From Research to Practice: Experiences Engineering a Production Metadata Database for a Scale Out File System

Charles Johnson, Kimberly Keeton, and Charles B. Morrey III, *HP Labs;* Craig A. N. Soules, *Natero;* Alistair Veitch, *Google;* Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer, Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro, *HP Storage*

https://www.usenix.org/conference/fast14/technical-sessions/presentation/johnson

This paper is included in the Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14).

February 17–20, 2014 • Santa Clara, CA USA

ISBN 978-1-931971-08-9

Open access to the Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14) is sponsored by

From research to practice: experiences engineering a production metadata database for a scale out file system

Charles Johnson[1], Kimberly Keeton[1], Charles B. Morrey III[1], Craig A. N. Soules[2], Alistair Veitch[3], Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer, Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro

HP Labs[1] Natero[2] Google[3] HP Storage

Abstract

HP's StoreAll with Express Query is a scalable commercial file archiving product that offers sophisticated file metadata management and search capabilities [3]. A new REST API enables fast, efficient searching to find all files that meet a given set of metadata criteria and the ability to tag files with custom metadata fields. The product brings together two significant systems: a scale out file system and a metadata database based on LazyBase [10]. In designing and building the combined product, we identified several real-world issues in using a pipelined database system in a distributed environment, and overcame several interesting design challenges that were not contemplated by the original research prototype. This paper highlights our experiences.

1 Introduction

Unstructured data, which accounts for more than 90% of the information in the world today [11], creates a number of challenges, including economically storing the data (even as it ages), effectively protecting and managing it, and extracting value from the stored data. To help customers tame their information explosion, HP wanted to provide an archival storage solution that would scale to billions of files and objects and create structure for unstructured data by allowing customers to exploit rich metadata services.

To help with the problem of extracting value, the solution would need to provide fast metadata search to support a variety of usage scenarios. For example, system administrators need to quickly and efficiently find files that match a given criteria to monitor storage operation (e.g., identify files created, modified, or deleted within a given time frame) and enforce compliance (e.g., determine which files are approaching retention expiration, or are on legal hold). Users want to "tag" files with custom metadata attributes and later search using those attributes. Such metadata services would also benefit external applications like backup and enterprise content management, by allowing them to avoid costly file system scans when determining which files have changed and must be backed up or indexed.

Ad hoc solutions in this space couple together an external relational DBMS and a scale out file store. This approach is unable to support the necessary scaling and performance requirements. Additionally, such solutions do not provide integrated search capabilities across system and custom metadata, and are likely to be expensive to maintain. Instead, our goal was to embed the metadata service within the file system to solve these challenges.

StoreAll with Express Query is a file archiving solution that couples a scale-out file system with an embedded database to accelerate metadata queries [3]. Initial releases target archival workloads, where files must be kept for an extended period of time, may be actively searched and may subject to business or regulatory requirements. In these systems, the number of files and aggregate data size can be extremely large, due to the need to retain files for many years.

This paper describes our experiences transforming a research metadata database (LazyBase [10]) into a production-quality metadata database, Express Query. In our work, we discovered several issues prompted by the scalable file archiving use case that we had not considered in the research prototype, and re-evaluated several of our original design decisions.

We begin by providing background on LazyBase and the scale out file system (§2). We highlight some of the challenges we encountered and overcame (§3), as well as the new capabilities we added to improve usability and flexibility (§4). Finally, we overview the related work (§5) and summarize the lessons we learned (§6).

2 Background

This section provides an overview of the original LazyBase [10] design and of the StoreAll file system architecture.

2.1 LazyBase

Express Query is based on LazyBase, a distributed database that provides scalable, high-throughput ingest of updates, while allowing a per-query tradeoff between latency and result freshness [10]. LazyBase provides this tradeoff using an architecture designed around batching and pipelining of updates. Read queries observe a stale, but consistent, version of the data, which is sufficient for many applications; more up-to-date results can be obtained when needed by scanning updates still being processed by later stages of the pipeline.

