2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)

ePipe: Near Real-Time Polyglot Persistence of HopsFS Metadata

Mahmoud Ismail*, Mikael Ronström†, Seif Haridi*, Jim Dowling*
*KTH - Royal Institute of Technology, † Oracle
{maism, haridi, jdowling}@kth.se, mikael.ronstrom@oracle.com

***Abstract*—Distributed OLTP databases are now used to manage metadata for distributed file systems, but they cannot also efficiently support complex queries or aggregations. To solve this problem, we introduce ePipe, a databus that both creates a consistent change stream for a distributed, hierarchical file system (HopsFS) and eventually delivers the correctly ordered stream with low latency to downstream clients. ePipe can be used to provide polyglot storage for file system metadata, allowing metadata queries to be handled by the most efficient engine for that query.**

For file system notifications, we show that ePipe achieves up to $56X$ throughput improvement over HDFS INotify and Trumpet with up to 3 orders of magnitude lower latency. For Spotify's Hadoop workload, we show that ePipe can replicate all file system changes from HopsFS to Elasticsearch with an average replication lag of only 330 ms.

I. Introduction

There has been significant growth in recent years in the volume of data stored in distributed file systems. With such a growth, it becomes increasingly difficult to index, manage, and track the origin and use of data within large datasets. Distributed file systems traditionally minimize the amount of metadata used for files and directories to ensure that the metadata can be managed in-memory by a single process [1]. However, as distributed file systems inexorably grow in size, this approach becomes increasingly untenable. HopsFS is a drop-in replacement for the Hadoop Distributed file system (HDFS) that solves this problem by storing its metadata in a distributed in-memory NewSQL database [2].

However, more metadata brings new challenges. As the number of files increases, finding files becomes more challenging and new requirements, such as the right-to-be-forgotten enshrined in the European GDPR legislation [3], necessitate distributed file systems to provide richer query support for their metadata than what is currently available in existing distributed hierarchical file systems. Such query support should, of course, return correct and fresh results, that is, from a very recent consistent view of the metadata. However, there is no single database that can efficiently process all query patterns on metadata [4]. Recently, the idea of polyglot persistence has become popular, where replicas of the data should be stored in more than one data store, enabling the data to be efficiently queried from different engines, such as OLAP or full-text search engines [5], [6]. The main research challenge in implementing polyglot persistence is ensuring reliable, timely, and consistent replication of the data between the different engines.

HopsFS stores its file system namespace metadata in a NewSQL database, MySQL Cluster. To support polyglot persistence of HopsFS' metadata in external stores, each external store needs an eventually consistent replication protocol to synchronize its metadata with the main NewSQL metadata store. Queries on the external store, such as full-text searches, should return correct results, but the results may be stale due to replication lag (the latency of replicating metadata from the NewSQL store to the secondary store). We consider such a replication protocol near real-time, if the vast majority of events replicated with sub-second replication lag, however, there is no guaranteed bound on the delay introduced by the processing and delivery of metadata to the external store.

In this paper, we present ePipe, a metadata system for HopsFS that provides replicated-metadata-as-a-service. The key component of ePipe is a databus that both creates a consistent, correctly-ordered change stream from HopsFS and eventually delivers the stream with low latency (sub-second) to external stores and downstream clients. Our main contribution in ePipe is a formal model and an implementation that generates a consistent file change stream from the unordered changelog produced by the NewSQL database (MySQL Cluster [7]). The problem with the unordered changelog is that it can include out-of-order events from a file system perspective (such as delete a file before it has been created). To solve this problem, we developed a highly performant event re-ordering protocol that ensures the correctness of ePipe's output replication stream, while not requiring the serialization of all events in the database changelog - to yield high performance.

In experiments based on a real-world Hadoop workload from Spotify, ePipe can achieve up to $56X$ throughput improvement for file system notifications over HDFS INotify and Trumpet with up to 3 orders of magnitude lower latency. We demonstrate that ePipe can scale to file system throughput levels several times higher than the largest HDFS deployments found today. Even at this scale, ePipe can consistently replicate all the file system changes from HopsFS to Elasticsearch in sub-second replication lag.

II. Background

In this section, we describe HDFS [1] and HopsFS [2] and present what they offer to support polyglot persistence of the metadata, that, in turn, would enable services such as an efficient query service of the namespace.

978-1-7281-0912-1/19/$31.00 ©2019 IEEE
DOI 10.1109/CCGRID.2019.00020

A. Hadoop Distributed File System (HDFS)

HDFS [1] is a distributed hierarchical file system that stores its namespace metadata in a single metadata server called the namenode. In HDFS, all file system operations are logged into a transaction log called the edit log [1]. The edit log is written to a quorum of journal nodes in high availability setups. Each file system operation is assigned with a monotonically increasing $transaction\_id$.

HDFS implements an *inotify* service in the namenode [8], where a client periodically polls the namenode for new transactions that happened after a given $transaction\_id$. This approach has poor scalability, and it doesn't support fine-grained watches over a specific directory. Trumpet [9] was developed to provide an inotify service for HDFS that does not poll the namenode - the namenode is already a bottleneck in HDFS [2]. Trumpet periodically polls the edit logs from the local file system of the namenode or a journalnode, and publishes the transactions as events into a Kafka topic. Clients then consume events from the Kafka topic. However, such an approach introduces a higher replication lag compared to the native inotify service provided by HDFS due to the cost of polling the local file system and publishing to Kafka. Moreover, HDFS implements a *find* operation [10] to search for files/directories based on their name. However, the solution is inefficient as it blindly traverses (scans) the whole namespace.

B. HopsFS

HopsFS [2] is a new distribution of HDFS that replaces HDFS' namenode with a distributed metadata service, where the metadata is stored fully normalized in a database. In HopsFS, multiple stateless namenodes are used to control access to the database. Currently, only MySQL Cluster (NDB) is supported as the backend database, but a plugin architecture allows, in principle, any database with support for transactions and row-level locking to be used [2]. In HopsFS, inodes (files or directories) are stored as rows in the *inodes table* with a primary key.

1) MySQL Cluster (NDB)

NewSQL databases are distributed, in-memory relational databases that partition tables over many database nodes [11]. MySQL Cluster's NDB storage engine is a NewSQL database that achieves high performance by supporting concurrent non-serialized transactions [7]. HopsFS uses locking primitives to ensure file system consistency for cross-partition transactions. There are alternative NewSQL architectures that use a global transaction manager and multi-version concurrency control to scale out over many nodes, such as MemSQL [12] and SAP Hana [13], while others, like VoltDB [14], serialize cross-partition transactions.

