# Nye's Trie and Floret Estimators: Techniques for Detecting and Repairing Divergence in the SCADS Distributed Storage Toolkit

*Jesse Trutna*
*David A. Patterson, Ed.*
*Armando Fox, Ed.*

Electrical Engineering and Computer Sciences
University of California at Berkeley



Technical Report No. UCB/EECS-2010-30
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-30.html

March 18, 2010

Copyright © 2010, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

# Nye's Trie and Floret Estimators: Techniques for Detecting and Repairing Divergence in the SCADS Distributed Storage Toolkit

by Jesse Trutna

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

**Committee:**

Professor David Patterson
Research Advisor

(Date)

* * * * * * *

Professor Armando Fox
Research Advisor/Second Reader

(Date)

# Nye's Trie and Floret Estimators: Techniques for Detecting and Repairing Divergence in the SCADS Distributed Storage Toolkit

Jesse Trutna

March 8, 2010

## Abstract

We present two novel data structures developed in the SCADS distributed storage toolkit [4] for synchronizing replicated datasets with predictable performance: *Nye's trie* is a lightweight index for ordered key-value sets that supports synchronization with time and bandwidth utilization proportional to the number of diverging entries. While efficient, this process is only predictable if the number of divergent entries can be measured. For this, we introduce the *floret estimator*, a novel sublinear-space set summarization structure used to estimate the cardinalities of set difference, union, and intersection operations.

We describe how these structures satisfy the design requirements of the SCADS system, detail their design and implementation, and present a set of microbenchmarks demonstrating their functionality.

## 1 Background

SCADS is a distributed storage system toolkit targeting web-scale, interactive applications. The goal of the project is to achieve "scale-independence," or the ability to scale an application from a single user to hundreds of millions without application modification or operator intervention. To provide scale-independence, SCADS leverages techniques from machine learning, utility computing, and a custom query language, PIQL [19].

Accepting that consistency relaxation is either desirable or unavoidable for many applications[1], SCADS embraces it to enhance scalability. All PIQL queries include explicit performance and consistency requirements of the form "FETCH user BY lastname, must complete in 30ms and be no more than 2 minutes out of date." Reifying consistency allows SCADS to propagate updates and maintain indexes asynchronously.

To handle increasing traffic, a machine learning agent, the Director, uses models of system behavior to automatically allocate or deallocate resources necessary to satisfy query requirements. The Director may request or release machines from the utility computing environment; migrate, replicate, and/or coalesce hot or cold data regions; create or destroy indexes; allocate or deallocate resources for index maintenance and update propagation; or perform any other action necessary to minimize resource consumption while maintaining service level objectives (SLOs). Achieving minimum resource utilization requires monitoring traffic and workload patterns, modeling the current state of the system, and predicting the effects of all potential actions. As a consequence, synchronization, like all mechanisms used by the Director, must have predictable performance.

### 1.1 Architecture

At its core, SCADS is a distributed key-value store in the vein of Chord [23], Dynamo [12], and BigTable [10]. This design pattern is common to most of

1. The presence of social networks make datasets non-partitionable, requiring inter-component coordination. In distributed systems, it is impossible to preserve availability in the presence of failures without relaxing consistency.[14]

the scalable data storage systems currently emerging from industry [17][1][2]. Unlike existing systems, SCADS supports a high level language that federates queries into lookups over materialized indexes. This raises the programming abstraction, but makes update propogation more complicated.

Records are partitioned over a cluster of independent key-value storage nodes called storage engines. Commonly, keys will be sorted lexicographically and each storage engine will be responsible for some small number of contiguous ranges. A data placement layer keeps track of the assignment of key ranges to storage engines and forwards this information to clients on request. (See Figure 1)

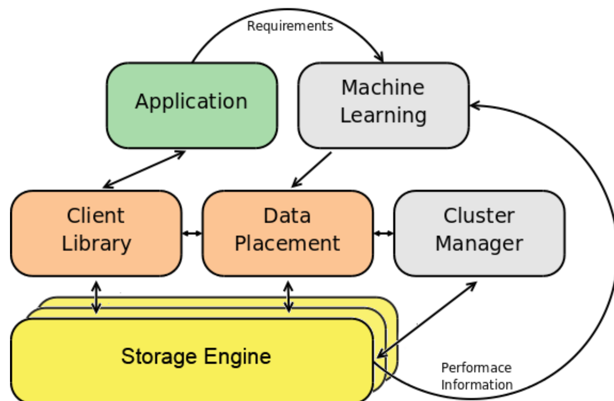


Figure 1: SCADS Architecture

Storage nodes unify caching and primary storage - if the Director needs to increase the server capacity for a particular piece of data, it allocates a storage engine from the utility computing environment, replicates the necessary data, and updates the mapping in the data placement layer. Synchronization is performed between storage engines to maintain replica consistency and as a step in the data migration process.

## 1.2 Storage Engine

In addition to synchronization, storage engines have two mechanisms that allow them to be directed effectively: responsibility awareness and a direct inter-engine copy channel. The Director combines the three mechanisms to perform all data migration and synchronization operations.

### 1.2.1 Responsibility Awareness

Each storage engine is made aware of the set of data for which it is responsible. Storage engines will return an exception if they receive a request for data outside their responsibility.

### 1.2.2 Direct Copy Port

SCADS requires frequent transfers of data between storage servers. To avoid creating bottlenecks at the data placement or director components, storage engines feature a optimized inter-engine communication channel. This channel is used for short-term, pairwise communication between storage nodes for copying or synchronizing data.

### 1.2.3 Synchronization/Reconciliation

The synchronization API call takes three arguments: destination host, synchronization range, and a conflict resolution function. Any key-value pair within the synchronization range that exists at the source, but not the destination, or which has differing values at the two hosts is passed to the conflict resolution function. The resulting value is then installed at the destination host.

## 1.3 Role of Synchronization

### 1.3.1 During Scale Up / Scale Down

As traffic increases, the Director will notice increasing response times or increased latencies in the propagation of updates to replicas. Depending on available resources and the distribution of load across the dataset, it will either replicate hot data items or migrate data away from overloaded machines. As traffic decreases or particular items become less popular, performance will improve and update latency will decrease. Eventually, the Director will coalesce data and release unnecessary resources to reduce costs.