LazyBase provides a service model that decouples update processing from read-only queries. Updates (e.g., adds, modifies, deletes) are *observational*, meaning that data additions and modifications must provide new or updated values, which will overwrite (or delete) existing data. Because data is batched, uploaded (potentially) out-of-order and processed asynchronously, it may not be possible to read the "current" value of a field to determine the new/updated value; the most recent update may not have been uploaded yet or may still be being processed by the pipeline.

To improve database ingest performance, update clients (also known as *sources*) batch updates together and upload them to LazyBase as a single *self-consistent update (SCU)*, which is the granularity of transactional (e.g., ACID) properties throughout the update pipeline. For read-only queries, LazyBase provides snapshot isolation, where all reads in a query will see a consistent snapshot of the database, as of the time that the query started; in practice, this is the last SCU that was applied at query start time.

LazyBase tables contain an arbitrary number of named and typed columns. Each table has a *primary sort order* and one or more optional *secondary sort orders* (analogous to materialized views), which contain a subset of the columns and rows of the primary sort order. Each sort order is a collection of fixed-size pages, called *extents*, which are stored in compressed form. Additionally, each sort order has an *extent index*, which stores the minimum and maximum value of the key in each extent of the underlying sort order. Because extents are typically large (64KB), and the index only stores min and max values, the index is small enough to fit into memory, even if the table is very large. As a result, LazyBase requires fewer disk I/Os to locate a data extent through the extent index than would be required for a traditional B-tree index. Primary and secondary sort orders, as well as extent indexes, are stored as DataSeries files [9].

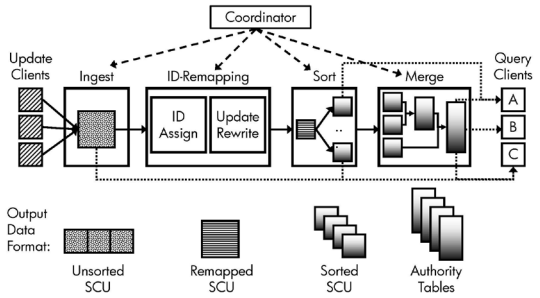

Figure 1: LazyBase prototype architecture [10].

Figure 1 illustrates LazyBase's update processing pipeline. The *ingest* stage accepts client uploads and makes them durable. The *ID-remapping stage* converts SCUs from using their internal temporary IDs to using global IDs common across the system. The *sort* stage sorts each of the SCU's tables for each of its sort orders. The *merge* stage combines multiple SCUs into a single sorted SCU. In addition to these stages, a coordinator tracks and schedules work in the system, maintaining availability and managing recovery.

2.2 StoreAll architecture

StoreAll's shared nothing clustered file system is subdivided into segments (volumes). Each segment contains a portion of the inodes (directories and files) in the file system. A segment is owned by one server, and the file system supports failover to other servers if the owning server fails. Each server handles reads and writes and manages locking for inodes in the segments it owns. A server can access an inode owned by another server in the cluster via internal network handshaking. The system supports NFS, CIFS, HTTP, FTP, and local file system access and scales to more than 16 PB of data in a single name space.

As the file system is updated, the system records metadata state changes (e.g., file creations, deletions, retention operations) into a per-segment *archive journal*. This journal is a transactionally reliable change log of file system metadata updates that each server maintains for the segments that it manages. Every few seconds the *archive journal writer* (`ajwriter`) flushes the archive journal files (`ajfiles`) for the segments owned by that

server; for each segment, the `ajwriter` closes the existing `ajfile` and starts a new one. Once the `ajfiles` are closed, they appear in the StoreAll namespace in a hidden directory and an update notification is sent to the subscribers of the `ajwriter`. This distributed publish/-subscribe event-driven architecture scales out well because changes are recorded locally and immediately. It avoids expensive file system scans for metadata changes and provides a difficult-to-bypass auditing mechanism.

3 Lessons Learned

Incorporating LazyBase into the StoreAll product pushed our initial LazyBase design in interesting new directions. In this section, we highlight several of the lessons learned, including the demands of the file system use case, the limits of our initial design, and how we addressed the challenges. We believe that these lessons and our solutions generalize to using a system like LazyBase in other distributed environments.

