The Design and Implementation of FusionDB

Adam Retter
Evolved Binary
<adam@evolvedbinary.com>

Abstract

FusionDB is a new multi-model database system which was designed for the demands of the current Big Data age. FusionDB has a strong XML heritage, indeed one of its models is that of a Native XML store.

Whilst at the time of writing there are several Open Source and at least one commercial Native XML Database system available, we believe that FusionDB offers some unique properties and its multi-model foundation for storing heterogeneous types of data opens up new possibilities for cross-model queries.

This paper discusses FusionDB's raison d'être, issues that we had to overcome, and details its high-level design and architecture.

1. Introduction

FusionDB, or Project Granite as it was originally known, was conceived in the spring of 2014. Ultimately it was born out of a mix of competing emotions - frustration, excitement, and ambition. The frustration came from both, the perceived state of Open Source NXDs (Native XML databases) at that time, and commercial pressures of operating such systems reliably at scale. The overall conclusion being that they each had several critical issues that were not being addressed over a prolonged period of time, and that they were rapidly being surpassed on several fronts by their newer NoSQL document database cousins. The excitement came from witnessing an explosion of new NoSQL database options, including many document oriented database systems, which offered varying consistency and performance options, with a plethora of interesting features and query languages. Whilst the ambition came from believing that the community, i.e.: the users of NXD systems, deserved and often wanted a better option, and that if it needed building, then we both could and, more importantly, should build it!

In 2014, we had over 10 years experience with an Open Source NXD, eXist-db, both contributing to its development, and helping users deploy and use it at a variety of scales. Well aware of its strengths and weaknesses in comparison to other NXDs and non-XML database systems, we set out to identify in a more scientific manner the problems faced by the stakeholders; ourselves as the develop-

ers of the NXD, our existing users, and those users which we had not yet attracted due to either real or perceived problems and missing features.

In the remainder of this section we set out what we believe to be the most important issues as identified by eXist-db users and developers. Whilst we do make comparisons between BaseX, eXist-db, MarkLogic and other NoSQL databases, we place a particular focus on eXist-db as that is where our experience lies. It should be remembered that each of these products has varying strengths and weaknesses, and that all software has bugs. Whilst one might interpret our discussion of eXist-db issues as negative, we would rather frame it in a positive light of transparent discussion, yes there are issues, but if we are aware of them, then ultimately they can be fixed. There are many eXist-db users operating large sites who are able to cope with such issues, just as there are with BaseX, MarkLogic, and others.

1.1. Issues Identified by Users

First, through either direct reports from existing and potential users, or acting as a proxy whilst deploying solutions for users, we identified the following issues:

1. Stability

   Under normal operation the NXD could stop responding to requests. As developers, we identified two culprits here, 1) *deadlocks* caused by the overlapping schedules of concurrent operations requiring mutually exclusive access to shared resources, and 2) read and write contention upon resources in the system, whereby accessing certain resources could cause other concurrent operations to stall for unacceptably long periods of time.

2. Corruption

   When the system stopped responding to requests, it had to be forcefully restarted. Often this revealed or led to corruption of the database. As developers, we identified both a lack of the Consistency and Durability properties of ACID (Atomicity, Consistency, Isolation, and Durability) semantics, which also affected the ability to recover safely from a crash (or forceful restart).

3. Vertical Scalability

   Adding CPUs with more hardware threads did not cause the performance of the database to scale as expected. As developers, we identified many bottlenecks caused by contention upon shared resources.

4. Horizontal Scalability

   There was no clustering support for sharding large data sets or splitting resource intensive queries across multiple machines.

5. Operational Reliability

   There was no support for a mirrored system with automatic failover in the event of a hardware failure.

6. Key/Value Metadata

    There was no facility to associate additional key/value metadata with documents in the database. Many users would store the metadata externally to the document, either in a second "metadata document", or an external (typically) SQL database, later combining the two at query time. Unfortunately, this lacks atomic consistency, as updates to both the metadata and documents are not guaranteed to be in sync. As developers, we identified that a key/value model which is atomically consistent with a document model would be advantageous.

7. Performant Cross-reference Queries

    The system supported indexes for `xml:id` XML identifiers and references within and across documents. However, more complicated XML Markup Languages such as DocBook, TEI, and DITA, may use much more complex linking schemes between nodes where composite key addressing information is expressed within a single attribute. As developers, we identified that for queries spanning documents with such complex references, more advanced data models and index structures could be used to improve performance.

1.2. Issues Identified by Developers

Both through those issues raised by users, and our experience as developers, we identified a number of technical issues with eXist-db that we believed needed to be solved. Each of those technical issues fell into one of three categories - Correctness, Performance, and Missing Features.

It is perhaps worth explicitly stating that we have ordered these categories by the most important first. Correctness has a higher priority than Performance, and Performance has a higher priority than developing new features. There is little perceived benefit to a database system which is fast, but fast at giving the wrong results!

1.2.1. Correctness

1. Crash Recoverable

    If the database crashes, whether due to a forced stop, software error, or the host system losing power, it should always be possible to restore the database to a previously consistent state.

    A lack of correctness with respect to the implementation and interaction of the WAL (Write Ahead Log) and Crash Recovery process needs to be addressed.

2. Deadlock Avoidance

    In a number of scenarios, locks within the database on different types of resources are taken in different orders, this can lead to a predictable yet avoidable deadlock between two or more concurrent operations.

A lack of correctness with respect to the absolute order in which locks should be obtained and released by developers on shared resources needs to be addressed.

3. Deadlock Detection and Resolution

   With dynamic queries issued to the database by users on an ad-hoc basis, it is prohibitively expensive (or even impossible) to know all of the shared resources that will be involved and how access to them from concurrent queries should be scheduled. Deadlock Avoidance in such scenarios is impossible, as it requires a deterministic and consistently ordered locking schedule.

