Evaluating Performance of Anti-Entropy Algorithms

Kenny Gao
Sriram Mohan, Advisor
Rose-Hulman Institute of Technology

May 18, 2012

Abstract

Distributed databases circumvent numerous performance issues presented by traditional databases, but introduce inherent limitations of their own - Brewer's CAP theorem states that a distributed system cannot guarantee all three properties of consistency, availability, and partition tolerance at the same time. This typically results in consistency being scaled back to the weaker eventual consistency. In order to enforce eventual consistency in a system containing replicated data, distributed databases like Amazon's Dynamo and Apache Cassandra make use of a process called anti-entropy, implemented using Merkle trees. Here, I evaluate the performance of Merkle tree-based anti-entropy and look at ways in which it can be improved.

Contents

1 Introduction

As software makes its transition from the desktop to the web, distributed databases have emerged as a popular solution to handle the issues presented by the need to provide functionality at scale. The high availability and tolerance for network partitions that they provide, however, comes with the cost of weakened consistency, as shown to be necessary by Brewer's CAP theorem[1]. In order to accommodate this limitation, distributed databases implement a suite of consistency-providing techniques. One such technique is called anti-entropy, and is implemented via the use of Merkle trees[5].

In this paper, we'll begin by elaborating upon the context behind Merkle tree-based anti-entropy in greater detail. From there, we'll delve into the motivation behind evaluating the performance of anti-entropy algorithms. Then, we'll establish what performance is and what algorithms to evaluate, describe the approach to and the implementation of the algorithm simulation framework, and review the resultant data. Finally, we'll extrapolate some implications from the results and discuss avenues for further work on this topic.

2 Background

2.1 Distributed Databases

A distributed database is a database in which data is held in multiple physical locations. Examples include Google's BigTable[2], Amazon's Dynamo[3], and Apache Cassandra[4].

2.2 The CAP Theorem

Eric Brewer's CAP theorem states that a distributed database cannot guarantee all three properties of consistency, availability, and partition tolerance at the same time.

2.3 Eventual Consistency

As a result, modern distributed databases are designed to guarantee only availability and partition tolerance. Consistency, on the other hand, is replaced with the weaker eventual consistency[6], which allows data to be inconsistent across replicas with the understanding that conflicts will eventually be resolved.

2.4 The Need for Anti-Entropy

Techniques like read-repair, write-repair, and hinted handoff cover most conflict resolution cases - particularly, those caused by minor replica failures. To resolve prolonged replica failures, distributed databases rely on a technique called anti-entropy. When a replica that has experienced prolonged failure rejoins the system, anti-entropy synchronizes its data with that of another replica, resolving the conflicts between the two. Without it, we would be forced to choose between slowing down I/O (read/write-repair), re-instancing the replica, or using another asynchronous conflict resolution method.

2.5 Merkle Trees

Distributed databases such as Dynamo and Cassandra implement anti-entropy using Merkle trees, or hash trees. Merkle trees are trees of hashes in which parent nodes hold the hash of the concatenation of the hashes of their children, and the leaf nodes are hashes of some array of data.

3 Motivation

Although there was a reasonable amount of literature describing the role and function of Merkle trees in anti-entropy schemes, there was a distinct lack of justification for using them. With even a basic understanding of how they function, it cannot be denied that Merkle trees perform their duty adequately, but as I read more and more, I wondered: *Why are they the* *best choice?* To answer that question, I first needed to answer the more immediate one of *Exactly how "good" are Merkle trees?*

Of course, both questions are vague and unanswerable without first establishing some definitions and context. The key concept that needs to be disambiguated is that of performance. What exactly does "good" mean?

4 What Performance is

There are a number of ways for us to define performance, and each have merits. A sample of them are listed below.