Migrating data during scale up and scale down is very similar. If the destination does not have a copy of the data, the Director first instructs the source server to copy the specified range to the destination server. During this copy, writes will continue to arrive at the source. Once the copy is complete, the Director adds the region to the responsibility policy at the destination, updates the mapping at the data placement layer, and removes the region from the responsibility policy of the source. Finally, to propagate writes that arrived at the source before the responsibility policy was set, the migrated region is synchronized with an appropriate conflict resolution function.

1. `source.copy_set(destination, 'm-z')`
2. `destination.set_responsibility('m-z')`
3. `source.set_responsibility('a-l')`
4. `source.sync_set(destination, 'm-z', resolver)`
5. `source.remove_set('m-z')`

Replication during scale-up is essentially identical to migration, except the original node retains responsibility for the copied range. To maintain consistency between the replicas, sync is called as often as necessary to maintain SLOs.

1. `source.copy_set(destination, 'm-z')`
2. `destination.set_responsibility('m-z')`
3. `source.sync_set(destination, 'm-z')` (periodically)

To remove a replica during scale-down, the region is removed from the responsibility policy and the range is synchronized.

1. `source.set_responsibility('m-z')`
2. `destination.sync_set(source, 'm-z')`

The exact interleaving of responsibility assignment, copy, and synchronization procedures produce different consistency-availability tradeoffs. For example, if, during the migration procedure, the responsibility policy is set before the data is copied, synchronization isn't necessary, but data will be unavailable for the duration of the copy.

#### 1.3.2 For Failure Recovery

Partial failures are a reality of distributed storage systems; individual machines, network switches, and inter-datacenter links are all prone to failure or congestion. Maintaining availability under these conditions may require accepting writes on both sides of a partition. When the partition is repaired, data replicated across the partition must be synchronized. In the case of single-machine or rack failures, this can be done by a one-way sync from the non-isolated replica. In the case of a non-isolating partitions, synchronization must be performed in both directions.

#### 1.3.3 For Anti-Entropy

With replication, a particular data item may be read from any of several locations. Reading and writing to a majority of replicas will prevent stale reads but can lead to reduced performance or downtime in the presence of failures. Non-quorum reads and writes solve this problem, but introduce the possibility of returning arbitrarily stale results. Synchronization can be used to ensure the propagation of updates in accordance with user-specified consistency requirements.

## 2 Sync Overview

The synchronization mechanism in SCADS must fulfill a variety of roles. Migration, failure recovery, and anti-entropy each place unique constraints on the design of the synchronization process. We summarize these requirements below.

**Efficient Synchronization over Subsets** Synchronization usually occurs over small regions of a node's data. Even when moving large ranges, the Director will iteratively migrate smaller pieces to simplify modeling. The synchronization process must be able to handle synchronizing over these small regions efficiently.

**Proportionality in Time and Bandwidth** The most important property for our synchronization implementation is that it be able to

synchronize over a wide range of divergences efficiently. For cases where divergence is low, like migration and anti-entropy, it should use minimal network bandwidth and complete quickly. For cases where divergence is moderate or high, the synchronization structure should not impose significant network overhead.

**Minimal Memory/CPU Overhead** Reducing either the memory available for data storage or the service capacity of a single machine directly increases the total number of servers required. In utility computing environments, additional servers means higher operational costs.

**Predictable Performance** In order for the Director to operate effectively, it must be able to predict the performance impacts of the operations it invokes. Assuming the proportionality requirement is achieved, the Director must be able to estimate the degree of divergence between replicated regions before beginning synchronization.

# 3 Nye's Trie

Named in honor of Bill Nye, Nye's Trie is a lightweight, incrementally-updatable aggregation and search-based synchronization mechanism. Nye's trie uses a modified burst trie [15] to build a lightweight index for generating Merkle trees [18] over subsets of an ordered key-value database. The use of a burst trie gives us a tunably space-efficient way to store a tree of precomputed results and ameliorates the consequences of having many small records. Merkle trees [18] allow us to synchronize remote sets efficiently.

Unfortunately, burst tries are not directly amenable for use as a synchronization structure. We detail a set of modifications that enable the efficient construction of Merkle trees over burst tries and allow efficient use of a pre-existing backing store.

## 3.1 Supporting Synchronization

### 3.1.1 Replace Interior Hashing with XOR-mixing

Merkle trees support synchronization with running time and bandwidth utilization proportional to the number of differences between sets, but are relatively expensive to maintain. When a key-value pair is inserted, the hash of every node along the path from root to leaf must be recalculated. (See Figure 2) This requires reading every sibling of every ancestor of the inserted key and recomputing $H$ cryptographic hashes, where $H$ is the length of the path.

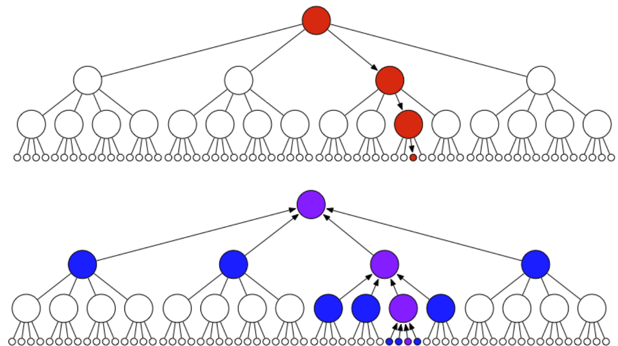

Figure 2: Insertion with cryptographic rehashing

The problem is exacerbated by moving to a burst trie. Here, when a record is inserted, the container node must recalculate the hash from the contained nodes. This necessitates recreating a complete trie comprised of all contained children, calculating the hash value for every leaf and interior node and returning this value. This may need to be done for up to $b$ containers, where $b$ is the maximum branching factor of the trie. In addition, maintaining the integrity of the hash calculation means locking the contained range during the recalculation of the hash.

In a Nye's Trie, we avoid these difficulties by slightly weakening the anti-collision properties of the the tree. Instead of computing a cryptographic hash at each node, we generate cryptographic hashes only at the leaves and combine interior values by taking their XOR. Since XOR is commutative and associative, updates are fast - updating the trie only requires

XOR'ing in the hashed value to every node in the root to leaf traversal. (See Figure 3)

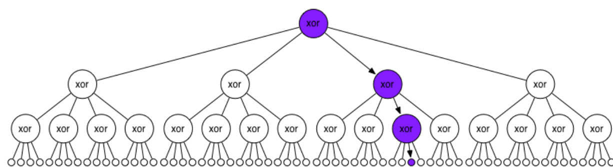


Figure 3: Insertion with xor mixing

This gives us an extremely efficient way to update the synchronization structure. On insertion, we calculate a cryptographic hash for the new key-value pair. We then perform a simple traversal of the trie, XOR'ing in the new hash as we descend.