MySQL Cluster supports the NDB (Network Database) storage engine with different APIs for accessing its data, C++, Java, and SQL. Given that HopsFS' metadata resides in NDB, an administrator could easily use SQL to search for files based on their attributes (such as name, size, and modification time). However, as MySQL Cluster is an OLTP database, many types of queries (such as, *SELECT * from inodes*) have the potential to overload the database which would, in turn, affect HopsFS' stability and performance. Moreover, NDB does not support full-text search over columns.

III. Notifications for HopsFS

Notification APIs are common for monolithic file systems but are rarely found in distributed file systems, where there is no standardized API for file system notifications. Crawling a large-scale distributed file system without proper indexing is inefficient by design. Moreover, since HopsFS supports many stateless, independent namenodes, and the metadata is stored in NDB, notifications of metadata changes must originate in the database. The approach we take is based on a feature of MySQL Cluster where we can subscribe to watch for ongoing changes to the metadata. The feature is similar to database triggers but is provided as a distributed service (*NDB Event API*) with no historical changelog, just a live change stream. So, a notification service that listens for events from the *NDB Event API* is not sufficient to build a reliable bus to replicate changes to external systems, as, in case of failure, we would miss events leading to inconsistent replication.

Our solution for ePipe is to add a logging table *"file system replication log(frol)"* to the database, where file system metadata changes are logged as part of file system transactions to ensure integrity and consistency of the log. We show later in Section VI that this solution is more efficient with negligible overhead on the file system. The storage overhead of an *frol* entry is $21 + L$ bytes, where L is the length of the file/directory name. For a Spotify workload, the average length of file/directory name is 34, so 1 million *frol* entries consume 56 MB, the database can be scaled to store up to 24 TB [2].

A. Selective Logging of Files/Dirs in HopsFS

We have extended the HopsFS APIs with two RPCs to enable and disable, respectively, selective logging of file system operations for a given subtree (a directory hierarchy in the file system namespace, including its child files/directories). The events that HopsFS logs include file/directory events such as create, append, delete, rename, set owner, and move. These logs are written to the database as rows in the *frol* table. The rows are spread across different database partitions, with the datanode used to store the row determined by the MD5 hash of the row's primary key.

B. Consuming the *frol* entries

Since HopsFS logs file system operations as rows in the *frol* table, we need a way to consume the entries from the *frol* in near real-time to notify interested subscribers. We introduce ePipe, a databus system to consume the *frol* entries and then either provide an enriched version of the events with extended metadata as described in Section V-C, or take actions based on some predefined criteria as described in Section V-E. Figure 1 shows a high-level description of ePipe and its interactions with NDB, HopsFS, and other downstream consumers, such as Elasticsearch and Hive. ePipe starts with a recovery phase (sync) to read all unprocessed events from the *frol*, at the same time it subscribes for any new changes on the relevant tables, without publishing those changes until the recovery is done.

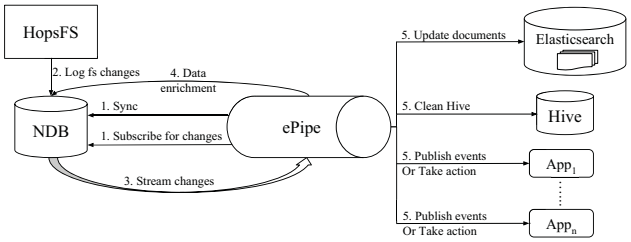


Fig. 1: Architecture diagram of the interactions between ePipe, NDB, and HopsFS. ePipe starts with a recovery phase (sync) to read all unprocessed changes from the database. Also, it subscribes for changes on the logging tables. Whenever a change happens, the database pushes a change event to ePipe, that processes the event(s), and enriches the associated files with extended metadata if available. The enriched events are then published to the subscribers of ePipe.

From then on, any change events in subscribed subtrees in the file system generate a new event that arrives at ePipe, and a data enricher component augments the event with file metadata (owner, group, size, etc.) and any extended metadata for that file or directory (see Section V-B). Finally, event handlers in ePipe either publish the event or take some action as defined for that event type/value.

C. NDB Event API

NDB provides a publish-subscribe API where applications can subscribe to row changes on a table, and then NDB streams the row changes to the corresponding subscribers. Each database node independently generates a change stream for the row changes it was involved in. That is, each database node is responsible for a subset of the change stream, and sends its assigned subset to all subscribers over the Event API, which in turns, groups and merges the change streams coming from all database nodes, and when all change streams for a given *epoch* have been received from database nodes, it publishes the events to the application, ePipe in our case. NDB implements a logical clock known as an *epoch* that is periodically incremented across all nodes in the cluster atomically using a leader driven protocol [15]. Epochs are used to maintain a total order of events on the cluster, that is required by various internal functions such as grouping sets of committed transactions together for later use by the NDB Event API.

IV. ePipe

A. System Model

In HopsFS, file system metadata operations, with the exception of subtree operations, are implemented as a single transaction T in the database. Subtree operations, on the other hand, are executed as a one or more transactions [2]. Read-only (file system) operations do not mutate the metadata, such as reading a file, while mutate-operations update the metadata, such as creating a file. For logging-enabled subtrees, mutate operations also write a log entry to the *frol* table, see Section III-A. For mutate-operations, a transaction T, once committed, updates the state of the database, and then NDB internally outputs any *frol* change events $\{r_x\}$ in T to consumers through the Event API. A *frol* change event r_x is defined as a change to a given inode x in the file system, i.e., a create, append, delete, move, set owner or rename event. Every transaction T belongs to a single epoch e, where e is the NDB epoch and the set of transactions that commit within e is defined as: $e.trans = \{T\}$. By default, the epoch number ($e.num$) is incremented every 100 milliseconds. For any given change event r_x (part of transaction T), we can determine its epoch number using $epoch(r_x) = e.num$. We use the *happens-before* symbol '$\rightarrow$' to denote the happened-before relation between any two given change events [16]. For example, for any two change events r_x and r_y, $r_x \rightarrow r_y$ implies that r_x happened before r_y. NDB provides the following ordering properties for the *frol* entries:

- **Property 1:** $\forall e_i, e_j$ where e_i and e_j are epochs i and j respectively, e_i happens before e_j if $e_i.num < e_j.num$. That is, epochs are totally ordered.
- **Property 2:** $\forall r_x, r_y \in T,\ epoch(r_x) = epoch(r_y)$. That is, all change events in the same transaction have the same epoch number.
- **Property 3:** $\forall r_x, r_y$ where x, y are inodes in the file system $r_x \rightarrow r_y\ if\ epoch(r_x) < epoch(r_y)$

However, these ordering properties guarantee only ordering across epochs not within the same epoch, which is not strong enough, for example, to prevent consumers observing files being deleted before they are created. To address this, we introduce:

Consistency Requirement CR_1: All change events on the same inode (file/directory) should be serialized to ensure a consistent view of the file system metadata.