3.1 Transaction model complications

The combination of observational updates, out-of-order events and asynchronous processing complicates the transactional model. Here, we describe three aspects of the problem and our solutions: out-of-order event processing, expressing freshness, and enforcing data integrity.

3.1.1 Out-of-order event processing

Depending on the order in which batches are uploaded, events may be processed by the database in a different order than they were generated in the file system. LazyBase's pipeline has built-in support for processing out-of-order updates. It uses both per-field and per-row timestamps, and makes no assumptions about where the timestamps come from, only that the timestamps generated for updates to a particular field must be totally ordered. When merging multiple versions of a given row, LazyBase compares the timestamps of all versions of a field and takes the newest.

In the research prototype for LazyBase, we used the event timestamps in the input data as the field timestamps. We assumed that all updates for a given field could be globally ordered based on their timestamps. In the product, we had to cope with the fact that event timestamps associated with the same file system object could be generated by different servers with skewed clocks. The clock skew issue prompted changes in the way we track event timestamps for StoreAll.

In StoreAll, servers that host client connections are called *entry servers (ESs)*. ESs initiate file system operations on one or more file system objects on behalf of their clients. However, durable modifications caused by these operations are made only at the server that owns the file system object; such servers are called *destination servers (DSs)*. Any ES in the system can initiate an operation that results in durable modifications to a file system object. Operations that do not generate any durable modifications (e.g., `read` and `getattr`) can be supported via caching on the ES, without requiring communication with the DS that owns the object. As in all distributed systems, the clocks on the individual ES and DS nodes will have skew.

Ultimately, we eliminated the clock skew issue by using the DS timestamp for all events that make durable modifications to file system objects. We use the ES timestamp to support read auditing, with the proviso that these timestamps are not comparable to those in non-audit tables and using the knowledge that audit events are never updated after insertion.

3.1.2 Freshness

The LazyBase research prototype expressed freshness as a single number. In contrast, in a distributed system such as StoreAll, where multiple servers upload new data to Express Query, freshness can't be expressed as a single number. As described in § 3.2, updates are batched independently for different segments, meaning that it is not possible to provide a single point-in-time view of the entire file system's metadata. Instead, the freshness provided by Express Query is a range, delimited by the oldest and newest of the freshness levels from individual segments. Segment freshness levels are affected by a number of issues, including events being cached before being flushed to an `ajfile` (as described in §2.2), or a segment going offline for a time and only uploading events once it comes back online.

To simplify the early Express Query design, we disabled freshness queries. Even though database clients cannot request a particular freshness, they still need to know about the achieved freshness of their query results. For example, a periodic backup application that queries for recently updated files and wants to start its next backup where the previous one left off needs to know the freshness for the previous query results to avoid missing modified files. To address this need, Express Query explicitly tracks each segment's freshness, and query results include the minimum (*FreshnessComplete*) and maxi-

mum (*FreshnessPartial*) freshness values across the segments. FreshnessComplete indicates the timestamp before which all events have been observed from all segments. FreshnessPartial indicates the timestamp for the latest event processed for any segment. Thus, in the window between FreshnessComplete and FreshnessPartial, query results include some, but not all, of the events generated in the file system. Database clients can use this information to determine how to use the query results.

3.1.3 Enforcing data integrity

As described in § 2.1, the combination of observational updates, out-of-order event arrival and asynchronous processing means that LazyBase does not support read-modify-write transactions. This property has interesting implications for file system event processing. For example, custom attributes for an old version of a file should no longer be visible once the file has been deleted. However, since StoreAll users need to be able to add an arbitrary number of custom attributes for a file, so we organized the schema to store custom attributes in a different table (with one row per attribute) from the rest of the system attributes (with one row per file system object). This meant that file deletions couldn't automatically delete custom attributes, because there was no way to reliably read and delete the up-to-date set of custom attributes when processing the deletion event.

Instead, we needed to explicitly enforce integrity constraints between the tables. Express Query tracks file creation and deletion times, as well as timestamps for custom metadata operations, and queries must include timestamp comparison logic to check for attribute validity. A lazy cleaning pass periodically gets rid of custom attributes for deleted files as well as file lifetime information for files that were created or deleted sufficiently long ago.