Since XOR is its own inverse, updates are similarly simplified. The old value is read from the backing store, its hash is calculated, he hash of the new value is calculated, and a single traversal from root to container node is performed, XOR-ing both values along the way.

It seems clear that replacing cryptographic rehashing with XOR will weaken the anti-collision properties of the trie - XOR is associative and cryptographic hashes are not. Still, XOR is entropy-preserving, and cryptographic hashing at the leaves provides a degree of protection. Intuitively, we note that the introduction of a new key-value pair to the trie has a 50%[2] chance of flipping each bit of the root hash, giving us strong avalanche properties. A formal analysis of the effects are future work, but similar tricks have been used in other papers [8] without comment.

#### 3.1.2 Implicit Containers

With XOR-mixing, we can use a burst trie as the basis of our synchronization structure. But doing so would be inefficient in terms of memory utilization - the key of every key-value pair in the database would be redundantly stored in a container node. In SCADS, we already have a data structure that stores the key-value pairs, the BDB tree that backs the storage engine. Nor is this a minor concern. Data items are likely to be small, on the order of hundreds of bytes. And with lexicographic sorting, keys are often semantically meaningful. They may include date stamps, GUIDs, and other meaningful data. Redundant storage of keys would likely incur significant space overhead for many applications

We can leverage the backing store to accomplish significant memory savings at the expense of a small increase in reconstruction time during synchronization. First, we replace each container node in the burst trie with a pointer to the appropriate portion of the BDB Tree. We can accomplish this implicitly since the path to a container node is precisely the prefix of the keys of the records it contains. Recognizing this, we turn container nodes into simple structures containing just the number, total size of records, and XOR'd digest of all "contained" records. If a container node needs to be burst, we simply retrieve all records prefixed by the path to the node from the backing store.

We no longer redundantly store key suffixes and, as important, we no longer store a digest for every key-value pair inserted into the trie. Again, key-value pairs are likely to be small, on the order of hundreds of bytes, and digests are large in relation - a 16-byte digest for each 100-byte record would impose a 16% memory overhead. Instead of storing a digest for each entry, we store digests for groups of entries whose aggregate size is approximately that of the burst threshold for containers. If we are required to examine the digest for entries in a container node, we simply burst it and reconstruct the trie of its contents.

The bursting size of containers is a tunable parameter. With it, the operator can control the tradeoff between the total size of the digest trie and the latency penalty for bursting containers. A reasonable size for a container would be some small multiple of the systems' block size.

#### 3.1.3 Efficient Sub-Trie Construction

The only occasion where synchronization is performed over a entire range is in the case of a partition between two identical replicas. Even in this case, the Director is likely to perform synchronization in a piecemeal fashion. It is much more common

2. Assuming a good hash function.

for synchronization to be performed over small portions of a replicated region. Before synchronization can begin, participating nodes must construct tries that cover just the requested ranges.

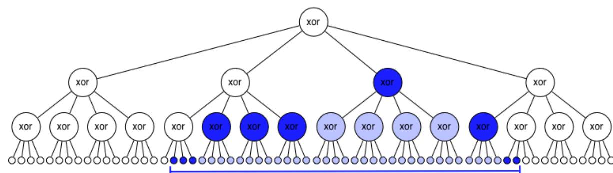

Figure 4: Filtering without hash recalculation

Replacing traditional hashing with XOR-mixing offers advantages here as well. Since XOR is associative and the digest of an interior node is the XOR of the hashes of all records with that prefix, we can quickly reconstruct a Nye's trie by simply pruning the parent trie. (See Figure 4) This requires traversing the trie with the begin and end keys for the range and splitting at most two container nodes.

### 3.2 Application to General Aggregates

We note that the technique detailed above for construction of subtries could be used to compute any associative aggregate over a contiguous subset of entries. Bursting and coalescing nodes already requires maintaining the count and aggregate size of inserted key-value pairs. Invertible, associative aggregates like sum and count can be maintained in exactly the same top-down fashion as the hash digest. Non-invertible aggregates like max or min require an additional bottom-up pass to handle deletions. Returning the value stored at the pruned root node allows a Nye's trie to be used as a index for calculating these aggregates. Returning subsequent levels allows the trie to be used as a kind of "prefix-width" histogram.

## 4 Floret Estimator

Providing time and bandwidth proportionality is not sufficient to make synchronization predictable: Without knowing how divergent candidate regions are, the Director cannot estimate how long synchronization will take. Ideally, we would be able to estimate the exact number of differing elements at each host. Unfortunately, existing set similarity techniques focus on fractional overlap and are either not incrementally updatable [7] or have estimation time linear in the size of the compared sets [3].

We present preliminary work on a new set similarity estimation technique, the *floret estimator*, that estimates the number of non-shared values in each of the compared sets, can be updated to reflect insertions and deletions, and has estimation time linear in the size of the sketch. The estimator's accuracy is inversely proportional to the number of differing items, making it functional as a set checksum.

A floret estimator is built around a degenerate counting Bloom filter [13]. To construct a floret estimator, each host builds a sketch consisting of an array of $n$ counters. Each element of the set is then hashed to a positive integer less than $n$ and the respective counter is incremented. For deletions or updates, the counter is decremented. To estimate similarity between two sets, we construct a new array by subtracting sketches element-wise. By examining the distribution of resulting values, we can estimate the number of entries unique to each side: The variance gives an estimate of the total number of differing values. The mean measures the relative assignment of these differences.

More formally, we construct a floret estimator by initializing an array of $n$ $b$-bit counters. $n$ and $b$ are selected to ensure that no particular bucket will receive more than $2^b$ items and to tune the accuracy of the estimator.[3] We hash each element to a counter using a hash function with good avalanche properties. (See Figure 5) To estimate the similarity between two sets, we construct a new array $C$. If $A_i$ is the value of the $i$-th counter for estimator $A$ and $B_i$ is the value of the $i$-th counter in estimator $B$, we set

$$C_i = A_i - B_i$$

3. The expected size is $m/n$ with variance $m/(n(n-1))$ where $m$ is the cardinality of the represented set. Increasing $n$ decreases the variance of the estimator.

We then calculate the sample mean

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} C_i.$$

and sample variance

$$S^2 = \frac{1}{n-1}\sum_{i=0}^{n-1}(C_i - \bar{x})^2$$

and return a pair of values, $[\frac{n}{2}(\frac{n}{n-1}S^2 + \bar{x}), \frac{n}{2}(\frac{n}{n-1}S^2 - \bar{x})]$ representing the number of unique elements in the left and right sets, respectively.