For two change events (r_x, r'_x) on the same inode (x) in different epochs, the happened-before relation holds using *Property 1* and *Property 3*. However, if r_x and r'_x happen within the same epoch, no order is guaranteed.

In order to satisfy CR_1, ePipe needs to ensure that consumers of change events observe the same order for file system metadata operations as clients in HopsFS. We define another function $trans(r_x) = T$ that returns the enclosing transaction T for a given change event r_x. We use the *transaction-ordering* symbol '$\twoheadrightarrow$' to denote an ordering between two transactions. In HopsFS, ordering between two conflicting transactions is implemented by both transactions acquiring write locks on the same row in the first operation they execute. That is, if $T_1 \twoheadrightarrow T_2$ then T_1 executes before T_2 through the use of write locks.

To ensure the same order of file system metadata operations between consumers of change events and HopsFS clients, we examined different solutions. The naive solution is to annotate all log entries with a monotonically increasing id, a logical clock, to ensure a serializable view of the events. But such a solution would introduce a bottleneck on HopsFS and would degrade its performance, as shown in our evaluation in Section VI-G. Instead, we adapted the logical clock approach to serialize events at the inode level, instead of at the database level. That is, instead of using a single logical clock across all events, we use a logical clock per inode in HopsFS, that is, a *version number* for the inode. This approach doesn't introduce

any bottlenecks since we piggyback the version number on the inode, and the update of the version number is serialized per inode due to the locking mechanisms used by the file system. We define the function $version(r_x) = v$ that, for a given change event r_x on an inode x, returns the inode's version number v at the time of r_x. With our newly introduced version number we can strengthen our ordering properties:

- **Property 4** $\forall r_x, r'_x$ where x is an inode in the file system $version(r_x) < version(r'_x)\ if\ trans(r_x) \nrightarrow trans(r'_x)$
- **Property 5** $\forall r_x, r'_x$ where x is an inode in the file system $if\ epoch(r_x) = epoch(r'_x)\ then\ r_x \rightarrow r'_x\ if\ version(r_x) < version(r'_x)$
- **Property 6** $\forall r_x, r_y$ where x, y are different inodes in the file system $if\ epoch(r_x) = epoch(r_y)\ then\ \neg(r_x \rightarrow r_y)$

With Properties 1-5, the consistency requirement CR_1 is satisfied. Property 6 states simply that ordering doesn't matter between changes events on different inodes in the same epoch - enabling such change events to be replicated in parallel to downstream consumers. Based on our ordering properties 1-6, we devise the event reordering algorithm, see Algorithm 1, to order the file system metadata change events from NDB. The algorithm awaits for new events coming from NDB (line 4). Once a new event arrives, we extract its epoch number and compare it with the last reported epoch (*lastEpoch*) (lines 5-9), guaranteeing *Properties 1-3* . If the event's epoch number is higher than the last reported epoch number (line 9), that is, the last epoch is now completed and all events from the last epoch have been received from the database nodes (see Section III-C), then we reorder all the events stored for that epoch and prepare for the next epoch (lines 9-13). NDB guarantees that all events from epoch n are delivered before events from epoch $n+1$, and there are no late events from a completed epoch in NDB. The events on the same inode are ordered based on their version numbers (line 17-18), guaranteeing *Properties 4,5*, otherwise no order is guaranteed for different inodes within the same epoch, *Property 6*.

Algorithm 1 Event Reordering algorithm

```
Require: Conn                          ▷ Connection to receive events from NDB
Require: OUT                           ▷ Output queue to write the ordered events
 1: lastEpoch ← ⊥
 2: currEpoch ← ⊥
 3: evts ← [ ]
 4: while Conn has new events do
 5:     r_x ← Conn.getEvent()
 6:     currEpoch ← epoch(r_x)
 7:     if lastEpoch = ⊥ then
 8:         lastEpoch ← currEpoch
 9:     else if currEpoch > lastEpoch then
10:         OUT ← SORT(evts, CMP)
11:         evts ← [ ]
12:         lastEpoch ← currEpoch
13:     end if
14:     evts.add(r_x)
15: end while

16: function CMP(r_x, r_y)
17:     if x.inodeID = y.inodeID then
18:         return version(r_x) < version(r_y)
19:     end if
20: end function
```

B. Architecture

The different components that make up ePipe are shown in Figure 2. ePipe has multiple *watch units* running in parallel. Each *watch unit* consists of a *DB Watcher* component which subscribes for change events on its associated table and produces events in-order to the output event queue. The *DB Watcher* component creates a barrier using Algorithm 1 to ensure that all events from a specific epoch are received and sorted before handling the next epoch. The *Batcher* component consumes the events from the *DB Watcher* queue, and if it reaches a configurable threshold of events it outputs a batch to its output queue. To ensure liveness, the *Batcher* has a timer which will periodically fire after a configurable amount of time even if the target batch size has not yet been reached. A group of *Data Enricher* components running in parallel will consume batches from the *Batcher* queue to read the required data from the database, then generate appropriate enriched events while preserving their correct order. The *Data Enricher*s enqueue their resultant, enriched events into a watch unit output queue. Then, using *App Handler*s, ePipe publishes the events to downstream applications, each application implements an *App Handler* that consumes the events and processes them according to application-specific requirements. *App Handlers* can consume the events directly from the *DB Watcher*, that is, bypassing the optional *Batcher* and *Data Enricher* components. For some use cases, if the application requires being notified or needs to immediately take action if a particular change happens, then the *App Handler* will consume directly from the *DB Watcher*. On the other hand, if the application requires a fully enriched event to be published to it, then, the *App Handler* will work normally by connecting to the *Data Enricher* output queue. For instance, the *Elastic Handler* consumes the enriched events from the queue, and merges these events together in a JSON object until the JSON object reaches a configurable size, then using the Elasticsearch Bulk API [17] it pushes these changes to Elasticsearch. Another example, is the *Hive Handler* that consumes events directly from the *DB Watcher* and takes actions to synchronize Hive with the metastore, as discussed in Section V-E. *App Handlers* also have a timer to ensure liveness of ePipe.

Even though we based our implementation in the paper on NDB due to the fact that HopsFS uses it as the metadata database, we believe our approach can be adapted to other NewSQL databases, especially ones that support a similar push-based *Event API*. For databases without an Event API, a well-known approach is to tail the database transaction logs [5], [6], that, could be used where the *DB Watcher* continuously tails the transaction logs produced by the database.