- **Speed (time)** - At first glance, this is the option that stands out as the most obvious. Ultimately, speed is king, but it's also difficult to measure reliably. Speed is the final result of countless variables that factor into performance, including both those which stem from theory and those which stem from implementation. Existing literature describing the implementation of Merkle trees in anti-entropy schemes reveal heavy usage of caching, lazy tree generation, and amortization of initial costs, and while those techniques are important, they reveal little of the intrinsic performance of Merkle trees.
- **Network usage** - An argument can be made that the real cost of anti-entropy is not experienced by the replicas themselves, but by the connections between them. Network usage is also largely unaffected by the implementation-level optimizations mentioned previously, as they operate exclusively within the replicas and have no bearing on the communication between replicas. As an additional bonus, network usage is fairly straightforward to measure via simulation.
- **Qualitative criteria** - Undeniably, there are both advantages and disadvantages to using Merkle trees that can't be quantitatively measured. One such advantage, mentioned above, is its conduciveness to optimization at the implementation level with techniques such as caching, lazy tree generation, and amortization of initial costs. Not all techniques can be optimized in this way, and as such, it may be serve as a major differentiating factor between it and other anti-entropy tech-

niques. Unfortunately, “ease of optimization” is not something that can be quantitatively measured.

After careful consideration, I chose network usage as the main criterion of evaluation. Its ease of measurement via simulation and its representation of a key component of anti-entropy performance made my goal of evaluating said performance both within reach and meaningful.

5 What to Evaluate

Having established network usage as the measurement criterion, we need to decide what we will be applying measurement to. That is, what algorithms are we interested in comparing the standard Merkle tree algorithm to?

A comparison study like this one requires adequate context in the form of a baseline to compare everything to. That is, we require, at the minimum, a base case algorithm to compare all other algorithms to. In the context of anti-entropy, that base case algorithm will be the one in which *Replica A* sends its entire dataset to *Replica B*, and *Replica B* informs *Replica A* of the values that require updating.

It may also be worth looking at variations on the standard Merkle tree algorithm. Specifically, we can fill in the gamut between the base case algorithm and the standard Merkle tree algorithm with algorithms that construct partial trees of varying depth. Additionally, we can look in orthogonal directions at ways to change components of the standard Merkle tree algorithm to optimize for certain inputs or cases.

To put all of these ideas into concrete examples, we could look at the following set of algorithms:

- Pairwise data comparison (naive)
- Naive algorithm with single hash of all data
- Naive algorithm with multiple layers of hashing
- Standard Merkle tree algorithm (full hash tree)
- Variations: non-binary trees, short-circuiting with base case failover

6 Approach

One of the key benefits of choosing network usage as the criterion of performance is its ease of measurement via simulation. We take full advantage of that fact in planning our approach.

Given the following variables:

- # of data items stored on replicas
- # of conflicts between replicas

Measure each algorithm's performance in terms of network usage.

Repeatedly simulate on randomly-generated data for each configuration of variables.

7 Implementation

The simulation framework is written in Python 2.7 and runs 100 independent simulations, for each configuration of variables, for both the naive algorithm and the standard Merkle tree algorithm, using randomly generated data.

The number of data items stored on replicas (*datacount*) was set to 16 and 32. Setting *datacount* to 64 (the simulation framework assumed that *datacount* was a power of two) led to prohibitively-long simulation times.

The number of conflicts between replicas (*conflictcount*) was set to values from 1 to *datacount*, inclusive.

8 Results

Interestingly, we see that while the standard Merkle tree algorithm outperforms the naive algorithm when *conflictcount* is low, the reverse is true when *conflictcount* is high. The break-even point for *conflictcount* is at only around 22% of *datacount*. We see this behavior for both *conflictcount* $= 16$ and *conflictcount* $= 32$.