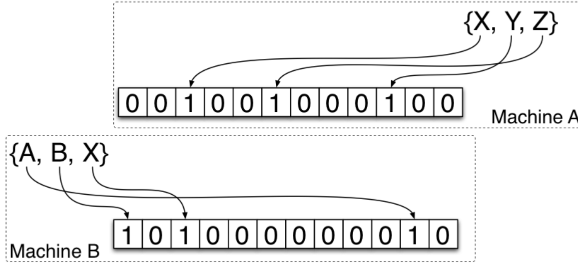


Figure 5: Floret estimator construction

## 4.1 Derivation

Consider a floret estimator $f_A$ built over a set $A'$ and a floret estimator $f_B$ built over a set $B'$ using the same number of buckets and the same hash function. Let $A'_i$ be the value of the $i$-th counter of $f_A$. Let $B'_i$ be the value of the $i$-th counter of $f_B$. Let $C$ be an array of size $n$ where

$$C_i = A'_i - B'_i$$

We note that any element $|A' \cap B'|$ will map to the same counter in both $f_A$ and $f_B$. Thus

$$C_i = A_i - B_i$$

where $A_i$ is the number of elements from $A-B$ falling into the $i$-th counter of $f_A$ and $B_i$ is the number of elements from $B - A$ falling into the $i$-th counter of $f_B$. Since all elements are assigned to random buckets, $A_i$ is a binomial random variable with mean

$$a = |A - B|$$

and

$$p = 1/n$$

and $B_i$ is binomial with mean

$$b = |B - A|$$

and

$$p = 1/n$$

We note that $a$ and $b$ are precisely the values we would like to estimate. Using the properties of expectation and variance, we derive

$$\begin{aligned}
\mathbb{E}[C_i] &= \mathbb{E}[A_i - B_i] \\
&= \mathbb{E}[A_i] - \mathbb{E}[B_i] \\
&= \frac{a}{n} - \frac{b}{n} \\
&= (a-b)\frac{1}{n} \\
Var[C_i] &= Var[A_i - B_i] \\
&= Var[A_i] + Var[B_i] - Cov[A_i, B_i] \\
&= a\frac{n-1}{n^2} + b\frac{n-1}{n^2} - 0 \\
&= (a+b)\frac{n-1}{n^2}
\end{aligned}$$

We know have two equations and four unknowns. We don't know the true expectation and variance of $C_i$, but we do have $n$ "samples" from $C_i$: $C_0$, $C_1$, ..., $C_{n-1}$. We can use these samples to calculate the sample mean

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} C_i.$$

and sample variance

$$S^2 = \frac{1}{n-1}\sum_{i=0}^{n-1}(C_i - \bar{x})^2.$$

then substitute these unbiased estimators for the unknown expectation and variance.

$$\bar{x} = (a-b)\frac{1}{n}$$

$$S^2 = (a+b)\frac{n-1}{n^2}$$

We can now solve for $a$ and $b$. After algebraic manipulation, we arrive at our estimates for $a$ and $b$.

$$a = \frac{n}{2}(\frac{n}{n-1}S^2 + \bar{x})$$

$$b = \frac{n}{2}(\frac{n}{n-1}S^2 - \bar{x})$$

### 4.2 Error

We note that in calculating the sample mean and sample variance, we've relied on the assumption that our samples from $C$ are independent. They are not. The algorithm could be altered to make the samples independent by incrementing *each* bucket with some fixed probability. This would require the generation of a seeded random number for each bucket and doesn't seem to improve accuracy in practice: The current approach has the advantage that $\bar{x} = \mathbb{E}[C_i] = \mu$ can be calculated exactly, removing estimation error for this quantity. Knowing $\mu$ means the only source of estimation error in our original approach is the error inherent in estimating $S^2$ and the error from assuming independence.

Another important characteristic of both approaches is that the error of the estimator decreases as the number of divergent elements decreases. This follows since $Var(\bar{x}) = \frac{\sigma^2}{n}$, $Var(S^2) = (\frac{2\sigma^4}{n-1} + \frac{\mu_4}{n})$, and $\sigma^2, \mu_4 \to 0$ as $a, b \to 0$. This is a particularly useful property for detecting single record changes in large data sets.

Another way to reduce estimation error would be to maintain multiple floret estimators in parallel. With different hash functions, the sample variances derived from each estimator become samples from the distribution of the sample variance itself. This would also help to minimize the effects of the faulty independence assumption.

An full analysis of the effects of the independence assumption, the tradeoff in accuracy between constructing multiple small floret estimators versus a single large one, and a more formal analysis of all floret estimator properties is an area of future work.

### 4.3 Other measures

Since the sum of the entries in a floret estimator is equal to the size of the represented set,

$$|A| = \sum_{i=0}^{n-1} A_i$$

and the floret estimator estimates $|A-B|$ and $|B-A|$, the floret estimator can be used to derive a number of alternative quantities. These include the cardinality of the intersection

$$|A \cap B| = |A| - |A - B|$$

the cardinality of the union

$$|A \cup B| = |A| + |B - A|$$

the Jaccard coefficient [16]

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

the Jaccard distance

$$J_\delta(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

and set containment [6]

$$c(A,B) = \frac{|A \cap B|}{|A|}$$

.

## 5 Implementation

### 5.1 Nye's Trie

The prototype for our synchronization trie is implemented as a in-memory data structure in Java. To provide a stub implementation of the BDB backing store, we used the Java version of the BDB library.

### 5.2 Floret Estimator

The prototype for the floret estimator is implemented as a simple Java class. The MurmurHash function was used to populate the floret estimator array. The Apache Commons Math library was used to calculate mean and variance.

### 5.3 Synchronization Procedure

The data placement layer begins the synchronization procedure by instructing a storage node to synchronize a range of keys with a destination node and particular conflict resolution function. The source node, $A$, constructs a trie over that range, opens a channel to the remote node and passes its own address, the synchronization range, conflict resolution function, and the hash value of its root node. The remote node constructs its own hash trie over the specified range and compares the hash of the root. If the hashes are the same, it returns true and the synchronization call is trivially complete. If the hashes are not the same, it returns false and synchronization begins.