C. Failure Recovery

With the help of the persistent *frol*, ePipe supports failure-recovery. Failures in ePipe do not result in a loss of events at consumers, although for consumers that do not support transactional delivery of events (such as Elasticsearch), it is possible that events will be received more than once (duplicates), which is ok for Elasticsearch since updates to

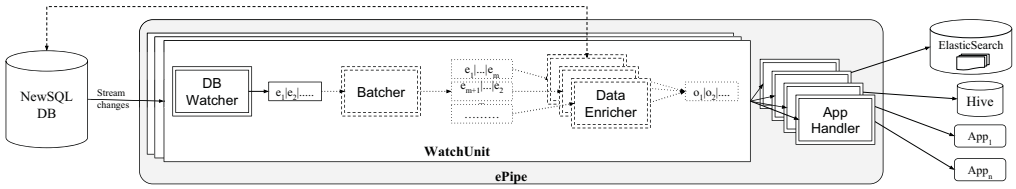


Fig. 2: System architecture for ePipe. ePipe consists of multiple Watch Units and App Handlers running in parallel, where each Watch Unit has a DB Watcher, Batcher, and a set of Data Enrichers. DB Watcher watches for new changes happening on the logging table, and then produces an ordered stream of changes. Batcher batches the individual incoming events into batches to be processed by DataEnrichers. A DataEnricher is an abstract interface to enrich the incoming events with data, for example, enrich the events with extended metadata for the affected inodes. AppHandlers either push the enriched events or execute an action according to the corresponding downstream application. The Batcher and DataEnricher components are optional, that is, the AppHandlers can connect directly to the DB Watcher as defined by the downstream application.

the same document are idempotent because duplicate events are guaranteed to arrive before later events in the *frol*. For Hive connector on HopsFS, however, we support transactional delivery of the event as the destination is the same NDB cluster that the *frol* event came from. In general, for the downstream clients, ePipe provides at least once delivery semantics by ensuring that the event logs are deleted from the *frol* only once they are successfully delivered to all downstream subscribers. If a downstream subscriber failed to receive the events or its handler timed out, then ePipe resends the events to ensure the delivery of a consistent change stream. It is up to the downstream subscriber and its *App Handler* to correctly handle duplicates to provide exactly-once processing guarantees. ePipe could fail at any stage of processing. The persistent *frol* table, however, enables ePipe to reprocess all the unprocessed events upon restart during the initialization phase (*sync*, see Figure 1). During recovery, ePipe reorders the events in the usual way - first by epoch, and then by the version number in case of conflicting operations on the same inode. This guarantees the same behaviour as normal running ePipe. Moreover, ePipe offers a bootstrap functionality to bring new applications up to speed with the *frol*.

ePipe is implemented as a multi-threaded application running on a single host/container with support for failure-recovery. As ePipe is stateless and lightweight, re-starting a failed daemon is fast and recovery time due to short-lived ePipe failures should not be significant. However, in certain scenarios high availability for ePipe is desirable, and this can be easily added by running active-standby ePipe servers, reusing the leader election protocol developed in HopsFS [18]. That is, both active and standby ePipe servers will periodically write and poll their status into/from a row in the database, and, whenever a standby server detects a dead active server, it will take over by starting all the watch units. A dead server is a server that fails to write its status to the database for two consecutive rounds.

V. ePipe Use Cases

In this section, we describe some of the use cases that are enabled by ePipe.

A. Metadata search

We built a fast metadata search service for HopsFS using ePipe and Elasticsearch. For this service, each inode is represented as a document in an Elasticsearch index, and each document is identified by its corresponding inode id. That is, all the change events for a specific inode will update the same document in Elasticsearch. The Elasticsearch Handler is responsible for transforming the enriched events into insert/update/delete actions on the documents in the index.

B. Applications for ePipe

All of the components that make up ePipe are pluggable, enabling ePipe to be configured for a variety of replication and notification connectors. For instance, the *DB Watcher* can be extended to only filter a specific pattern of files or operations. This functionality could be used to efficiently build applications such as intrusion detection systems (IDS) that watch directories and files for mutations and take actions on observing such mutations. Currently, the *DB Watcher* uses NDB Event API to stream the changes from the database, however, it can be extended to support any other database by tailing its transaction log. Another scenario is to extend the *Data Enricher*s to enrich the events with more information before sending them to the downstream applications such as extended metadata associated with files/directories, see Section V-C. Moreover, ePipe notification capability is not limited to HopsFS or file system notifications, but to any system that can write a row in a table and then extend ePipe to stream these changes by adding a new watch unit with a *DB Watcher* that fits the system requirements.

C. Extended Metadata

File systems support basic attributes for files/directories, such as file size, permissions, and modification timestamp. However, in many cases, users and administrators will need to have extended attributes for files and directories beyond the ones available in the default attributes. Many existing file systems provide extended attributes to allow arbitrary attributes to be attached to files and directories. In HopsFS, a file/directory is represented as a row in the inodes table, where an inode is identified by a primary key. Therefore, attaching metadata to a file or directory is as simple as creating another table in the database and adding a foreign key constraint on the primary key columns of the inodes table. We extended HopsFS to support two different possible strategies to attach metadata to inodes, either a *Schemaless* approach or a *Schema-Based* approach. To support extended metadata in ePipe, we added another *Watch Unit* to watch for events happening on extended metadata. Also, we extended the *Data Enricher*s to combine

the extended metadata with the events before sending them to the downstream applications.

Schemaless: In this approach, the metadata should be stored in a self-contained manner, where there is no predefined schema needed to interpret the metadata. Metadata is stored in a single self-contained JSON object that can be attached to a file or a directory.

Schema-Based: In this approach, users can define schemas for their extended metadata similar to how they would create a schema in a relational database. A schema can have multiple columns with different types. The schema-based approach enables the validation of extended metadata by the database.

D. Hopsworks

Hopsworks is a project-based multi-tenant platform for secure collaborative data science built on HopsFS and YARN [19]. It introduces new abstractions of *Projects* and *Datasets* that provide the basis for which users can securely upload and privately process data and securely collaborate with other users on the platform. Hopsworks provides a metadata designer where users can design a metadata schema and attach validated metadata values to files, directories, or *Datasets*. Hopsworks leverages HopsFS and ePipe to provide an intuitive search capability using Elasticsearch, where users can search for files, directories, and *Datasets* based on their attributes or any extended metadata attached. Both *Projects* and *Datasets* are implemented as directory subtrees with extensive extended metadata. Hopsworks writes logging events for *Projects*, *Datasets*, and extended metadata similar to the *frol* entries implemented in HopsFS, and then implements different watch units in ePipe to watch for these logging tables in order to replicate the same *Project*/*Dataset* structure into Elasticsearch.