3.2 Batching

As Cipar et al. observed, the choice of batch size causes a tradeoff between ingest throughput and latency [10]. Larger batches lead to greater pipeline processing efficiency (and hence better throughput), but also increase the delay before data can be queried – essentially, this decreases the freshness of the query results. We considered increasing batch size by including updates from multiple sources in the same batch, but quickly realized that this complicates the transactional model: it is more difficult for individual sources to abort, when the other sources in the same batch want to commit. As a result, we elected to create independent batches for different sources.

Express Query treats each file system segment as a source. A user-space tool called the *archive journal scanner*, or `ajscanner`, subscribes to the `ajwriter` notifications (§2.2). For each `ajfile`, the `ajscanner` parses the event data to create a batch of updates to upload to Express Query. The `ajscanner` processes `ajfiles` for each segment in order (determined using the `ajfiles`' `mtimes`), and uploads data from different segments in parallel. From Express Query's perspective, each segment appears as a separate source, uploading a stream of SCUs, one per `ajfile`. We use the fact that `ajfiles` are created regularly every few seconds to strike a balance between pipeline throughput and pipeline latency (freshness).

3.3 Auto-increment IDs

The LazyBase research prototype supported the concept of a 64-bit integer auto-increment ID column, also known as a database surrogate key [7]. IDs can more space-efficiently represent long values (e.g., file pathnames), by substituting the ID wherever the value would have been used in a table. The exact savings depends on a variety of factors, including the length of the strings, the strings' compressibility, and how many string fields are present in a table. The expectation was that by converting long values into integers, the ID-remapping mechanism would improve ingestion performance. Indeed, we found that using IDs sped up merge performance for a simulated file creation benchmark by an average of 54%. However, ID-remapping has both query and ingestion costs that must be considered.

The LazyBase prototype included IDs for a variety of string fields, including pathnames, and used these IDs as the primary key for most tables. Because LazyBase uses in-memory extent indexes to support point and range queries, sorting a table by the ID effectively randomized the data order, requiring a full table scan for what otherwise should be point or range queries. Furthermore, every query that selected or filtered on an ID-remapped attribute (in combination with other attributes) required a join with the ID table. In the file system context, this meant that all pathname-based queries (e.g., "find all files in a directory" or "show pathnames for all files modified in the last day") required a join between the path ID table and the table(s) containing the other metadata; often, these other tables required full table scans. In contrast, if IDs were not used and pathnames were included in the tables containing the other metadata attributes, with a sort order by pathname, path-based lookups could have been satisfied by an indexed lookup to the table(s) con-

Experiment	IDs (sec)	No IDs (sec)
File lookup	55.16 +/- 4.23	0.12 +/- 0.14
Directory lookup (small)	509.83 +/- 12.51	0.44 +/- 0.03
Directory lookup (med)	819.42 +/- 105.11	8.28 +/- 0.10

Table 1: ID vs. no-ID execution time (in seconds) for file and directory lookup queries, for 100M file dataset. Values shown are average +/- standard deviation for ten trials. The small directory lookup examines about 148k files; the medium directory lookup examines about 3.84M files. Directory lookups compute the max file size to eliminate output processing costs.

taining the other attributes. As shown in Table 1[1], the combination of full table scans and joins proved to be unacceptably inefficient.

The ingestion costs proved to be non-trivial, as well. The ID-remap stage must look up each incoming value to determine what global ID to assign, which requires all prior SCUs to be queryable and thus violates the goal of delayed processing for efficiency. Because the preceding individual SCUs may not have been merged into larger SCUs, remapping may require reading input data from many files, with the number of I/Os depending on the distribution of values in the input data. Additionally, the ID-remap stage proved to be a scalability bottleneck: since processing is serialized due to the need to look at all prior SCUs, the stage can only be scaled by partitioning the namespace. Although parallelizing ID-remap would help ingest-time scalability, it still would not address the query-time concerns described above.

Our solution was to eliminate the use of auto-incrementing IDs and the ID-remap stage entirely. This approach improved query performance dramatically and simplified many stages of the pipeline, including the coordinator job scheduling and recovery processing.

