DOI:https://doi.org/10.1145/2678508.2678

Augmented Inverted Indexes to Track Causality in Eventually Consistent Data Stores

Christopher Meiklejohn
Basho Technologies
cmeiklejohn@basho.com

ABSTRACT

We propose a novel algorithm for providing highly-available and fault-tolerant distributed search queries in a Dynamo-inspired distributed data store. It leverages the use of advanced causality tracking mechanisms to allow quorum reads of an inverted index, while maintaining the ability to reason about the correct response. This algorithm improves on existing distributed search mechanisms by providing higher availability in the event of failures, given the use of multiple replicas.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Design, Reliability

Keywords

Information Retrieval, Riak, Solr

1. INTRODUCTION

In replicated, eventually consistent, Dynamo-inspired [4] data stores, quorums are used to perform read and write operations to provide high availability in the face of failures. Because of this, the values of each object can diverge when writes do not complete against all replicas. Quorum read operations use causality tracking mechanisms such as vector clocks and version vectors [6] to reason about which replicas contain the most up-to-date data.

To resolve this divergence, these systems usually use two anti-entropy mechanisms: an asynchronous process that repairs data by iterating the entire dataset between pairs of replicas, and a mechanism for repairing missing replicas as part of the read operation lifecycle.

We examined two commercial, open-source offerings of the Dynamo-model for search capabilities: Riak and Cassandra. Both provide an integration, maintained either by the vendor or the product's community, of the Lucene-based full text search engine, Solr. Indexes are sharded across the key space and computed at each replica as data is being written by the data store. As replicas are being repaired for divergences, we also update the indexes to the reflect the current state of the data stored. This type of index partitioning is commonly referred to as document-based partitioning [3], given the indexes are sharded and located along with the replica from which the index was derived.

To provide high availability and fault tolerance for distributed search queries, as we do for coordinated quorum reads in Riak, it is highly desirable to perform these operations against a quorum of indexes and compute the final result from received responses. The main contributions of this paper discuss an algorithm and prototype implementation of a novel indexing strategy, based on the use of DVV sets (dvvs) [1] to provide causality tracking. This causality tracking mechanism allows us to reason about the most recent record to return as the result of the distributed search query.

2. BACKGROUND

The following section describes background information about how Solr's two distributed search offerings operate and Riak's integration with Distributed Search, Yokozuna.

2.1 Distributed Search

Distributed Search provides the ability in Solr versions prior to 4.0, to perform a search query over a series of shards. This mechanism provides only the distribution of the search query, but not the sharding of the index, which must be done manually. It assumes that the shards represent disjoint sets of data; in the event that the same value appears in two indexes, the search query may be non-deterministic. In the event of failure, results are absent from the result set. [5]

2.2 SolrCloud

SolrCloud provides the distributed search functionality in Solr 4.0. SolrCloud automatically shards the data across multiple nodes and allows for replication of each shard to provide fault tolerance. Zookeeper is used to coordinate writes to shards and their replicas.

When performing searches, queries are sent to all shards of a given index. In the event that one of the shards is unavailable, the entire query is failed. SolrCloud also provides the option to be tolerant to the failure; in this case, the query will not be failed, but the results from the missing shard will not be returned with the result set. [7]

2.3 Yokozuna

Yokozuna is the code name for the integration of Distributed Search with the Riak 2.0 data store. This integration ties Solr's indexing into the write lifecycle in Riak; as data is written to partitions in the Riak data store, those objects are indexed by Solr. Distributed Search's existing query mechanism is used when a user performs a search, however, Riak provides a disjoint set of replicas representing the entire key space. This query mechanism is susceptible to the same problems as discussed in 2.1. [1]

3. IMPLEMENTATION

In this section we outline how the concepts from dvvs are used for tracking and resolving changes when performing distributed search queries over multiple eventually consistent replicas of the same index.

We begin by extending the inverted index to store a dvvs associated with the object when the object is indexed. This dvvs represents the causal history of the object, along with a unique identifier that monotonically advances at the coordinating replica, establishing its write in the causal history.

3.1 Update Algorithm

The following section outlines the algorithm for updating the indexes on each write operation at a local replica.

For every index, i_n, perform the following:

- For every key k_n, store the version vector for the key, along with an empty set on an initial write, known as the *dots*.
- For every write of k_n, update the version vector associated with the key with the current version vector of the object.
  - If the value should be indexed; insert into the dot set. For instance, for version vector $[\{a, 1\}]$ store $[\{a, 1\}]$ in the *dots*.
  - If the value should not be indexed, remove all *dots*.

3.2 Resolution Algorithm

We now examine how we go about resolving the index and determining which values we should return from the search query.

For each key in the index, perform the following resolution logic.

Consider the following:

Replica a	Replica b
$[\{a,1\}]\ [\{a,1\}]$	$[\{a,1\},\{b,2\}]\ [\{b,2\}]$

In this case, we have indexes located on disjoint replicas. The results returned at replica a show that replica b dominates it, therefore the index shows that the value of key k_n should be included in the results.

Consider this case, where removals are observed on only one replica:

Replica a	Replica b
$[\{a,1\}][]$	$[\{a,1\}][\{a,1\}]$

When examining this case the replica containing the empty set has causally observed the addition and removal of the value from the index. In this case, or its inverse, we determine the index should omit the value of k_n from the results.

Consider the case of a stale replica:

Replica a	Replica b
$[\{a,1\}][\{a,1\}]$	$[][]$

In this case, replica b has not observed the write yet, so we resolve by inclusion of k_n in the results.

Finally, in the case of concurrent writers, which will generate a sibling, or concurrent, value in the data store:

Replica a	Replica b
$[\{a,1\}][\{a,1\}]$	$[\{b,1\}][\{b,1\}]$

In this case, k_n has to be included in the index, given both values concurrently have been entered in the index.

3.3 Future Work

In summary, we have applied the concepts of dvvs to distributed indexes, allowing for correct resolution of the index in the event of partial failures and concurrent writes. We hope to further extend this work to systems which exhibit similar problems, but are not Dynamo-inspired eventually consistent data stores, such as Elasticsearch. [2] We plan to continue to research and improve this algorithm, ensuring correct operation in the event of failures.

[1]Conceptually, this is a disjoint set. However, partitions are repeated but supplied with a filter for non-overlapping objects which produce disjoint sets. This is due to a minor implementation detail in Riak where the same virtual node coordinates writes for n replica sets.

4. REFERENCES

[1] P. S. Almeida, C. Baquero, R. Gonçalves, N. Preguiça, and V. Fonte. Scalable and accurate causality tracking for eventually consistent stores. In *Distributed Applications and Interoperable Systems*, pages 67–81. Springer Berlin Heidelberg, 2014.

[2] S. Banon. Elasticsearch. `http://www.elasticsearch.org/overview/elasticsearch/`, Sept. 2014.

[3] B. B. Cambazoglu, A. Catal, and C. Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In *Proceedings of the 21st International Conference on Computer and Information Sciences*, ISCIS'06, pages 717–725, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[5] E. Erickson. What is distributed search? `https://wiki.apache.org/solr/DistributedSearch`, Sept. 2014.

[6] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing*, 7(3):149–174, 1994.

[7] C. Targett. Read and write side fault tolerance. `https://cwiki.apache.org/confluence/display/solr/Read+and+Write+Side+Fault+Tolerance`, Sept. 2014.