To perform the sync, both nodes maintain four data structures: A "failed" queue that contains the integer ids of those nodes whose hashes do not match. A "consider" queue containing a serialized representation of a nodes children. A "sync" queue that contains the integer ids of key-value pairs which should be transferred from the source to destination. And a "edge" dictionary that maps integer ids to nodes in the hash trie. The "edge" dictionary represents the unresolved nodes in the synchronization process. It can be thought of as a wavefront descending the trie. The destination maintains a fourth queue, "records", containing key-value pairs that should be resolved.

```
while !sync_finished
    id <- failed.pop
    node <- edge.get(id)
    if node.is_a? container
        node <- burst(node)
    end
    if node.occupancy == 1
        sync << node
    else
        consider << construct_childset(id, node)
    end
end
```

Figure 6: Source Synchronization Thread

All communication between the storage nodes is done asynchronously through the insertion and removal of entries from these queues. Before sending the "false" response, the destination node adds

```
def construct_childset(id, node)
    res = ""
    res << id
    res << node.children.size
    for child_edge in node.out_edges
        child_id = ID_SOURCE++
        child = node.get_child(child_edge)
        edge[child_id] = child
        res << child_id
        res << child_edge
        res << child.hash
    end
    edge.delete(id)
    return res
end
```

Figure 7: Child Set Constructor

```
while !sync_finished
    child_set <- consider.pop()
    id <- child_set.id
    node <- edge.get(id)
    if node.is_a? container
        node <- burst(node)
    end
    for child in child_set
        local_child = node.get_child(child.edge)
        if local_child == null
            sync << child.id
        else if local_child.hash != child.hash
            edge[child.id] = local_child
            fail << child.id
        end
    end
    edge.delete(id)
end
```

Figure 8: Destination Synchronization Thread

a mapping from 0 to its root node in its edge set. Upon receiving a "false" response, the source node puts a mapping from 0 to its root node in its edge set and a 0 in the failed queue. The source and destination then start their respective synchronization threads.

Independent threads at the source drain the "consider" and "sync" queues, transferring child_sets and key-value pairs to the destination "consider" and "records" queues respectively. Similarly, threads exist at the destination which drain the "failed" and "sync" queues, transferring integer node ids to the source "failed" and "sync" queues.

A thread at the destination polls the "records" queue and applies the conflict resolution function to the remote and the local values for retrieved records. The result of this resolutionis written into the database. If a bidirectional sync is desired, the algorithm must be modified to achieve consensus on the result in the presence of intervening writes.

### 5.4 Nye's Trie - Bulk Loading

With implicit containers, our Nye's Trie doesn't actually store any of the records it summarizes. This presents a unique problem when constructing the trie over a large pre-built data set: The naive approach of iteratively inserting records requires rereading ranges from the underlying store as containers are repeatedly burst.

To avoid this overhead, we present a algorithm for efficiently bulk-loading a burst trie. We start with a single root node, then insert records in increasing lexicographic order sorted by key.

For each key, we construct a full, uncompressed path through the trie - if the key was 'foo', we would construct nodes corresponding to 'f', 'fo', and 'foo'. After the path is constructed, the previously inserted key is inspected. We consider the currently and previously inserted keys as siblings of the node corresponding to their largest common prefix. We note that no records will be inserted into the subtrie rooted at the child node corresponding to the "left", or previously inserted, sibling. It is thus safe to traverse that subtrie and collapse any branches that do not meet the bursting criteria. Since container nodes are implicit, his is a simple as removing all the nodes children.

Pseudocode follows:

```
class Node
    occupancy = 0
    volume = 0
    children = {}

    insert(key, value)
        occupancy <- occupancy + 1
        volume <- volume + value.size
        if key.size > 0
            edge <- key.charAt(0)
            if !children.include?(edge)
                children[edge] = new Node
            end
            child = children[edge]
            suffix = key.substring(1, key.length)
            child.insert(suffix, value)
        end
    end
end

constructFrom(records) {
    root = new Node();
    prev = []
    for record in records
        key = record.key
        value = record.value
        root.insert(key, value)
        newPath = traverse(root, key)
        i <- 0
        while prevPath[i] == newPath[i]
            i <- i + 1
        end
        while i < prevPath.length
            ptr <- prevPath[i]
            if ptr.volume < THRESHOLD
                ptr.children = null
            end
            i++
        end
    end
    return root
end
```

# 6 Evaluation

### 6.1 Synchronization - Bandwidth

This test measures the network utilization of synchronization as a function of divergence. We construct a replicated region consisting of one million 100-byte

sequential key-value pairs on two separate machines Increasing percentages of both sets were modified and the synchronization algorithm was run. The communication channel between the engines was instrumented to record the total volume of data transferred. The line marked "enumeration" indicates the total size of the range as reported by the underlying berkeley database and represents the bandwidth cost of transferring the entire data set directly. In a real implementation, compression would be used in both algorithms.

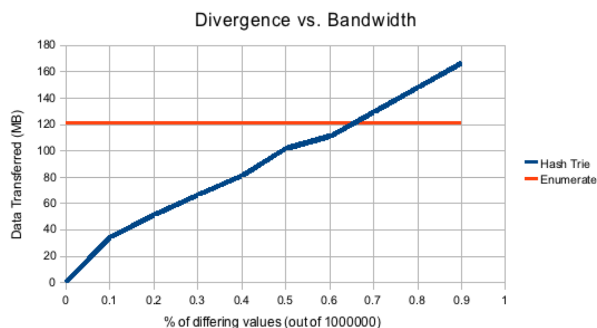


Figure 9: Network utilization of hash trie

Figure 9 shows the desired linear increase in bandwidth utilization as the degree of divergence is increased.

## 6.2 Synchronization - Time

For this test, we again constructed a replicated region consisting of one million 100-byte key-value pairs on separate machines. Increasing percentages of both sets were modified and the synchronization algorithm was run.

Figure 10 shows the desired, approximately, linear increase in synchronization time as the degree of divergence is increased

## 6.3 Nye's Trie - Memory Footprint

To test the memory overhead of the Nye's trie, we constructed a BDB database with 400,000 256-byte records, each with a 13-byte key. Using the Classmexer instrumentation library, we measured the deep

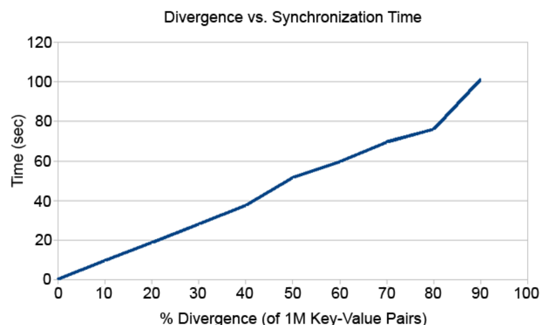


Figure 10: Synchronization time

memory utilization of the database. We then measured the overhead of creating the same database with a hash trie, a Nye's trie with burst size of 4KB, and a Nye's trie with a burst size of 32KB. Figure 11 shows the percentage overhead incurred by each structure.