3.4 Primary key

Our initial Express Query design used pathname as the primary key for most tables, to transparently support backup/restore and remote replication, which preserve pathnames. This choice worked well for the archival use cases we initially targeted, where files were almost never modified after being created, and were not renamed. However, to support a more general file system use case, the system needed to provide support for renames and hard links. Unfortunately, with pathname as the primary key, this more general use case required reassigning the primary key, a costly operation. As a result,

[1]The equipment used for all experiments is an HP DL380p Gen8 server (2 x Intel Xeon E5-2697v2 CPUs, 2.70 GHz, 12 cores, 24 hyperthreads) with 384GB of DRAM. LazyBase/Express Query data is stored on an HP D2700 disk array with a P822 RAID controller and 25 146GB 15k RPM SAS drives.

	Primary sort order (sec)	Secondary sort order (sec)
Point key	129.08 +/- 4.17	0.05 +/- 0.01
Range (10%)	131.48 +/- 2.94	16.97 +/- 0.17
Range (25%)	136.44 +/- 2.60	39.68 +/- 0.34
Range (50%)	138.52 +/- 4.91	77.60 +/- 0.37
Range (75%)	142.02 +/- 3.67	115.80 +/- 1.00

Table 2: Execution time (in seconds) for point and range queries for primary sort order (table scan) vs. secondary sort order (index lookup), for 100M file dataset. Values shown are average +/- standard deviation for ten trials. The table shows range query results for four different selectivities (fraction of rows used to calculate result). Range queries compute a count to eliminate output processing costs.

the next version of our design chose as its primary key a globally unique file system-internal identifier for all file system objects. Tables continue to store the file system object's pathname and to define a secondary sort order based on the pathname, to avoid the auto-increment ID issues described in §3.3.

3.5 Secondary sort orders

As with any data management system, a universal challenge is how to organize the data to balance between query cost efficiency and data maintenance efficiency. In Express Query, this challenge amounts to which secondary sort orders to maintain, and how many columns each secondary sort order should contain.

For queries that filter on a secondary sort order's search key, the sort order provides efficient indexed lookups. Table 2 compares query execution time for indexed lookups vs. full table scans. If the secondary sort order is populated with a sufficiently large subset of the columns of the primary sort order, then a single secondary sort order can satisfy queries that access multiple attributes. For example, a query to select all pathnames, file sizes and file owners for files that have been recently modified could be efficiently satisfied by a secondary sort order that is sorted according to `mtime` and also contains the pathname, size and owner.

Creating and maintaining secondary sort orders during the update pipeline requires resources, however. The more secondary sort orders and the more columns per secondary sort order, the longer ingesting takes, and hence the freshness of the queryable data suffers. Table 3 quantifies the cost of update pipeline processing for additional fully-populated secondary sort orders.

To reap the potential query-time performance benefits from secondary sort orders, our initial Express Query schema maintained a fully-populated secondary sort order for each of the system attributes in the file objects table. We continue to experiment with reducing the num-

	Primary only	Primary + 15 secondary	Slowdown
Durable	1965 sec	6939 sec	3.53X
Queryable	2379 sec	11157 sec	4.69X

Table 3: Update pipeline processing time (in seconds) for ingesting 100M simulated file creations. "Durable" is time until the data is made durable (i.e., through the ingest pipeline stage). "Queryable" is time until the data is queryable (i.e., through the complete pipeline, including ingest). "Primary only" is a schema with no secondary sort orders for the file object data. "Primary + 15 secondary" is a schema with 15 fully-populated secondary sort orders, one per system attribute.

ber of secondary sort orders and the fraction of columns in various secondary sort orders, to improve ingest resource utilization and query freshness.

4 New Features

The goals for StoreAll's metadata database were to support user-initiated operations, such as assigning custom metadata tags to files, efficiently performing ad hoc file searches (e.g., a fast Unix `find`) and generating file system utilization reports. Additionally, the database needed to support external applications, such as a backup service tracking recently changed files. Finally, it needed to support internal file system operations, such as content validation scans and storage tiering policies. The query API needed to be flexible in the face of schema changes, and to facilitate rapid prototyping and experimentation by developers of the file system services using the database. The end user-visible interface needed to be intuitive and simple.