   With the ad-hoc locking of resources to ensure consistency between concurrent query operations, deadlocks cannot be prevented.

   To ensure the correct and continual functioning of the system, deadlocks must not cause the system to stop responding. If pessimistic concurrency with locking is further employed, a deadlock must be detected and resolved in some fashion which allows forward progress of the system as a whole. An alternative improvement would be the employment of a lock-free optimistic concurrency scheme.

4. Transaction Isolation

   With regards to concurrent transactions, Isolation (the *I* in ACID) [1], is one of the most important considerations of a database system and deeply affects the design of that system.

   Also equally important, is a clear statement to users about the available isolation levels provided by the database. Different user applications may require varying isolation levels, where some applications, such as social media, may be able to tolerate inconsistencies provided by weaker isolation levels, accounting or financial applications often require the strongest isolation levels to ensure consistency between concurrent operations.

   For example, BaseX provides clear documentation of how concurrent operations are scheduled [10]. It unfortunately does not explicitly state its isolation level, but we can infer from its scheduling that it likely provides the strictest level - *Serializable*. Likewise, MarkLogic dedicates an entire chapter of their documentation [14], to the subject, a comprehensive document that unexpectedly does not explicitly state the isolation level, but likely causes one to infer that *Snapshot Isolation* is employed; a further MarkLogic blog post appears to confirm this [15].

   The exact isolation level of eXist-db is unknown to its users and likely also its developers. Originally eXist-db allowed *dirty-reads* [2], therefore providing the weakest level of transaction isolation - *Read-Uncommitted*. Several past attempts [3][5][6][7] have been made at extending the lock lease to the transaction boundary, which would ensure the stronger *Read-Committed* or *Repeatable-Read* level. Unfortunately, those past attempts were incomplete, so we can only

infer that eXist-db provides at least *Read-Uncommitted* semantics, but for some operations may offer a stronger level of isolation.

At least one ANSI (American National Standards Institute) ACID transaction isolation level must be consistently supported, with a clear documented statement of what it is and how it functions.

5. Transaction Atomicity and Consistency

   A transaction must complete by either, *committing* or *aborting*. Committing implies that all operations within the transaction were applied to the database as though they were one unit (i.e. atomically), and then become visible to subsequent transactions. Aborting implies that no operation within the transaction was applied to the database, and no change surfaces to subsequent transactions.

   Unfortunately, the transaction mechanism in eXist-db is not atomic. If an error occurs during a transaction, it will be the case that any write operation prior to the error will have modified the database, and any write operation subsequent to the error will not have modified the database, the write operations are not undone when the transaction aborts!

   There is no guarantee that a transaction which aborts in eXist-db will leave the database in a logically consistent state. Unless a catastrophic failure happens (e.g. hardware failure), it is still possible although unlikely, for an aborted transaction to leave the database in a physically inconsistent state. Recovery from such physically inconsistent states is attempted at restart by the database Recovery Manager.

   Transactions must be both Atomic and Consistent. It should not be left as a surprise for users running complex queries, that if their query raises an error and aborts, to discover that some of their documents have been modified whilst others have not.

1.2.2. Performance

1. Reducing Contention

   Alongside the shared resources of Documents and Collections that the users of NXDs are concerned with, internally there are also many data structures that need to be safely shared between concurrent operations.

   For example, concurrent access to a Database Collection in eXist-db is effectively mutually exclusive, meaning that only a single thread, regardless as to whether it is a reader or writer, may access it. Likewise, the same applies to the paged storage files that eXist-db keeps on disk.

   To improve vertical scaling we therefore need to increase concurrent access to resources. The current liberal deployment of algorithms utilising coarse-grained locking and mutual exclusion need to be replaced, either with single-

writer/multi-reader locking, or algorithms that use a finer-grained level of locking, or where possible, non-blocking lock-free algorithms.

2. System Maintenance

   There is occasionally the need to perform maintenance tasks against the database system, such as creating backups or reindexing documents within the database.

   Often such tasks require a consistent view of the database, and so acquire exclusive access to many resources, which limits (or even removes) the ability of other concurrent queries and operations to run or complete until such maintenance tasks finish.

   For example, in eXist-db there are two backup mechanisms. The first is a best effort approach which will run concurrently with other transactions, but does not guarantee a consistent snapshot of the database. The second will wait for all other transactions to complete, and then will block any other transaction from starting until it has completed, this provides a consistent snapshot of the database, but at the cost of the database being unavailable whilst the backup is created. Another example is that of re-indexing, which blocks any other Collection operation, and therefore any other query transaction.

   Such database operations should not cause the database to become unavailable. Instead, a *version snapshot* mechanism should be developed, whereby a snapshot that provides a point-in-time consistent view of the database can be obtained cheaply. Such maintenance tasks could then be performed against a suitable snapshot.

1.2.3. Missing Features

1. Multi-Model

   The requirement from users for Document Metadata informs us that we also need the ability to store *key/value model* data alongside our documents.

   Conversely, the requirement from users for more complex linking between documents appears to us to be well suited to a *graph model*. Such a graph model, if it were available, could be used as a query index of the connections between document nodes.

   If we disregard *mixed-content*, then JSON's rising popularity, likely driven by JavaScript and the creation of Web APIs [16], places it now heavily in-demand. Many modern NoSQL document databases offer JSON (JavaScript Object Notation) [11][12][13] document storage. Undoubtedly an additional JSON *document model* would be valuable.

   Neither eXist-db nor BaseX have multi-model support, although both have some limited support for querying external relational-models via SQL (Structured Query Language), and JSON documents in XQuery [17][18]. Berkeley

DB XML offers atomic access to both key/value and XML document models [19]. MarkLogic offers both graph and document models natively [20].

It is desirable to support key/value, graph, and alternative document models such as JSON, to compliment our existing XML document model.