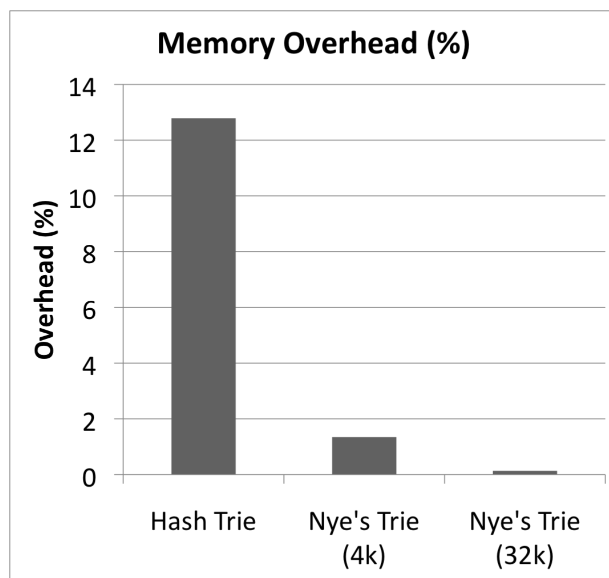


Figure 11: Memory overhead of tries

Figure 11 shows a significant reduction in memory overhead between the Hash Trie and the default Nye's trie (12.8% to 1.3%). Increasing the burst size to 32K

reduces the memory overhead to .14%.

### 6.4 Nye's Trie - Write Overhead

To test the impact of maintaining the Nye's trie on write speed, we measured the time required to insert 100,000 256-byte records into a BDB database without synchronization support, with a hash trie, and with a Nye's trie. Writes were made to sequential keys and, to avoid the overhead of bulk-loading, the structures were pre-populated. This test is really a measure of update speed. In the common case for a Nye's trie, the update path is nearly identical to the regular write path with the exception that one additional hash must be calculated.

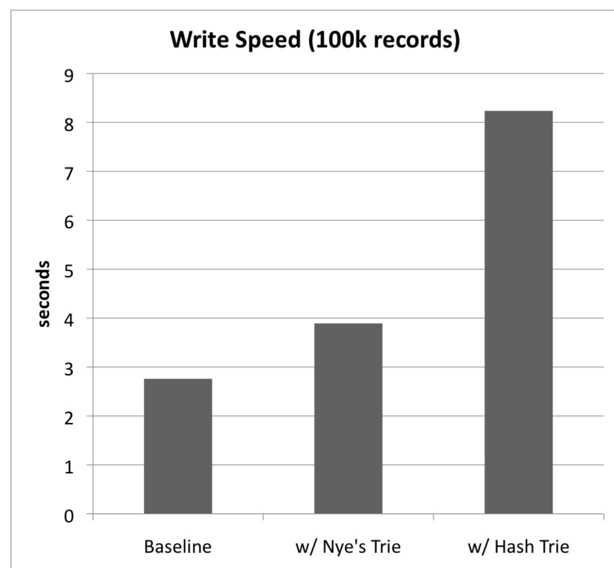


Figure 12: Insertion overhead of tries (non-durable writes)

Figure 12 demonstrates a 64% slowdown for the Hash trie and about a 21% slowdown for the Nye's trie. These were non-durable writes, so updates were not flushed to disk. If BDB is configured to use durable writes, insert performance drops so sharply that overhead from either structure becomes negligible.

### 6.5 Floret Estimator

To test the accuracy of the floret estimator, we construct two sets with 1000 identical elements. From time 0 to 50, we insert additional unique records into the first floret estimator $a$. From time 50 to 150, we insert additional elements into the second floret estimator $b$. The first 50 of these records are the same as those previously inserted into $a$. We chart the actual and estimated number of differing values.

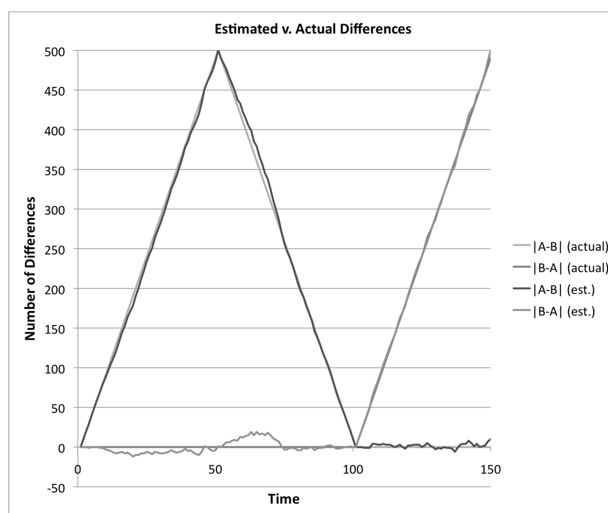


Figure 13: Estimator accuracy

As evidenced by Figure 13, the floret estimator closely estimates the actual divergence of the underlying sets. Around $t = 69$, there is a spike in the estimation of the number of unique elements in the second floret estimator. The value should be zero, but spikes to about 18. This is a consequence of the fact that estimator accuracy is proportional to the total degree of divergence, e.g. the total number of differing entries, not the one-sided difference.

To test the effect of changing the number of buckets in a floret estimator, we construct a series of floret estimator with increasing bucket sizes. (See figure 14) We ran 50 trials, each with 131072 divergent values and recorded the standard deviation of the results as a percentage of the total.

At 512 buckets, or a sketch size of about 1.5KB, the

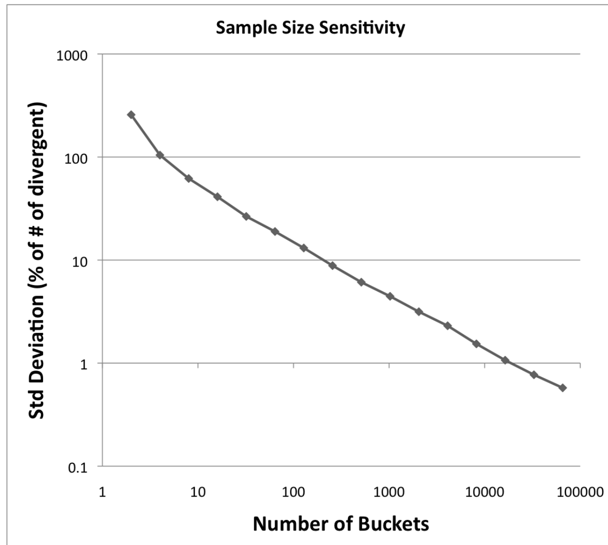


Figure 14: Estimate Std. Dev as % of total number of differing values

floret estimator has a standard deviation of about 6% from the total number of differing records.