E. Apache Hive

Apache Hive [20] is a petabyte-scale data warehousing system that allows users to query tables using a SQL-like query language. Hive stores its data as databases, tables, and partitions in a distributed file system (HDFS) while the information schema is stored in a metastore backed by a relational database (by default a MySQL server). However, as the schemas are stored in a MySQL server and the data files are stored on HDFS, inconsistencies can arise between the two systems. In work on Hive for HopsFS [21], Hive's metadata is unified with the inode metadata in HopsFS to ensure strong consistency of the Hive metadata. However, a few tables in Hive could not be linked to inodes with foreign keys (to ensure their integrity), so we extended ePipe to watch for changes happening in HopsFS and reflect those changes in the Hive metastore to ensure that there is no orphaned metadata for Hive if tables or databases are removed directly from HopsFS. In this scenario, the *Hive Handler* consumes the events directly from the *DB Watcher* and then it will take appropriate actions accordingly to synchronize the metastore metadata with the data files in HopsFS.

VI. Evaluation

In this section, we examine the throughput and latency of ePipe in replicating the HopsFS *frol* to Elasticsearch to implement the metadata search service. We also compare the latency and throughput of ePipe's metadata search service with the equivalent *inotify* and *find* services for HDFS. All the experiments were run on the SICS ICE cluster using Dell PowerEdge R730xd servers(Intel(R) Xeon(R) CPU E5-2620 v3 (2.40GHz, 256 GB RAM, 4 TB 7200 RPM HDD) connected using a single 10 GbE network adapter. In the experiments, we used a 6 node database cluster, NDB version 7.5.6, and the database replication degree was set to 2. We used NDB's default configuration for epoch handling where epochs are incremented atomically every 100 milliseconds. Elasticsearch version 2.4.1 was used on a single node setup. We used Trumpet 2.3.1 with Kafka 1.1.0 on a single node setup. In order to stress test ePipe with a high load of events, we ran a HopsFS cluster with 10 namenodes. We also ran, an equivalent highly available (HA) setup of HDFS with 5 servers (1 namenode, 1 Secondary namenode, and 3 Journal Nodes co-located with 3 Zookeeper nodes). For 10-namenode setups, HopsFS has three times the throughput of HDFS, thus, generating three times more events for ePipe to process than HDFS can produce at max load. We used the same benchmarking utility for HopsFS and HDFS as described in [2]. Also, we ran experiments based on a real-world Hadoop workload from Spotify, published in [2]. Finally, we present a real-world statistics for ePipe, and Elasticsearch from a production cluster, at SICS North AB (http://hops.site).

A. Overhead of frol extension to HopsFS

In this experiment, we ran the Spotify Hadoop workload for 60 seconds and recorded the throughput while varying the number of namenodes. The goal of the experiment is to measure the overhead of incorporating the *frol* extension on existing HopsFS operations. W ran the experiment for two setups: *frol* enabled and *frol* disabled. As shown in Figure 3, the *frol* extension has no overhead on the throughput of HopsFS. However, the Spotify workload is a read intensive workload where only 4.77% of the operations are mutating the namespace by running create, delete, rename, and move operations [2]. So to test the overhead of the *frol* extension even further, we ran another set of experiments with 100% create, 100% delete, and 100% rename. Figure 4 shows that the overhead is negligible at the start with fewer namenodes, but it increases gradually when more namenodes are added since the database nodes became more overloaded.

In real-world scenarios, industrial workloads are mostly read heavy where rename operations are rare and the combined

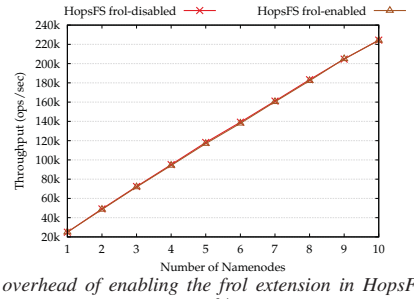


Fig. 3: The overhead of enabling the frol extension in HopsFS for a real-world Spotify workload in which 4.77% of the operations are mutating the namespace (create, delete, rename, and mv operations).

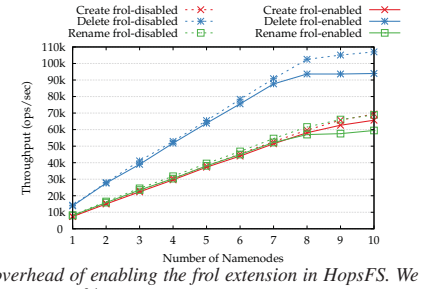


Fig. 4: The overhead of enabling the frol extension in HopsFS. We ran different experiments where 100% *of file system operations are logged as Create, Delete, or Rename operations.*

percentage of delete and create operations is 3.45% of all operations [2].

B. ePipe vs. HDFS INotify vs. Trumpet

HDFS INotify is a pull-based notification service, where clients keep polling the namenode for events. The client can control how many events to retrieve on each poll request. The default and recommended number of events is 1000. Increasing this number will overload the namenode and incur higher network traffic. Also, having many concurrent clients polling the namenode will overload it. Trumpet also provides a pull-based notification service for HDFS. Trumpet publishes the file system events into a Kafka topic, where clients can subscribe to that topic to consume its events. On the other hand, the *frol* extension is push-based where the NDB Events API pushes the events to the subscribers, ePipe in this case.

In this experiment, we used a 10-namenode HopsFS cluster and a highly-available HDFS cluster. We ran a 100% create microbenchmark on both clusters for 30 seconds, then we measured the average latency, and the throughput by which the events arrive at HDFS INotify client, Trumpet client, and ePipe while varying the load on HDFS and HopsFS by using 100, 1000, and 4000 concurrent clients. The major observable difference between both clusters is that HopsFS delivers $10X - 12X$ the throughput of HDFS. That is, ePipe is consuming $10X - 12X$ the events consumed by the HDFS INotify client and the Trumpet client. Figure 5 shows the throughput of ePipe, Trumpet, and HDFS INotify. In case of HDFS INotify we test three different configuration for the maximum number of requested events (1K, 10K, 100K). ePipe provides $10X - 56X$ the throughput in case of HDFS INotify, and $14X - 32X$ in case of Trumpet.

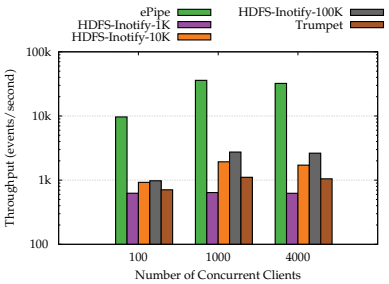


Fig. 5: Comparison of the throughput of ePipe/HopsFS, HDFS INotify, and Trumpet. HDFS INotify clients can set the maximum number of events to request from the namenode, so we compare against three different configurations for HDFS INotify (1K, 10K, and 100K). The default and recommended configuration for HDFS INotify is 1K. The Y-axis is in log scale with base 10.