2. Clustering

   With the advent of relatively cheap off-the-shelf commodity servers, and now to a more extreme extent, Cloud Computing, when storage and query requirements dictate it should be possible to distribute the database across a cluster of machines.

   Should any machine fail within the cluster, the entire database should still remain available for both read and write transactions (albeit likely with reduced performance). Should the size of the database reach the ceiling of the storage available within the cluster, it should be possible to add more machines to the cluster to increase the storage space available to the cluster. Likewise, if the machines in the cluster are saturated by servicing transactions on the database, adding further machines should enable more concurrent transactions.

   Ideally, we want to achieve a shared-nothing cluster, where both data and queries are automatically distributed to nodes within the cluster, thus achieving a system with no single point of failure.

2. Design Decisions

Due to both our technical expertise and deep knowledge of eXist-db, and our commercial relationships with organisations using eXist-db, rather than developing an entirely new database system from scratch, or adopting and enhancing another Open Source database, we decided to start by forking the eXist-db codebase.

Whilst we initially adopted the eXist-db codebase, as we progressed in the development of FusionDB we constantly reevaluated our efforts against three high-level objectives, in order of importance:

1. Does a particular subsystem provide the best possible solution?
2. We must replace any inherited code whether from eXist-db or elsewhere that we cannot trust and or/verify to operate correctly.
3. We would like to maintain eXist-db API compatibility where possible.

2.1. Storage Engine

The storage engine resides at the absolute core of any database system. For disk based databases, the task of the storage engine is to manage the low-level reading and writing of data from persistent disk to and from memory. Reads from disk

occur when a query needs to access pages from the in-memory buffer pool that are not yet in memory, writes occur when pages from the in-memory buffer pool need to be flushed to disk to ensure durability.

eXist-db has its own low level storage engine, which combines the usually segregated responsibilities of managing in-memory and on-disk operations. eXist-db's engine provides a B+ tree with a paged disk file format. Originally inherited from dbXML's B+ tree implementation in 2001 [9][8], and is still recognisable as such, although to improve performance and durability it has received significant modifications, including improved in-memory page caching and support for database logging.

After an in-depth audit of the complex code forming eXist-db's BTree and associated Cache classes which make up its storage engine, we concluded that we could not easily reason about its correctness, a lack of unit tests in this area further dampened confidence. In addition, due to the write-through structure of its page cache, we identified that concurrent operations on a single B+ tree were impossible and that exclusive locking is required for the duration of either a read or write operation.

Without confidence in the storage engine of eXist-db, we faced a fundamental choice:

- Write the missing unit tests for eXist-db's storage engine so that we may assert correct behaviour, hopefully without uncovering new previously unknown issues. Then re-engineer the storage engine to improve performance for concurrent operations, add new tests for concurrent operation, and assert that it still passes all tests for correctness.

- Develop a new storage engine which offers performant and concurrent operation, with clean and documented code. A comprehensive test suite would also need to be developed which proves the correctness of the storage engine under both single-threaded and concurrent operation.

- Go shopping for a new storage engine! With the recent explosion of Open Source NoSQL databases, it seems a reasonable assumption that we might be able to find an existing well-tested and trusted storage engine that could be adapted and reused.

2.1.1. Why we opted not to improve eXist-db's

As eXist-db's storage engine is predominantly based on a dated B+ tree implementation, and not well tested, we felt that investing engineering effort in improving this would likely only yield a moderate improvement of the status quo. Instead, we really wanted to see a giant leap in both performance and foundational capabilities for building new features and services.

Considering that within the last year at least one issue was discovered and fixed that caused a database corruption related to how data was structured within a B+ tree [4], it seemed likely that further issues could also surface.

Likewise, whilst the B+ tree is still fundamental and relevant for database research, hardware has significantly advanced and now provides CPUs with multiple hardware threads, huge main memories, and faster disk IO in the form of SSDs (Solid State Disks). Exploiting the potential performance of modern hardware requires sympathetic algorithms, and recent research has delivered newer data structures derived from B-Trees. Newer examples include the B^{link} tree [21] which removes read locks to improve concurrent read throughput, Lock-Free B +Tree [22] which removes locks entirely to reduce contention and improve scalability under concurrent operation, Buffer Trees [24] and Fractal Trees [25] which perform larger sequential writes to improve linear IO performance by coalescing updates, and the Bw-Tree [23] which both eschews locks to improve concurrent scalability and utilises log structuring to improve IO.

Given these concerns and access to newer research, we elected not to improve eXist-db's current storage engine.

2.1.2. Why we opted not to build our own

Whilst we possess the technical ability, the amount of engineering effort in producing a new storage engine should not be understated.

Of utmost importance, when producing a new storage engine is ensuring correctness, i.e. that one does not lose or corrupt data. Given that the storage engine of eXist-db evolved over many years and may still have *correctness* issues, and that a search quickly reveals that many other databases also had correctness issues with their storage engines, that in some cases took years to surface and fix [26] [27] [28] [29], we have elected not to develop a new storage engine.

In line with our organisations philosophy of both, not re-inventing a worse wheel, and gaining from contributing to open source projects as part of a larger community, we believe our engineering resources are best spent elsewhere by building upon a solid and proven core, to further deliver the larger database system features which are of interest to the end users and developers.

2.1.3. How and why we chose a 3rd-party

Having decided to use an existing storage engine to replace eXist-db's, we initially started by looking for a suitable Open Source B+ Tree (or derivative) implementation written in Java. We wanted to remain with a 100% Java ecosystem if possible to ease integration. We had several requirements to meet:

- Correctness

  We must be able to either explicitly verify the correctness of the storage engine, or have a high degree of confidence in its correct behaviour.

- Performance

  The new storage engine should provide the same or better single-threaded performance than eXist-db's B+ tree. Although we were willing to sacrifice some single-threaded performance for improved concurrent scalability with multi-threading.