# 7 Future Work

In designing the SCADS synchronization mechanism we've focused on satisfying the requirements of higher layers with a simple, versatile approach whose performance characteristics can be predicted easily. There are a number of alternative approaches to the problems of migration and anti-entropy whose performance characteristics are less clear.

Dynamo-like [12] systems use a procedure called read-repair to handle diverging replicas. Here divergence is detected at read time through the use of vector clocks. This might be characterized as lazy conflict resolution whereas our approach is preemptive.Read-repair would be implemented at the client library in the SCADS system, but would remove the necessity of supporting this behavior at the storage engine and might change some of the design considerations for the synchronization structure.

Hinted-Handoff is another technique used by the Dynamo system, this time to improve availability. Under hinted-handoff, if a replica cannot be reached, the write is cached at a peer node who periodically attempts to forward the write to the responsible node. The performance characteristics and complexity of implementing hinted-handoff in a system supporting scale-down is unclear, but supporting this behavior at the storage node may worth investigating.

Data migration is currently a three part procedure implemented by independent mechanisms. Supporting this process as a single primitive could yield significant performance improvements. In particular, the VMWare approach to migrating live virtual machines [11] seems like it could be adapted for synchronization. Under this approach, while performing the initial copy, we would log writes made to the copied region. After the initial copy was complete, we would copy the logged writes to destination host. While copying this smaller set, we would again log any write made to a record being copied. This process of copying and logging writes would continue until the set of pending copies stops shrinking. At that point the responsibility policy would be atomically swapped to prevent further writes, and the final logged set would be synchronized. Modeling this process seems significantly more complicated, but the potential performance enhancements may be worth it

Approaches inspired by log shipping are another avenue for exploration. Our current approach to synchronization determines difference based on value. For frequent synchronizations, this requires retransmitting key-value pairs that may have already been determined not to conflict. A log shipping based approach would include the notion of time. Here, storage nodes would maintain state about their synchronization history, synchronizing only those writes arriving after the previous synchronization. This would introduce a quantity of state between storage nodes which would complicate modeling and scaling. A simpler alternative might be to have storage nodes maintain just a pairwise synchronization log, and only transmit writes that occurred after the preceding sync.

Hash tries also have a number of weaknesses that may be addressed with hybrid or alternate synchronization algorithms: For moderate to high degrees

of divergence, every container (or interior node in an non-burst trie) will contain a diverging node. In these cases, the entire interior of the trie will be sent, necessitating a large number of round trips and unnecessary network overhead.

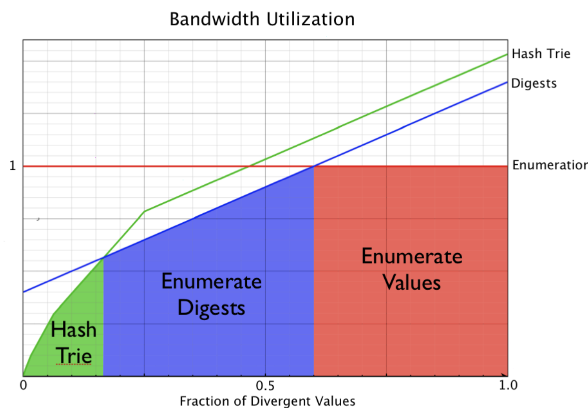


Figure 15: Divergence v. Bandwidth (normalized)

For moderate degrees of divergence, it may be faster and more bandwidth efficient to just transmit a digest for every key-value pair, then respond with unmatched records. For very high degrees of divergence, it may be more efficient to send the entire data set unmodified. (See Figure 15)

# 8 Related Work

## 8.1 Set Reconciliation

Merkle Trees and Hash Tries are both examples of "search-based" approaches to the more general problem of set reconciliation. Here, iterative rounds of communication are used to isolate those elements of the respective sets that do not match.

An alternative to search-based methods are enumeration methods. We have previously mentioned two; sending the entire data set sending a digest for every element of the data set. These approaches give predictable performance, but are bandwidth inefficient when the degree of divergence is small.

In [21], Minsky et al. describe a synchronization approach with nearly optimal communication complexity. Here, entries in the set are represented as $b$-bit integers over a finite field. Each peer maintains a $n$-degree polynomial of the form $(X - e_n)(X - e_{n-1})...(X - e_0)$ where $e_i$ is the $i$th element of the set. This polynomial has roots at exactly the elements of the set. To perform synchronization, nodes exchange the upper terms of this polynomial, divide, and factor to recover missing elements. This yields a one round synchronization process with computational complexity $O(n^3)$. [20] demonstrates a technique to reduce this complexity by engaging in multiple rounds of communication. This technique is complex and the fixed-length requirement poses problems for variable-length data. Variable-length entries can be converted to a fixed length elements through a process like hashing, but this requires maintaining a dictionary mapping hashes to elements which may grow large for high degrees of divergence.

There are also a variety of approximate reconciliation techniques that aren't directly useful for our needs, but could be used as a preliminary step to reduce the degree of divergence and, hence, overall sync time. One approximate approach, presented in [9], is to uses Bloom filters to construct approximate representations of each set. To synchronize, the destination node maintains a Bloom filter of all its records and transmits this to the source. The source iterates through its elements, checking for their presence in the Bloom filter. Any element not found in the filter is known to either be a mismatch or an element only present at the source. False positives will result in diverging replicas being missed. Bloom filters induce minimal communication overhead, but require the source to iterate through its entire set of entries. This makes the approach infeasibly slow for small degrees of divergence.

In [8], Byers and Mitzenmacher detail a hybrid merkle tree/Bloom filter approach that collapses interior and leaf nodes into Bloom filters and uses this to prune the hash tree search. A variant of this may be useful for reducing the degree of divergence before running our exact synchronization approach and bears investigation.

Set reconciliation is also related to the more general area of error-correcting codes and there may be techniques from this area of work that could be adapted.