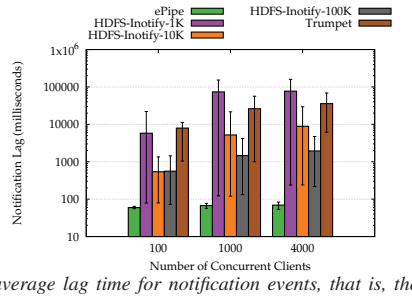


Fig. 6: The average lag time for notification events, that is, the time taken from creating/deleting an inode until it reaches HDFS INotify client, Trumpet client, or ePipe. The Y-axis is in log scale with base 10.

1) The notification lag (latency)

The notification lag is the time taken from inode creation/deletion until a notification event reaches HDFS INotify clients, Trumpet client, or ePipe. Figure 6 shows the comparison between the average lag time for the HDFS INotify client, Trumpet client, and ePipe. ePipe's lag time is almost constant ($66-70$ msec) independent of the number of concurrent clients, whereas HDFS INotify and Trumpet vary according to the number of concurrent clients and the batch size (in case of HDFS INotify). ePipe has $8X - 1113X$ lower lag time than HDFS INotify. The main difference in performance is due to the polling mechanism and the fact that In HDFS, the file system transactions are written first to a quorum of journal nodes, 3 in our setup, and then they are available for consumption by the inotify clients. However, for HopsFS the logging events are available to ePipe almost instantly through the push based distributed Event API. On the other hand, Trumpet has a higher lag time by design compared to HDFS INotify since it continuously polls the events from the edit log directory on the local file system. ePipe has $120X - 519X$ lower lag time than Trumpet.

In the case of ePipe, the lag time is mainly affected by how fast the epoch numbers are incremented in NDB. To understand that relationship, we ran an experiment with 100% create, and then we calculated the average lag time as reported by ePipe while varying the NDB epoch incremental interval. Figure 7 shows that the average lag time increases while increasing the interval by which NDB increment the epochs. Also, we noticed that the lag time is bounded by the interval between epochs. Based on Figure 7, we can decrease our lag time to 10 milliseconds, however, that means that all database nodes in the cluster will execute epoch increment protocols every 10 milliseconds, which would negatively affect the performance of the database and clients using it.

C. ePipe vs. HDFS find

In this experiment, we searched for file names that have 4000 duplicates. HDFS find performs badly since it traverses the whole namespace to search for a file. Elasticsearch is two orders of magnitude faster than HDFS find, see Table I. This experiment demonstrates the potential of polyglot storage of metadata, as full-text search operations would not be possible

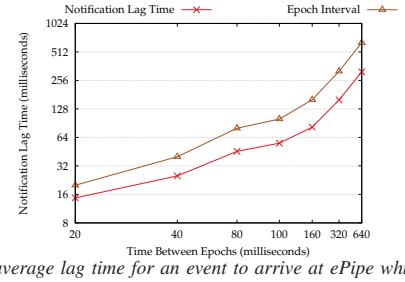


Fig. 7: The average lag time for an event to arrive at ePipe while varying the interval used by NDB to increment the epoch numbers, the default value is 100 milliseconds. The plot is in log-log scale with base 2.

on HopsFS' metadata database either. Full-text search is only enabled by correct replication of metadata to an external database that supports full-text search, such as Elasticsearch. As ePipe also indexes extended metadata for files/directories in Elasticsearch, the full-text search can also use user-defined 'tags' to find files, opening up new possibilities for data governance, archival, and data provenance.

Number of files/directories	HDFS-find (msec)	Elasticsearch (msec)
500K	26931	77
1M	55803	74
1.5M	64915	79

TABLE I: The time taken by HDFS-find and ePipe/Elasticsearch to execute a full-text query to search for a file.

D. ePipe performance

In this experiment, we used a 10 namenode HopsFS cluster, and we ran the Spotify workload for 60 seconds while varying the configuration of ePipe. First, we tested the effect of changing the *Elastic Handler* queue batch size, as shown in Figure 8a. We also tested two different batch sizes for the *Batcher* queue. For the experiment setup, an Elastic Handler batch size of $200K$ and Batcher batch size of 100 seems to be the best configuration where the average time taken to replicate events to Elastic search is 330 milliseconds. Figure 8b dissects the times to better understand how the Elastic Handler batch size affects the average time per event. There are 4 main components that make up the end-to-end time: 1) the *Epoch Barrier* which is the barrier that ePipe uses to collect the events for an epoch and reorder them before receiving new events from the new epoch, 2) the *Batcher queue* time, which is the time the event spends in the Batcher queue until either the batch size reached or the timer fired, 3) the *Data Enrichment* time, which is the time taken by ePipe to read any extended metadata or data in general as specified by the *Data Enricher*, and 4) the *Elastic Handler* time, which is the time taken to bulk insert the events into Elasticsearch including the queueing and indexing time. As shown in Figure 8b, most of the time is taken by events queuing in the Elastic Handler and then by Elasticsearch bulk insertion and indexing. There is a huge spike in time at batch size $100K$, because ePipe was consuming more events than the available capacity in the Elastic Handler. Also, for batches bigger than $200K$, the queuing time increases, due to the fact that the Elastic Handler queue is now bigger than the ePipe consumption rate, and events have to wait longer in the queue before indexing. Figure 8c shows the relative contributions for each of the aforementioned stages to the average time taken by an event in ePipe for a specific Elastic Handler batch size which is $200K$. The *Epoch Barrier* time is almost constant and depends on how fast NDB increments the epoch numbers, similar to the lag time as discussed in Section VI-B1. Figure 8d shows the total time and throughput of ePipe to consume all events produced by the Spotify workload for 60 seconds. At a batch size of $100K$, it took 79 seconds to consume all events, and that is due to the fact that the Elastic Handler was not able to keep up with the consumption rate. For the other batch sizes, they almost finished just a few seconds after the end of the experiment, and that is due to the fact that the last batch probably won't be complete and we rely on the Elastic Handler timer to push the last batch to Elasticsearch, configured at 10 seconds. For most configurations, ePipe manages to process the events at almost $11K$ events/second.