- Scalability

  As previously discussed in Section 2.1, eXist-db's B+ tree only allows one thread to either read or write, and due to this it cannot scale under concurrent access. The new storage engine should scale with the number of available hardware threads.

Initially, we studied the storage engines of several other Open Source databases written in Java that use B+ Trees, namely: Apache Derby, H2, HSQLDB, and Neo4j. Whilst each had a comprehensive and well tested B+ Tree implementation backed by persistent storage, there were several barriers to reuse:

- Tight Integration - each of the B+ tree implementations were tightly integrated with the other neighbouring components of their respective database.
- No clean interface - there was no way to easily just reuse the B+ tree implementation without adopting other conventions, e.g. configuration of the hosting database system.
- Surface area - to reuse an existing B+ Tree would have meant a trade-off, where we either: 1) add the entire 3rd-party database core Jar as a dependency to our project, allowing for easy updates but also adding significant amounts of unneeded code, or 2) we copy and paste code into our project which makes keeping our copy of the B+ Tree code up-to-date with respect to upstream updates or improvements a very manual and error prone task.

Succinctly put, these other database systems had likely never considered that another project might want to reuse their core storage engine, and so were not designed with that manner of componentisation in mind.

A second route that we considered, was looking for a very lean standalone storage engine implemented in Java that could be reused within our database system. We identified MapDB as a potential option, but soon discounted it due to a large number of open issues around performance and scalability [30].

Having failed to find an existing suitable Java Open Source option for a storage engine, we needed to broaden our search. We started by examining the latest research papers on fast key/value database systems, and reasoning that databases are often considered as infrastructure software and therefore more often than not written in C or C++, we removed the requirement that it must be implemented in Java. We also broadened our scope from B+ tree like storage, to instead requiring that whatever option we identified must provide excellent performance when

dealing with the large number of keys and values that make up our XML documents, including for both random access and ordered range scans.

From many new possibilities, we identified three potential candidates that could serve as our storage engine. Each of the candidates that we identified were considered to be mature products with large userbases, and stable due to being both open source and having large successful software companies involved in their development and deployment.

- LMDB (Lightning Memory-Mapped Database Manager)

  LMDB offers a B-Tree persistent storage engine written in C. It was originally designed as the database engine for OpenLDAP.

  LMDB provides full ACID semantics with *Serializable* transaction isolation. Through a Copy-on-Write mechanism for the B-Tree pages and MVCC (Multi-Version Concurrency Control), it provides read/write concurrency and claims excellent performance; readers can't block writers, and writers can't block readers, however only one concurrent write transaction is supported. One distinguishing aspect is that the code-base of LMDB is very small at just 6KLOC, potentially making it easy to understand. The small codebase is achieved by relying on the memory mapped file facilities of the underlying operating system, although this can potentially be a disadvantage as well if multiple processes are competing for resources.

  There is no built-in support for storing heterogeneous groups of homogeneous keys, often known as *Columns* or *Tables*. LMDB also strongly advises against long running transactions, such read transactions can block space reclamation, whilst write transactions block any other writes. Another consideration, although of less concern, is that LMDB is released under the OpenLDAP license. This is similar to the BSD license, and arguably an open source license in good faith, however in the stricter definition of "Open Source" the license does not comply with the OSI's (Open Source Initiative) OSD (Open Source Definition) [31].

- ForestDB

  ForestDB offers a novel HB+-Trie (Hierarchical B+-Tree based Trie) persistent storage engine written in C++11 [32]. It was designed as a replacement for CouchBase's CouchStore storage engine. The purpose of the HB+-Trie is to improve upon the B+Tree based CouchStore's ability to efficiently store and query variable-length keys.

  ForestDB provides ACID semantics with a choice of either *Read Uncommitted* or *Read Committed* transaction isolation. Through an MVCC and append only design, it supports both multiple readers and multiple writers, readers can't block writers, and writers can't block readers, however because synchronization between multiple writers is required it is recommended to only use a single writer. The MVCC approach also supports database snapshots, this

could likely be exploited to provide stronger *Snapshot Isolation* transactions and online database backups.

ForestDB through its design intrinsically copes well with heterogeneous keys, however if further grouping of homogeneous keys is required it also supports multiple distinct *KV store* across the same files. Unlike LMDB, ForestDB because of its append only design requires a compaction process to run intermittently depending on write load, such a process has to be carefully managed to avoid blocking write access to the database. ForestDB is available under the Apache 2.0 license which meets the OSI's OSD, however one not insignificant concern is that ForestDB seems to have a very small team of contributors with little diversity in the organisations that are contributing or deploying it.

- RocksDB

  Initially, we identified LevelDB from Google but at that time development appeared to have stalled, we then moved our attention to RocksDB from Facebook which was forked from LevelDB to further enhance its performance and feature set. Facebook's initial enhancements included multi-threaded compaction to increase IO and reduce write stalls, dedicated flush threads, universal style compaction, prefix scanning via Bloom Filters, and Merge (read-modify-write) operators [36]. RocksDB offers an LSM-tree persistent storage engine written in C++14. The purported advantage of the LSM tree is that writes are always sequential, both when appending to the log and when compacting files. Sequential writes, especially when batched, offer performance advantages over the random write patterns which occur with B+Trees. The trade-off is that reads upon an LSM tree require a level of indirection as the index of more than one tree may need to be accessed, although Bloom filters can significantly reduce the required IO [33] [34].

  RocksDB provides only some of the ACID properties, Durability and Atomicity (via Write Batches). Write batches act as a semi-transparent layer above the key value storage that enable you to stage many updates in memory, reads first access the write batch and then fall-through to the key value store, giving *read-your-own-writes* and therefore a primitive for building isolation with at least *Read Committed* strength [35]. Unfortunately, the developer has to manually build Consistency and Isolation into their application via appropriate synchronisation[1]. Through MVCC and append only design, it supports both multiple readers and multiple writers. Like ForestDB, the MVCC approach also supports database snapshots, which likewise could be

[1]At the time that RocksDB was adopted for FusionDB's storage engine, RocksDB offered no transactional mechanisms and therefore we developed our own. RocksDB now offers both pessimistic and optimistic transactional mechanisms

exploited to enable a developer to provide *Snapshot Isolation* strength transactions and online database backups.