This section describes two APIs – SQL and REST – that we implemented to improve usability and flexibility for internal and external users of the database, respectively. The system continues to support programmatic queries where flexibility is not required, or performance overrides other considerations.

4.1 SQL API

We added a full SQL front end to Express Query, using the foreign data wrapper (FDW) API from PostgreSQL [5]. We define FDWs on top of the Express Query native tables, using the DataSeries (DS) storage layer and translation logic to access the tables. SQL queries are parsed, optimized, and partially executed by PostgreSQL, using foreign table accesses (table scans and index lookups) at the leaf nodes of the query execution tree, instead of native PostgreSQL table or index scans. Our approach uses multiple components: a Transaction Manager, a DS FDW, a DS row iterator, and a shim layer to translate between the FDW and row iterator. These components cooperate to request data from the Express Query pipeline workers, perform data translation operations, and implement transactional properties.

The *Transaction Manager* keeps track of active transactions and which versions of the Express Query tables they access, to ensure that all table accesses in the same transaction see a consistent view of the underlying database (i.e., per-transaction snapshot isolation). This mapping also informs garbage collection: the Transaction Manager prevents the garbage collector from reclaiming any versions that are still in use by an active transaction.

FDW. The FDW interfaces with the rest of PostgreSQL's query execution engine. It allows query qualifications (e.g., conditions in a SQL SELECT WHERE statement) to be passed to Express Query, to permit filtering of the rows examined to satisfy the query, rather than requiring a full table scan. Only qualifications with $=$, $<$, $<=$, $>$, $>=$, or LIKE operators on search keys are passed through, because they can be used by Express Query's index interface.

Translation shim. For each foreign table involved in a query, this layer communicates with the rest of Express Query to register the foreign table's transaction id with the Transaction Manager and learn which ingest pipeline worker(s) to contact to retrieve the data. The shim layer translates PostgreSQL's generic data types into Express Query data type-specific values, and prepares the DS search keys from the PostgreSQL qualifications. It uses these search key(s) to request data from the Express Query ingest worker(s) for the table.

DS row iterator. This layer applies the appropriate equality or range search key filters, and returns data from the Express Query pipeline worker one row at a time.

With this breakdown, the FDW needs no knowledge of Express Query, and Express Query needs no knowledge of PostgreSQL.

4.2 REST API

Although Express Query's SQL read query front end met the goal of enabling ad hoc queries, it did not isolate end users from the specifics of the database schema. To provide a simpler and more flexible interface, we defined a REST API [6], to permit users to request file and directory attributes, search for all paths matching a set of attribute criteria, and define custom attributes.

File-mode REST requests ("queries" in REST parlance) have three components: the path to be queried, the at-

tributes to be returned, and the query expression itself. In addition, several options specify recursive search, limitations on the number of results returned, and result order. The API supports both system and custom attributes. System attributes include the attributes stored in the file's inode (e.g., `size` and `mode`), as well as attributes particular to StoreAll's retention-enabled file system (e.g., storage tier, retention state). The API also provides attributes that summarize the last activity for a file (e.g., content modifications, custom metadata changes, file creations and deletions); we added these attributes to help database clients like backup providers efficiently discover what files had recent changes, to facilitate their own operations (e.g., choosing which files to back up). Users can also specify their own custom attributes, which are associated with paths as string key-value pairs.

We automatically translate each REST API query into a SQL query to retrieve the relevant metadata; results are presented in JSON.

5 Related work

Spyglass [12] provides an engine customized for file metadata indexing and querying. It leverages the property that files have many common attributes (e.g., owner and path prefix) to optimize index structures. Exploiting these properties achieves very high query performance, but sacrifices flexibility, in that the system does not support arbitrary user-specified attributes. Instead of constantly updating as the file system changes, Spyglass relies on efficient scans of periodic snapshots, which can result in highly variable freshness, depending on how often snapshots are taken. It also prevents the system from offering auditing capabilities, but enables a valuable feature in historical metadata search.