E. ePipe Recovery

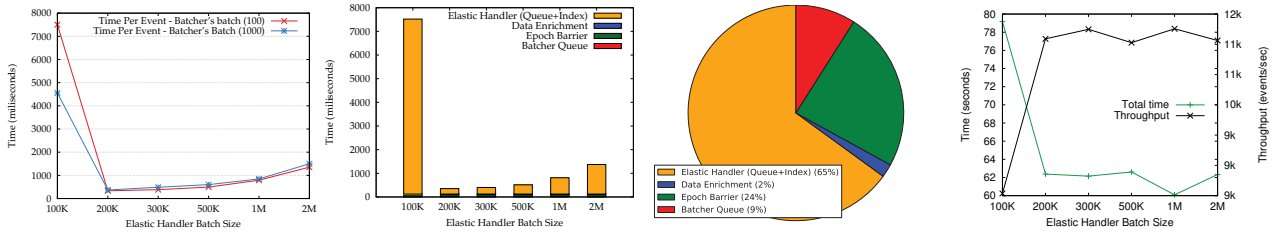
ePipe does not delete the *frol* entries until after it has been successfully delivered them to all downstream clients, ensuring failures will not result in data loss. Upon recovery, ePipe reads all the logs from the *frol* table in the database and reorders the events using the same logic as is used when events are received from the NDB Event API, thus ensuring consistent event re-ordering semantics. Table II shows the time taken by ePipe to recover from a failure while varying the number of failed events from $200K$ to $30M$ events. ePipe can recover $200K$ events in 6.86 seconds, and $30M$ in 768.7 seconds with a throughput ranging from $29K$ to $50K$ events/second.

Number of *frol* entries	Recovery Time (sec)	Throughput (events/sec)
200 K	6.86	29.72 K
500 K	11.28	45.19 K
1 M	29.12	34.36 K
1.5 M	34.94	42.1 K
2.5 M	51.46	50.4 K
10 M	263	38.2 K
30 M	768.7	39 K

TABLE II: The time taken by ePipe to recover from failures and the throughput while varying the number of events in the frol table.

F. Statistics from a Production Cluster

We have collected some statistics for ePipe and Elasticsearch from a Hops cluster at RISE SICS North AB that is administered by Logical Clocks AB. The cluster runs Hopsworks [19], where ePipe services including Hive metadata management and metadata search. As of April 2018, there are around 400 registered users, 523 projects, 2603 datasets out of which 144 datasets are shared between multiple projects, and 631 are searchable. Under the hood, HopsFS stores almost 47 million files and directories. Elasticsearch stores almost 10 million documents, where a document is a file/directory associated with its extended metadata. These documents consume almost $1GB$ of storage. The average indexing and querying times reported by Elasticsearch are 0.11 milliseconds and 18.29 milliseconds respectively. We have collected statistics from ePipe for several days of representative use, and the average reported replication latency taken per event is 139.82 milliseconds with a standard deviation of 80.3 milliseconds. The average time for each stage is as follows; 1) the *Epoch Barrier* takes 105.13 milliseconds,



(a) *(b)* *(c)* *(d)*

Fig. 8: a) shows the average time for two different configurations of the Batcher's batch size (100/1000), b) shows the relative contributions for each of the 4 stages (Epoch Barrier, Batcher Queue, Data Enrichment, and Elastic Handler), c) shows the relative contributions but only for a specific Elastic Handler batch size ($200K$), and d) shows the total experiment time and throughput for ePipe.

2) the *Batcher queue* takes no time since ePipe is running with Batcher queue of size 1, 3) the *Data Enrichment* takes 0.3 milliseconds, 4) the *Elastic Handler* takes 34.39 milliseconds which is very low since we are running on a batch size of 100 bytes. Figure 9 shows the percentiles for each of the stages and for the total time (replication lag). The time taken by *Elastic Handler* increases rapidly after the 90^{th} percentile due to the *Elastic Handler* timeout set at 500 milliseconds.

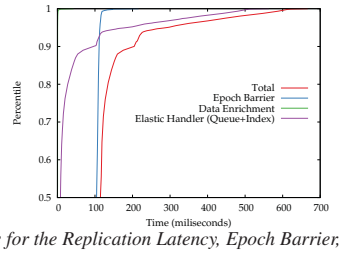


Fig. 9: Percentiles for the Replication Latency, Epoch Barrier, Data Enrichment, and Elastic Handler.

G. AutoIncrement Columns Overhead

In this microbenchmark, we evaluate the usage of AutoIncrement columns in NDB as a version number (logical clock) for rows in the *frol* table. However, since we are dealing with a distributed database where there is no global auto increment counter, each client generates a batch of auto increment IDs from a shared global counter, which will guarantee unique keys among the clients. A downside of this solution is that gaps may appear in the IDs, when clients do not use up all the IDs in their batch. In this experiment, we test the effect of using different batch sizes while varying the number of concurrent clients, see Figure 10. Setting the batch size to 1 will ensure a total order among the clients, with no gaps, but as can be seen, it severely impacts the performance of the whole application. The throughput at batch size 1 is $5K$ regardless of the number of concurrent clients. The throughput at batch size 10 reaches a plateau after 40 clients, while batch sizes 100 and 1000 seem to continuously increase throughput by increasing clients. The performance bottlenecks introduced by using an auto increment column with a batch size of 1 is the motivation we use for introducing a version number for each inode.

VII. Related Work

As file systems grow in size, the need for efficient real-time full-text search for files and directories becomes a much sought-

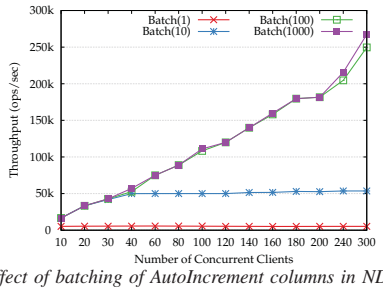


Fig. 10: The effect of batching of AutoIncrement columns in NDB with respect to the number of concurrent clients. The higher the batch size the higher the throughput, however, the auto increment IDs are not serializable across the clients.

after feature. There have been many studies to provide a file system with search capabilities [22], [23], [24], [25], [2], [26], [27]. The Semantic File System [23] is one of the early solutions for supporting file search within the file system by embedding search query results as a dynamically generated virtual directory in the hierarchical namespace. The Inversion File System [27] and WinFS [26] are examples of the first attempts to use a general purpose DBMS as the core file system metadata store, rather than the traditional inodes layout. Spyglass [24] took a structured approach to managing metadata, by exploiting the tree structure of the hierarchical namespace to partition the search space into fixed-size subtrees. Spyglass crawls a periodic snapshot of the metadata changes in order to update the search index. However, Spyglass doesn't support arbitrary user-defined attributes and a distributed index setup. Propeller [25] provides real-time file search in a distributed system by partitioning the namespace according to file access pattern "access causality". In order to infer the access causality, an Access Causality Graph (ACG) is constructed that adds extra overhead. Propeller supports user-defined attributes. Although it is distributed, it only considered multiple monolithic file systems rather than a distributed file system.