Figure 1: network usage for $datacount = 16$

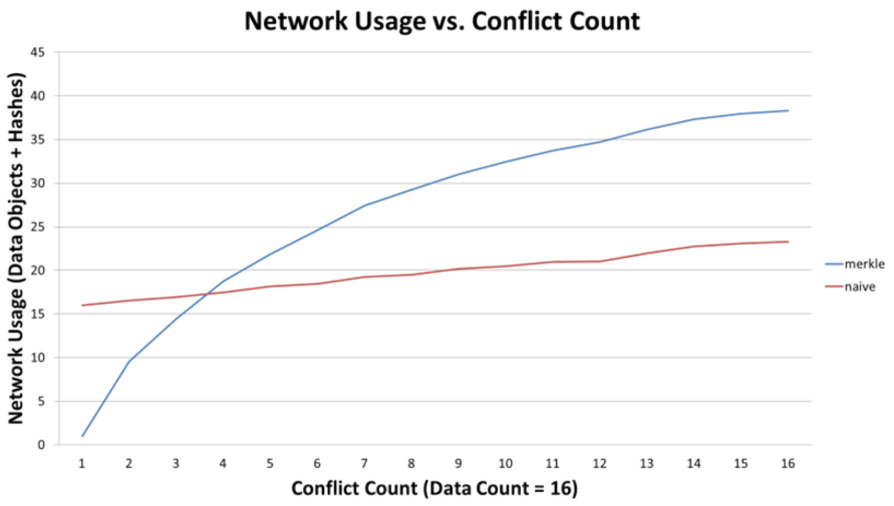
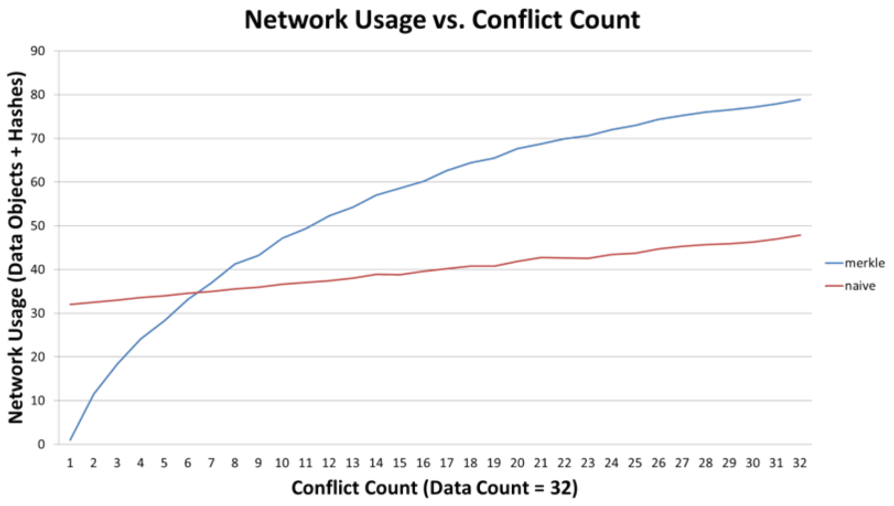


Figure 2: network usage for $datacount = 32$



In addition, we notice that the naive algorithm grows linearly with respect to *conflictcount* (predictably, given the nature of the algorithm), whereas the standard Merkle tree algorithm grows approximately logarithmically. Again, we see this behavior for both $conflictcount = 16$ and $conflictcount = 32$.

Due to computational limitations, we must assume that these results do scale with respect to *conflictcount*.

9 Implications

The most immediate implication is that the standard Merkle tree algorithm is indeed not the "best" in terms of network usage in all circumstances. In fact, it loses to the naive base case algorithm for $conflictcount > 22\%$ of *datacount*! We now have evidence that different algorithms may be "best" at different points on the *conflictcount* spectrum.

From the above, it follows that current anti-entropy methods could be improved by estimating the *conflictcount* of replica pairs. Given the performance profiles of a set of algorithms with respect to *conflictcount* and an estimated value for *conflictcount*, we can always pick the algorithm with the best performance for that particular value.

It should be reiterated, however, that these results are, by design, only concerned with network usage. Other criteria for performance may be equally or more important in various circumstances, and are not represented by these results.

It's also worth noting that we only considered *conflictcount* as our independent variable; other properties such as the spacial locality of the conflicts may also factor into performance.

10 Further Work

10.1 Implement remaining algorithms

Implementing and simulating the remaining algorithms in the list presented earlier would serve as a very reasonable next step. Perhaps we'd find that the standard Merkle tree algorithm is not the "best" in terms of network usage for any value of *conflictcount*. Or, perhaps we'll discover the opposite, giving credence to the widespread usage of the algorithm.

10.2 Extend performance evaluation

Also worth exploring are ways to evaluate performance other than network usage. We discussed some of these alternatives earlier, but it was certainly not an exhaustive discussion. Establishing multiple criteria for performance would yield a more nuanced perspective on how "good" the algorithms are in relation to one another.

10.3 Investigate other variables that might impact performance

We only considered *conflictcount* as an independent variable, but it's certainly not the only valid choice. Intuitively, the structure of Merkle trees would favor datasets in which conflicts were grouped together and wound up being represented in the same Merkle subtree. Certainly, the algorithm could be enhanced to take advantage of such cases. Thus, the spacial locality of the conflicts between datasets could be an important variable to consider when evaluating anti-entropy algorithm performance.

References

[1] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[4] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[5] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

[6] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.