RocksDB offers *Column Families* for working with heterogeneous groups of homogeneous keys, each Column Family has its own unique configuration, in-memory and on-disk structures, the only commonality is a shared WAL which is used to globally sequence updates. Each Column Family may be individually configured for specific memory and/or IO performance, which offers the developer a great deal of flexibility in tuning the database for specific datasets and workloads. In addition, due to the global sequencing, updates can be applied atomically across column families, allowing a developer to write complex read and update operations across heterogeneous sets of data.

Like ForestDB, RocksDB runs a background compaction process to merge storage files together, this could potentially lead to write stalls, although the multi-threaded approach of RocksDB eases this, it still has to be carefully managed. One concern is that RocksDB is available under the BSD-3 clause license with an additional patent grant condition, this is a complex issue, whilst the BSD license certainly meets the OSI's OSD, the patent grant is proprietary and does not.[2].

From the three potential candidates, each of which could technically work out very well, we ultimately selected RocksDB.

We quickly discounted ForestDB because of three concerns:

- Its' almost sole adopter seems to be CouchBase. This appears to be confirmed by its small GitHub community of contributors and users.
- We felt that we did not need its main advantage, of excellent performance for variable length keys. eXist-db (ignoring extended indexes) uses seven different key types for the data it needs to store, however within each key type, the key length variation is usually just a few bytes.
- There was no existing Java language binding, unlike LMDB's lmdbjava and RocksDB's RocksJava.

The choice between LMDB and RocksDB was not an easy one. Initially, we experimented by replacing just eXist-db's persistent DOM store with RocksDB, we added a layer of storage engine abstraction between eXist-db and RocksDB so that we could more easily replace RocksDB with LMDB or another storage engine should RocksDB not meet out requirements.

The reasons that we chose RocksDB over LMDB or any other storage engine were not just technical. RocksDB is the result of a team of highly skilled engineers, having first been built at Google and then advanced by Facebook (and oth-

[2]RocksDB was later relicenced by Facebook as a dual-licensed Open Source project, available under either GPL 2.0 or Apache 2.0.

ers), the storage engine is involved in almost every interaction that a user makes with the Facebook websites (facebook.com, instagram.com, and messenger.com), as of the third quarter of 2018, Facebook had 2.27 billion monthly active users. From this, we can likely assume that RocksDB has been thoroughly battle tested in production and at scale. Whilst LMDB is used in a handful of important infrastructure projects such as - OpenLDAP, Postfix, Monero, libpaxos, and PowerDNS [37], RocksDB is used by some of the largest web companies including - AirBnb, LinkedIn, NetFlix, Uber, and Yahoo [38]. LMDB seems to be predominantly developed by Symas Corporation, whereas RocksDB whilst led by Facebook has a larger number of diverse contributors. RocksDB also appears to have a much more rapid pace of development, with new features and bugfixes appearing frequently, in contrast LMDB appears to have stabilised with little new development.

Technically, RocksDB offered a much richer feature set that any other option we evaluated, a number of which we felt would be well suited for our use case. Many of the keys used within a key type by eXist-db are very uniform with common prefixes, this is highlighted when storing large XML documents with deeply nested levels, as the DLN (Dynamic Level Numbering) node identifier encoding that it employs will produce long strings with common prefixes. RocksDB offers both *Prefix Compression* and a *Prefix Seek API*, the compression ensures that common prefixes are only stored once [39] which reduces disk space and therefore IO, whilst the prefix seek builds bloom filters of key prefixes to enable faster random lookup and scan for keys with common prefixes [40]. eXist-db provides an update facility which is similar to XQuery Update, but where the updates are applied immediately to the data, in eXist-db these updates are not isolated to the transaction, RocksDB provides a *WriteBatchWithIndex* feature, which allows you to stage updates in-memory to the database, reading from this batch allows you to *read-your-own-writes* without yet applying them to the database, we recognised that this would allow us to provide stronger isolation for both eXist-db's XQuery Update equivalent and its various other updating functions. There are also many other RocksDB features that we are making use of such as atomic commit across column families, which we would otherwise have had to build ourselves atop another storage engine such as LMDB.

Since our initial prototyping with RocksDB in 2014 and subsequent wholesale adoption in 2015, it has since also been adopted as the primary storage engine by a number of other NoSQL databases with varying data models - Apache Kafka (event stream), ArangoDB (document, graph, and key/value), CockroachDB (tabular i.e. SQL), DGraph (graph), QuasarDB (time-series), SSDB (key/value), and TiKV (key/value). RocksDB storage engines have also been developed as replacements for the native storage engines of Apache Casandra (wide column store), MongoDB (document i.e. JSON), and MySQL (tabular i.e. SQL) [38] [41].

2.2. ACID Transactions

Having decided that FusionDB must support full ACID semantics, as well as the technical implementation we must also have a clear statement of our transaction isolation level. This enables developers building their applications with FusionDB to understand how concurrent transactions on the database will interact with each other, what constraints this places on the type of applications that FusionDB is best suited for, or whether they need to add any additional synchronisation within their application to support stronger isolation semantics.