### 8.2 Set Similarity

The simplest technique for the similarity of two sets are based on random sampling. Here, a set of elements is selected more or less at random from one set and checked for inclusion at the other. The fraction of shared elements is used to estimate the similarity of the sets. Ensuring an accurate measurement generally requires a large sample and techniques do not handle small degrees of divergence well. Estimating similarity requires transmitting the entire set of sampled elements to the remote host and performing a lookup for every element. This is slow and potentially bandwidth intensive for large samples.

A more efficient way to estimate similarity is to use a sketch, or approximate set representation. Assuming a finite, ordered universe of elements $U$, min-wise sketching [5] involves creating $N$ random permutations of the universe, applying each permutation to the input set, and taking the minimum element of each resulting set. This universe is generally a finite set of consecutive integers. This results in a vector of $N$ elements. Similarity is estimated by comparing the ratio of matching elements to the total number of elements. A particular element will match only if the same pre-image exists in both sets (w.h.p. if hashing is applied). The sketch may be updated when new elements are added to the represented sketch by applying the $N$ permutation functions and taking the minimum of this generated value and the current value in the sketch. Min-wise sketches are a compact, fast way to estimate set similarity, but cannot handle deletion or updates of existing elements - the "min" operation is not invertible.

Wrapped filters are sketch-based similarity estimation approach that can handle insertions and deletions [3]. Here, a counting Bloom filter is maintained for each set. To estimate similarity, the filter is transmitted to the remote host. The remote host iterates through its entire dataset, checking each element against the Bloom filter. If element matches the Bloom filter, the buckets touched by that element are decremented. Even if the element was not actually in the filter, the decrementing process is likely to cause the actual element to not match when it is checked. Since only the number of matching elements is desired, these two "errors" cancel out. Unfortunately, wrapped filters require iterating through every element in one of the sets and are prohibitively slow for our purposes.

Outside of set similarity, there is a large body of work on estimating similarity for computer forensics, data streams, document clustering, and locality-sensitive hashing. Most of these are focused on finding similarity for static data or append-only data items and do not handle updates or deletions. Still, investigating these areas for applicable techniques is an area of future work.

md5bloom also uses the statistical properties of Bloom filters to estimate similarity, but bases its analysis on the number of matching bits between regular Bloom filters, not the magnitude of difference between counting Bloom filters [22]. md5bloom was designed for computer forensics on static files and does not handle updates or deletions.

# 9 Conclusion

We have demonstrated two novel data structures, the Floret Estimator and Nye's Trie, which form a synchronization mechanism for the SCADS data storage system that is predictable, lightweight in both memory and CPU overhead, and uses time and bandwidth in proportion to the degree of divergence in the synchronized range.

# 10 Acknowledgments

It is with deep gratitude that I'd like to thank my advisors Armando Fox and David Patterson; Professor Michael Franklin; fellow graduate students and SCADmates Nick Lanham, Michael Armbrust, Beth Trushkowsky, Peter Bodik, and Kristal Sauer; undergraduate researchers Steven Schlankser, Borden Liu, and Haruki Oh; the RAD Lab; and U.C. Berkeley for the opportunity to work, study, and discover with them on a fun and fascinating project. I'd also like to thank Andrew Krioukov for harassing me about early versions of my abstract, Blaine Nelson for help with maths, Neil Conway for insight into databases,

and the CSUA's Brandon Liu for assistance in naming my data structures. Finally, I must express my deep, personal gratitude to the illustrious Phillip Edward Nunez for being such an unbelievable friend, mentor, and inspiration. Thanks Phil.

I'd also like to thank the many sponsors without whom my research would not have been possible. This research was supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant #CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

# References

[1] Hypertable: An open source, high performance, scalable database. http://hypertable.org/.

[2] Project Voldemort. http://project-voldemort.com/.

[3] Agarwal, S., and Trachtenberg, A. Approximating the number of differences between remote sets. In *IEEE Information Theory Workshop, 2006. ITW'06 Punta del Este* (2006), Citeseer, pp. 217–221.

[4] Armbrust, M., Fox, A., Patterson, D., Lanham, N., Trushkowsky, B., Trutna, J., and Oh, H. SCADS: Scale-independent storage for social computing applications. In *Proc. CIDR* (2009).

[5] Broder, A., Charikar, M., Frieze, A., and Mitzenmacher, M. Min-wise independent permutations. *Journal of Computer and System Sciences 60*, 3 (2000), 630–659.

[6] Broder, A., et al. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences* (1997), vol. 1997, Citeseer.

[7] Broder, A. Z., Charikar, M., Frieze, A. M., and Mitzenmacher, M. Min-wise independent permutations. *Journal of Computer and System Sciences 60* (1998), 327–336.

[8] Byers, J., Considine, J., and Mitzenmacher, M. Fast approximate reconciliation of set differences. In *BU Computer Science TR* (2002), pp. 2002–19.

[9] Byers, J., Considine, J., Mitzenmacher, M., and Rost, S. Informed content delivery across adaptive overlay networks. *IEEE/ACM Transactions on Networking (TON) 12*, 5 (2004), 767–780.

[10] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *OSDI 06* (November 2006).

[11] Clark, C., Fraser, K., Hand, S., Hansen, J., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines.

[12] Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. In *SOSP (2007)* (New York, NY, USA).

[13] Fan, L., Cao, P., Almeida, J., and Broder, A. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON) 8*, 3 (2000), 293.

[14] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. In *In ACM SIGACT News* (2002), p. 2002.

[15] Heinz, S., Zobel, J., and Williams, H. E. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst. 20*, 2 (2002), 192–223.

[16] Jaccard, P. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles 37* (1901), 547–579.

[17] Lakshman, A., and Malik, P. Cassandra: A structured storage system on a p2p network. Presented at SIGMOD 2008.

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

[19] Michael Armbrust, Nick Lanham, S. T. A. F. M. J. F. D. A. P. Piql: A performance insightful query language for interactive applications, 2010.

[20] Minsky, Y., and Trachtenberg, A. Practical set reconciliation, 2002.

[21] Minsky, Y., Trachtenberg, A., and Zippel, R. Set reconciliation with nearly optimal communication complexity. In *in International Symposium on Information Theory* (2000), p. 232.

[22] Roussev, V., Chen, Y., Bourg, T., and Richard, G. md5bloom: Forensic filesystem hashing revisited. *digital investigation 3* (2006), 82–90.

[23] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. pp. 149–160.