: once a transaction has been committed, a change API request against the master is guaranteed to return that change.

Connecting External Systems. The previously mentioned properties simplify the implementation of clients consuming the changes: all clients need to do is periodically fetch the latest changes and replicate them to the synchronization target in sequence, without caring about re-ordering, redundancy, and aspects like deferred invalidation.

Making it easy to consume and propagate changes shifts the challenge of building replication clients towards the operational side, with the key requirement to establish a replication architecture that is tolerant towards temporary outages (e.g. coping with system upgrades, and temporary fail-over) of the source and target system(s).

In order to ease the setup of such a fault-tolerant replication system, we provide configurable templates that can be automatically deployed via the CloudFormation automation service (see Fig. 1). By providing just a few instance specific configuration parameters (such as application name, the source database endpoint providing the CDC API, network configuration, the replication target system endpoint, polling frequency, etc.), a fault-tolerant replication stack can be deployed automatically within minutes.

At its core, the replication stack uses serverless AWS Lambda functions, which are periodically triggered via a CloudWatch event rule (obeying the specified polling frequency). When invoked, the Lambda functions send a request to Neptune's CDC API to extract new changes and propagates them to external systems (in the case of our demo, Amazon Elasticsearch – but users can implement their own adapters). System state (such as the last processed change event id) is maintained in a DynamoDB table. To ease operation and debugging, the stack comes with the support of application log archival (Amazon Cloudwatch), trace logs (AWS Xray), and metrics and alarms (number of change logs processed, approximate replication lag in Amazon Cloudwatch).

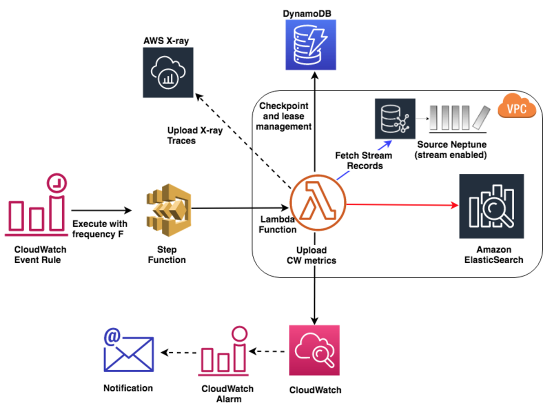

Fig. 1. Blueprint cloud architecture for consuming Neptune's CDC stream. Solid lines represent mandatory, dashed lines optional components.

Demo. In the demonstration, we will showcase Amazon Neptune's CDC system in action. As a showcase, we will use a movie graph data set that captures relationship between real movies, actresses/actors and other persons of interest, locations, etc. Going beyond plain text indexing (names, titles, descriptions, etc.), we also leverage the geospatial indexing capabilities of Elasticsearch to store and index geo coordinates. Using this scenario as a showcase, in the demo we will:

1. Discuss and demonstrate Neptune's CDC mechanism for synchronization between graph and keyword search systems. In particular, we will discuss possible mappings between the graph data and document-centric systems.
2. Showcase the creation and operation of a fully functional cloud stack that periodically replicates data from Neptune to Elasticsearch, in a scenario where we continuously update the graph database with new movies.
3. Execute both stand-alone and hybrid queries over data graphs and the synchronized search cluster, demonstrating integration benefits by means of sample requests that combine structural data aspects (provided by Neptune) with efficient ranked/fuzzy keyword search, as well as geospatial queries (provided by Elasticsearch). Integrated querying is made possible by integrating Elasticsearch via a dedicated SPARQL SERVICE clause that reaches out to Elasticsearch during regular SPARQL query evaluation.
4. Discuss lessons learnt during the design and implementation of CDC for graph database, including problematic cases (such as conceptual issues regarding blank node scope and semantics when synchronizing multiple RDF databases via CDC, or transactional guarantees while replaying CDC streams).

References

1. SPARQL Query Language for RDF. W3C Recommendation 15 January 2008. grrp2(5), 99–110 (2016). https://www.w3.org/TR/rdf-sparql-query/, last accessed 2019/06/26.
2. Apache Tinkerpop: a graph computing framework for both graph databases (OLTP) and graph analytic systems (OLAP). http://tinkerpop.apache.org/, last accessed 2019/06/26.
3. Shapes Constraint Language (SHACL). W3C Recommendation, July 20, 2017. https://www.w3.org/TR/shacl/, last accessed 2019/06/26.
4. Andy Seaborne, Ian Davis: Supporting Change Propagation in RDF. https://www.w3.org/2009/12/rdf-ws/papers/ws07, last accessed 2019/06/26.
5. Tummarello, G., Morbidoni, C., Bachmann-Gmür, R. and Erling, O., 2007. RDFSync: efficient remote synchronization of RDF models. In The Semantic Web (pp. 537-551). Springer, Berlin, Heidelberg.
6. Passant, A. and Mendes, P.N., 2010, May. sparqlPuSH: Proactive Notification of Data Updates in RDF Stores Using PubSubHubbub. In SFSW.
7. Bao, J., Ding, L. and McGuinness, D.L., 2009. Semantic history: Towards modeling and publishing changes of online semantic data. *The 2nd Social Data on the Web*.
8. RDF 1.1 N-Quads. W3C Recommendation 25 February 2014. https://www.w3.org/TR/n-quads/, last accessed 2019/06/26.