Historically, there were four levels of transaction isolation, as defined by ANSI, each of which is expressed in terms of possible phenomena *dirty-read*, *fuzzy-read*, and *phantom*, that may occur when concurrent transactions operate at that level. From weakest to strongest these are: *Read Uncommitted*, *Read Committed*, *Repeatable Read*, and *Serializable*. Often users expect or even desire the strongest level, Serializable, but as this can require a great deal of synchronization between concurrent transactions, it can have a serious performance impact, and few database systems offer this option, and typically even then not as the default. To work around the dichotomy of needing to provide Serializable semantics *and* excellent performance, additional transaction levels have been developed in recent years. These are not as strong as Serializable, but exhibit less, or other, potentially acceptable phenomena, these include - *Cursor Stability*, *Snapshot Isolation*, and *Serializable Snapshot Isolation* [1] [42].

Table 1. Isolation Types Characterized by Possible Anomalies Allowed

Isolation level	**P0 Dirty Write**	**P0 Dirty Read**	**P4C Cursor Lost Update**	**P4 Lost Update**	**P2 Fuzzy Read**	**P3 Phantom**	**A5A Read Skew**	**A5A Write Skew**
Read Uncommitted	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
Read Committed	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
Repeatable Read	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible

Isolation level	P0 Dirty Write	P0 Dirty Read	P4C Cursor Lost Update	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	A5A Read Skew	A5A Write Skew
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL Serializable	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Reproduced from [1] - "Table 4. Isolation Types Characterized by Possible Anomalies Allowed."

As discussed in Section 1.2.1, the ACID semantics of eXist-db whilst acceptable for the projects that suit it, are much weaker than we require for FusionDB. eXist-db provides no user controllable transactions, internally it provides a `Txn` object that is often required when writing to the database, however this object in reality just manages transaction demarcation for its database log. In eXist-db, writes are immediate, and visible to all other transactions, likewise aborting a `Txn` does not cause previous statements to be rolled-back. The commit and abort mechanisms of `Txn` strictly exist for the purposes of attempting to return the database to a consistent state during crash recovery by replaying the database log.

- Atomicity

  With eXist-db, the atomicity is provided internally though multi-reader/single-writer locks for Documents, and exclusive locks for everything else including Collections and B+ tree page files.
- Consistency

  Unfortunately, as there are no true transactions in eXist-db, there is no real mechanism for taking the database from one consistent state to the next.
- Isolation

  The effective Isolation level in eXist-db is approximately *Read Uncommitted*.
- Durability

  eXist-db does take serious strides to ensure durability. It provides both a WAL which is *fsynced* to disk, and a synchronization task which periodically flushes dirty in-memory pages to persistent disk.

Fortunately for us, RocksDB provides the Durability and Atomicity properties of ACID. For durability, data is written to both in-memory and a WAL upon commit, in-memory updates are later batched and flushed to disk. Through switching to RocksDB for its storage engine, FusionDB also utilises RocksDB's WAL instead

of eXist-db's, thus discarding eXist-db's previously error prone crash recovery behaviour. This results in greater confidence of recovery from those system crashes that are beyond the control of our software. For Atomicity, RocksDB provides *Write Batches* where batches of updates may be staged and then applied atomically, in this manner all updates succeed together or fail. However, RocksDB leaves us to build the level of Isolation and Consistency that we wish to gain ourselves.

2.2.1. Transactions for FusionDB

Utilising a number of features provided by RocksDB we were able to build a level of isolation for FusionDB which is at least as strong as *Snapshot Isolation*. Firstly, we repurposed eXist-db's `TransactionManager` and associated `Txn` object to provide real user controllable database transactions. Secondly, due to its previous nature, whilst the `Txn` object was often required for write operations in eXist-db, it was rarely required for read operations, this meant modifying a great deal of eXist-db's internal APIs so that a `Txn` is always required when reading or writing the database.

Internally in FusionDB, when a transaction is begun, we make use of RocksDB's MVCC capability and create a Snapshot of the database. These Snapshots are cheap to create, and scale well up to hundreds of thousands, at which point they may significantly slow-down flushes and compactions. However, we think it unlikely that we would need to support hundreds of thousands of concurrent transactions on a single instance. Each snapshot provides an immutable point-in-time view of the database. A reference to the snapshot is placed into the `Txn` object, and is used for every read and scan operation upon the database by that transaction. When only considering reads, this by itself is enough to provide `Snapshot Isolation`, we can then, in fact, remove all of the database read locks that eXist-db used. As the Snapshot is immutable, we no longer need the read locks as no concurrent transaction can modify it.

To provide the isolation for write operations, when a transaction is begun we also create a new Write Batch and hold a reference to it in the `Txn`. This Write Batch sits above the Snapshot, and in actuality all reads and writes of the transaction go to the Write Batch. When the transaction writes to the database, it is actually writing to the Write Batch. The Write Batch stages all of the writes in order in memory, the database is not modified at this point. When the transaction reads from the database, it actually reads from the Write Batch. The Write Batch first attempts to answer the read for a key from any staged updates, if there is no update staged for the key, it falls through to reading from the Snapshot. As each transaction has its own Write Batch, which is entirely in memory, any writes made before committing the transaction are only visible within the same transaction. At commit time, the Write Batch of the (transaction) (`Txn`) is written atomi-

cally to the database, or if the transaction is aborted the memory is simply discarded, as no changes were made there are no changes to roll-back/undo. Whether committed or aborted, when the transaction is complete we release the database snapshot. The semantics are stronger than *Read Uncommitted* because no other transaction can read another transaction's Write Batch, stronger than both *Read Committed* and *Repeatable Read* because every read in the transaction is repeatable due to the snapshot, but also different than *Repeatable Read* as it could exhibit *write skew*. Like reads, for writes, we find that combining a Write Batch with a Snapshot enables us to maintain *Snapshot Isolation* semantics. In FusionDB, we could remove the Write Locks taken by eXist-db upon the database, but we have not yet done so, the write locks are only held for short durations rather than the duration of the transaction, however we are considering providing a configuration option which would extend them to the transaction lifetime, thus yielding *Serializable* isolation.