A number of systems (e.g., [4, 8, 1, 2, 13]) offer full file system search capabilities that can include metadata attributes. They typically rely on either some form of inverted index (fast for queries, but expensive to update and rebuild) or rely on a conventional RDBMS, which severely limits their scalability and performance properties. (Our early experiments with using both open-source and commercial RDBMSs for this purpose motivated the original LazyBase research.) By focusing on keyword search, these systems are somewhat orthogonal to our purposes, as they are not customized for metadata-intensive applications; many do not even index file metadata. Many of these systems also do not allow for custom metadata, rely on inefficient file system scans, or are not integrated into the kernel, and thus cannot offer auditing.

6 Conclusions

This paper highlights some of our experiences transforming a research prototype of a pipelined database into a production metadata database in a scale out file system. We summarize these experiences as follows:

Fallacies in our initial design. Despite our initial intuition, auto-incrementing IDs and ID-remapping provided unacceptable query and ingest performance slowdowns; therefore we removed them. We also realized that in a distributed environment, freshness is a window, not a single number; this complexity compelled us to disable freshness queries and report the achieved freshness range as part of query results.

Usability and flexibility sometimes override performance. Although our initial focus was on performance of the update pipeline and a fast programmatic query API, we learned that the flexibility to do ad hoc queries and rapid prototyping merited the inclusion of a SQL query API. Similarly, the desire to provide a simple interface that isolated users from schema changes prompted the development of a REST API.

Issues that we hadn't considered, motivated by our use case. LazyBase's lack of read-modify-write transactions meant that some data integrity constraints (e.g., custom attribute suppression for deleted files) needed to be explicitly enforced. Similarly, our initial choice of pathname as a primary key, while convenient for our initial archive use case, proved to be the wrong choice for a more general file system use case.

Modifications to the environment to ensure LazyBase assumptions hold. For example, we forced batches to contain only updates from a single source to ensure isolation between sources. Additionally, we forced timestamps on a particular field to have a total ordering, to ensure that LazyBase's out-of-order processing worked correctly.

Need to balance ingest-time and query-time processing. We observed tensions between ingest processing efficiency and query performance when selecting batch sizes and choosing which secondary sort orders to include in the schema. As in most data management systems, such design decisions must balance these competing demands.

7 Acknowledgments

We thank Jiri Schindler, our shepherd; Steven Hand; and the anonymous reviewers for constructive comments that have significantly improved the paper.

References

[1] Apache Solr. http://lucene.apache.org/solr/, Jan. 2014.

[2] Autonomy. http://www.autonomy.com/, Jan. 2014.

[3] HP StoreAll with Express Query. http://www.hp.com/go/storeall/, Jan. 2014.

[4] Introduction to Spotlight. https://developer.apple.com/library/mac/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.html, Jan. 2014.

[5] PostgreSQL. http://www.postgresql.org/, Jan. 2014.

[6] Representational state transfer. http://en.wikipedia.org/wiki/Representational_state_transfer, Jan. 2014.

[7] Surrogate key. http://en.wikipedia.org/wiki/Surrogate_key, Jan. 2014.

[8] Windows search. http://windows.microsoft.com/en-us/windows7/products/features/windows-search, Jan. 2014.

[9] ANDERSON, E., ARLITT, M., MORREY III, C. B., AND VEITCH, A. DataSeries: An efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review 43*, 1 (January 2009), 70–75.

[10] CIPAR, J., GANGER, G., KEETON, K., MORREY III, C. B., SOULES, C. A. N., AND VEITCH, A. LazyBase: Trading freshness for performance in a scalable database. In *Proc. of European Systems Conference (EuroSys)* (April 2012), pp. 169–182.

[11] GANTZ, J., AND REINSEL, D. Extracting value from chaos. *IDC report* (June 2011).

[12] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, scalable metatdata search for large-cale storage systems. In *Proc. 7th USENIX Conf. on File and Storage Technologies FAST* (2009), pp. 153–166.

[13] MANBER, U., AND WU, S. Glimpse: A tool to search through entire file systems. In *Proc. of the Winter 1994 USENIX Conference* (San Francisco, CA, 1994), pp. 23–32.