GOODS [28] is a system that transparently crawls different storage systems at Google to discover datasets and infer their metadata. GOODS supports user-defined tags, facet search based on collected attributes and user-defined tags, provenance of the datasets, and hooks when a specific attributes change in a dataset. GOODS focuses on structured datasets. Unlike GOODS, ePipe focuses on hierarchical distributed file systems instead of supporting ubiquitous data sources, and ePipe is a

databus, not a crawl-based indexer.

Wormhole [5] and Databus [6] are systems to tail the database transaction logs and provide a stream of events to their subscribers/fetchers. Wormhole provides a push-based event notification while Databus favours a pull-based solution. Both systems are used to replicate changes asynchronously to a set of downstream applications. ePipe follows a similar approach but instead of relying on polling the transaction logs, we use the NDB Event API which pushes the events from many database nodes directly to ePipe. Synapse [29] addresses the polyglot persistence problem by replicating attributes stored in the database for MVC based web applications across heterogeneous databases. On the other hand, ePipe offers enrichment capabilities for its events before replicating to the downstream applications.

VIII. Conclusions

In this paper, we introduced ePipe, a databus that generates a consistent change stream for HopsFS and eventually delivers the correctly ordered stream with low latency to downstream clients. We have shown that ePipe can be integrated into existing HopsFS clusters with minimal overhead on file system operations, and that ePipe can scale to replicate a Hadoop workload from Spotify to Elasticsearch with sub-second replication lag. We also showed that metadata search using Elasticsearch and ePipe is more than two orders of magnitude faster than HDFS-find, and delivers up to 56X the throughput of HDFS-INotify and Trumpet for thousands of concurrent clients. Finally, we showed the flexibility of ePipe as a general purpose databus for the HopsFS changelog, by showing how a variety of downstream applications can easily be integrated from metadata search to strongly consistent SQL-on-Hadoop (Hive-on-HopsFS).

Acknowledgment

This work was funded by the Swedish Foundation for Strategic Research projects “Smart Intra-body network, RIT15-0119” and “Continuous Deep Analytics, BD15-0006”, and by the EU Horizon 2020 project AEGIS under Grant Agreement no. 732189.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Mass Storage Systems and Technologies, 2010*, May 2010, pp. 1–10.

[2] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström, “Hopsfs: Scaling hierarchical file system metadata using newsql databases,” in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 89–104.

[3] “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation),” *Official Journal of the European Union*, vol. L119, pp. 1–88, May 2016.

[4] M. Stonebraker and U. Cetintemel, ““one size fits all”: an idea whose time has come and gone,” in *21st International Conference on Data Engineering (ICDE’05)*. IEEE, 2005, pp. 2–11.

[5] Y. Sharma, P. Ajoux, P. Ang, D. Callies, A. Choudhary, L. Demailly, T. Fersch, L. A. Guz, A. Kotulski, S. Kulkarni *et al.*, “Wormhole: reliable pub-sub to support geo-replicated internet services,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 351–366.

[6] S. Das, C. Botev, K. Surlaker, B. Ghosh, B. Varadarajan, S. Nagaraj, D. Zhang, L. Gao, J. Westerman, P. Ganti *et al.*, “All aboard the databus!: Linkedin’s scalable consistent change data capture platform,” in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 18.

[7] M. Ronström and J. Oreland, “Recovery principles of mysql cluster 5.1,” in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 1108–1115.

[8] “Hdfs inotify,” https://issues.apache.org/jira/browse/HDFS-6634, [Online; accessed 15-Aug-2017].

[9] “Trumpet,” http://verisign.github.io/trumpet/, [Online; accessed 20-April-2018].

[10] “Hdfs find,” https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html\#find, [Online; accessed 23-Nov-2015].

[11] M. Stonebraker, “Newsql: An alternative to nosql and old sql for new oltp apps,” *Communications of the ACM. Retrieved*, pp. 07–06, 2012.

[12] N. Shamgunov, “The memsql in-memory database system.” in *IMDM@ VLDB*, 2014.

[13] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, “Sap hana database: data management for modern business applications,” *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[14] M. Stonebraker and A. Weisberg, “The voltdb main memory dbms.” *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[15] A. Davies and H. Fisk, *MySQL Clustering*. MySQL Press, 2006.

[16] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] “Elasticsearch bulk api: Cheaper in bulk,” https://www.elastic.co/guide/en/elasticsearch/guide/current/bulk.html, [Online; accessed 15-Aug-2017].

[18] G. B. Salman Niazi, Mahmoud Ismail and J. Dowling, “Leader Election using NewSQL Systems,” in *Proc. of DAIS 2015*. Springer, 2015, pp. 158 –172.

[19] M. Ismail, E. Gebremeskel, T. Kakantousis, G. Berthou, and J. Dowling, “Hopsworks: Improving user experience and development on hadoop with scalable, strongly consistent metadata,” in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 2525–2528.

[20] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O’Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang, “Major technical advancements in apache hive,” in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1235–1246.

[21] F. Buso, “Sql on hops,” Master’s thesis, KTH, School of Information and Communication Technology (ICT), 2017.

[22] C. Johnson, K. Keeton, C. B. Morrey, C. A. Soules, A. Veitch, S. Bacon, O. Batuner, M. Condotta, H. Coutinho, P. J. Doyle *et al.*, “From research to practice: Experiences engineering a production metadata database for a scale out file system,” in *Proceedings of the 12th USENIX conference on File and Storage Technologies*. USENIX Association, 2014, pp. 191–198.

[23] D. K. Gifford, P. Jouvelot, M. A. Sheldon *et al.*, “Semantic file systems,” in *ACM SIGOPS Operating Systems Review*, vol. 25, no. 5. ACM, 1991, pp. 16–25.

[24] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, “Spyglass: Fast, scalable metadata search for large-scale storage systems.” in *FAST*, vol. 9, 2009, pp. 153–166.

[25] L. Xu, H. Jiang, L. Tian, and Z. Huang, “Propeller: A scalable real-time file-search service in distributed systems,” in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 2014, pp. 378–388.

[26] “Winfs: Windows future storage,” https://en.wikipedia.org/wiki/WinFS, [Online; accessed 23-Nov-2015].

[27] M. A. Olson and M. A, “The design and implementation of the inversion file system,” 1993.

[28] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang, “Goods: Organizing google’s datasets,” in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD ’16. New York, NY, USA: ACM, 2016, pp. 795–806.

[29] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh, “Synapse: A microservices architecture for heterogeneous-database web applications,” in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys ’15. New York, NY, USA: ACM, 2015, pp. 21:1–21:16.