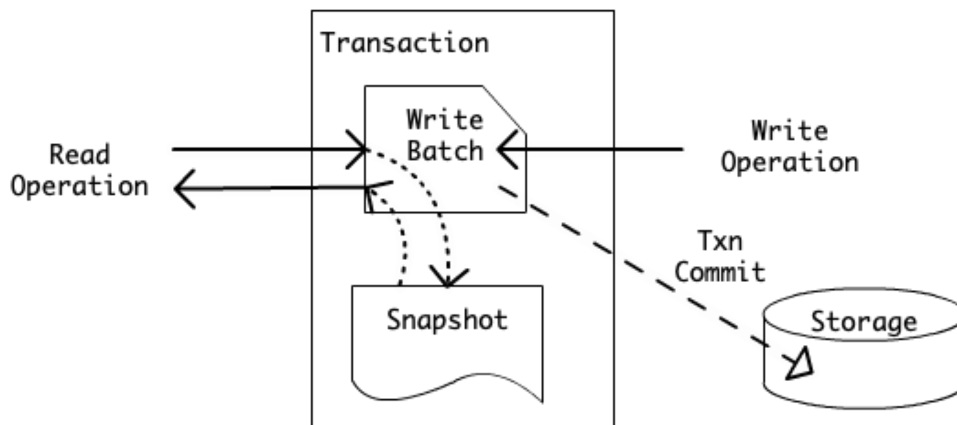

Figure 1. FusionDB Transaction Architecture

2.2.2. FusionDB Transactions and XQuery

At present FusionDB reuses eXist-db's XQuery engine but enforces that an XQuery is executed as a single transaction. This means that an XQuery will either commit all updates atomically or abort, this is in contrast to eXist-db, where there is no atomicity for an XQuery. Consider the XQuery listed in Figure 2, when this query is run in eXist-db if an error occurs during the `update insert` statement, the `/db/test.xml` document will have already been inserted into the database but will be missing its update, and so will have an incorrect balance! In FusionDB because a transaction maintains isolation and commits or aborts atomically, if an error occurs anywhere within the query, the document would never be inserted into the database.

```
1
2 import module namespace xmldb = "http://exist-db.org/xquery/xmldb";
```

```
  3
  4 let $uri := xmldb:store(
  5          "/db",
  6          "some-account.xml,
  7          <account currency="gbp" id="223344"><balance>0</balance></
account>)
  8 return
  9     update insert <balance>9.99</data> into doc($uri)/test
 10
```

Figure 2. Simple Compound Update XQuery

Whilst an XQuery does execute as a single transaction, FusionDB also provides a mechanism akin to *sub-transactions*. Sub-transactions are exposed to the XQuery developer naturally via XQuery *try/catch* expressions. The body of each try clause is executed as a sub-transaction, i.e. an atomic unit, if any expression within the *try body* raises an error, then all expressions within the try body are atomically aborted and the catch clause is invoked, otherwise all expressions in the try body will have been correctly executed. These sub-transactions permit the XQuery developer a fine level of control over their query. Consider the XQuery listed in Figure 3, with FusionDB if an error happens with any of the operations inside try body clause, the sub-transaction is atomically aborted, and so no documents are moved, the task log file then records the failure. With eXist-db due to a lack of atomicity, it an error occurs whilst moving one of the documents, it is entirely possible that some documents could have already been moved even though the query recorded the failure in the task log, thus meaning that the database is no longer logically consistent.

```
  1
  2 import module namespace xmldb = "http://exist-db.org/xquery/xmldb";
  3
  4 let $archived-uris :=
  5     try {
  6         for $record in collection("/db")/record[date lt
xs:date("2001-01-01")]
  7         let $uri := document-uri(root($record))
  8         let $filename := replace($uri, ".*/(.+)", "$1")
  9         return
 10             (
 11                 xmldb:move("/db", "/db/archive", $filename),
 12                 update insert
 13                         <entry>Archived {$uri}</entry>
 14                             into doc("/db/archive-log.xml")/log
 15             )
```

```
16     } catch * {
17         <archive-failure>{$err:code}</archive-failure>
18     }
19 return
20     xmldb:store(
21             "/db",
22             "task-log-" || current-dateTime() || ".xml",
23             <task id="123">{$archived-uris}</task>)
24
```

Figure 3. XQuery with Try/Catch Sub-transaction

XQuery allows try/catch expressions to be nested within the try or catch clause of another try/catch expression, likewise FusionDB supports nesting sub-transactions within sub-transactions. By using the standard try/catch recovery facilities of XQuery, we believe that FusionDB naturally does what the user expects with regards to transaction boundaries.

2.2.3. FusionDB Transactions and APIs

FusionDB strives to maintain API compatibility with eXist-db, and as such FusionDB provides the following eXist-db compatible APIs: REST, RESTXQ, WebDAV, XML-RPC and XML:DB. Apart from the XML:DB API, none of the other eXist-db APIs expose any mechanisms for controlling transactions, and regardless eXist-db does not implement the XML:DB API Transaction Service. FusionDB treats each call to any of these APIs as a distinct transaction, to maintain compatibility with eXist-db there are no mechanisms to establish a transaction across multiple API calls. When executing XQuery via these APIs, the use of transactions and sub-transactions as described in Section 2.2.2 apply. In future, it is likely that FusionDB will provide new APIs to allow external applications greater transactional control.

2.3. Concurrency and Locking

As we previously identified, eXist-db had a number of classes of concurrency problems. We needed to study and understand each of these to ensure that they would not also become issues for the components of eXist-db inherited by FusionDB. As part of our company philosophy of giving back to the larger Open Source community, we recognised that we should fix these problems at their source. As such, we undertook a code-audit project whereby we identified and fixed many concurrency and locking issues directly in eXist-db, we subsequently backported these fixes to the fork of eXist-db that we use for FusionDB. This backporting also enabled us to further increase our confidence in our changes by showing that not only did our code pass the new tests that we had created, but

that the modified version of eXist-db could pass the existing full-test suites of both eXist-db and FusionDB.

The changes we made to the locking and concurrency mechanisms in eXist-db were numerous and far-reaching. We have recently published two comprehensive technical reports detailing the problems and the solutions that we implemented [43] [44]. These comprehensive technical improvements have been incorporated into both eXist-db 5.0.0 and FusionDB. Instead of reproducing those technical reports within this paper, we will briefly highlight the improvements that were made and their impact for FusionDB.

2.3.1. Incorrect Locking

We identified and addressed many issues in the eXist-db codebase that were the result of incorrect locking, these included:

- Inconsistent Locking, whereby locking was applied differently to the same types of objects at varying places throughout the codebase. We documented the correct locking pattern, and made sure that it was applied consistently. This gave us improved deadlock avoidance of different types of locks which must interleave, e.g. Collection and Document locks, which are now always acquired and released in the same order.
- Insufficient/Overzealous Locking, whereby in some code paths objects were accessed either without locks or with more locking than is required for a particular operation. We modified many code paths to ensure that the correct amount of locking is used.
- Incorrect Lock Modes, where shared reader/writer locks were used, we repaired some cases whereby the wrong lock mode was used. For example, where a read lock was taken but a write was performed, or vice-versa.
- Lock Leaks and Accidental Release, whereby a lock is never released, is released too soon, or too often for reentrant locks. We introduced the concept of Managed Locks, and deployed them throughout the codebase. Our Managed Locks make use of Java's *try-with-resources* expression, to ensure that they are always released, this is done automatically and in line with the developer's expectations.

2.3.2. Lock Implementations

We identified that alongside standard Java Lock implementations, eXist-db also made use of two proprietary lock implementations. Whilst no issues with the lock implementations had been directly reported, we questioned the likely correctness of them, theorising that they could be a contributor to other reported problems with database corruption. As part of our need to understand them we were able to research and develop an incomplete provenance for them.

1. Collection Locks. eXist-db's own `ReentrantReadWriteLock` class was used for its Collection Locks. It was originally copied from Doug Lea's `ReentrantLock`, which was itself superseded by J2SE 5.0's locks. The eXist-db version has received several modifications which make it appear like a multi-reader/single-writer lock, and its naming is misleading, as in actuality it is still a mutually exclusive lock.

   Document Locks. eXist-db's own `MultiReadReentrantLock` class was used for its Document Locks. It was likely copied from the Apache Turbine JCS project which is now defunct. Strangely, this is a multi-reader/single-writer lock, which also appears to support lock upgrading. However, lock upgrading is a well-known anti-pattern which is typically prohibited by lock implementations. The eXist-db version has received several changes which were only simply described as "bug-fixes".

Ultimately, we felt that the custom Lock Implementations in use by eXist-db were of questionable pedigree and correctness. We replaced them with implementations that we believe to be both correct and reliable. We switched Document Locks to Java's standard `ReentrantReadWriteLock`. Whilst for Collection Locks we switched to `MultiLock` [45] from Imperial College London, which is itself based on Java's standard locking primitives. MultiLock is a multi-state intentioned multi-reader/single-writer lock, thus allowing for concurrent operations on Collection objects.

2.3.3. Asymmetrical Locking

Previously, the correct locking pattern in eXist-db when performing read operations on Documents within a Collection, was to lock the Collection for read access, retrieve the Document(s) from the Collection, lock the Documents for read access, perform the operations on the documents, release the Document locks and then finally release the Collection locks.

We were able to optimise this pattern to reduce the duration that Collection Locks are held for. Our asymmetrical pattern allows the Collection lock to be released earlier, after all the Document locks have been acquired, thus reducing contention and improving concurrent throughput.

2.3.4. Hierarchical Locking

Locks in eXist-db were previously in a flat space with one Lock per-Collection or Document object. Unfortunately, this meant that user-submitted concurrently executing queries could acquire locks for write access in differing orders which would could cause a deadlock between concurrent threads. A deadlock in eXist-db is unresolvable without restarting the system, at which point the Recovery Manager has to attempt to bring the database back to a logically consistent state.

We created a modified version of Gray's hierarchical locking scheme [46] for Collection Locks, whereby the path components of eXist-db's Collection URIs represent the hierarchy. The hand-over locking through this hierarchy that we implemented, coupled with intention locks provided by MultiLock, permit multi-reader/single-writer Collection access, but make it impossible to deadlock between concurrent Collection lock operations. We also added a configuration option (disabled by default) that allows for multi-reader/multi-writer locking of Collections, but its use requires careful application design by the XQuery developer.

In eXist-db, we have not as yet extended the hierarchical locking scheme to cover Documents, and so it is still possible to deadlock between Collection and Document access, if user-submitted queries have different lock schedule ordering for the same resources.

In FusionDB, less locking is required due to our MVCC based snapshot isolation, however at present such deadlocks are still possible for write operations. However, it is possible to resolve a deadlock by aborting a transaction in FusionDB, leaving the database in a consistent and functioning state.

2.3.5. Concurrent Collection Caching

eXist-db made use of a global Collection Cache to reduce both object creation overhead and disk I/O. Unfortunately, this cache was a point of contention for concurrent operation, as accessing it required acquiring a global mutually exclusive lock over the cache. For eXist-db we replaced this Collection Cache with a concurrent data structure called Caffeine which allows fine-grained concurrent access without explicit locking. For FusionDB, we need such global shared structures to be version aware so that they integrate with our MVCC model. We are working on replacing this shared cache with something similar to Caffeine but also supports MVCC.

2.3.6. New Locking Features

Alongside many technical fixes that we have made to eXist-db, we have also added three new substantial features:

1. A centralised Lock Manager, whereby all locking actions are defined consistently in a single class, and regardless of the underlying lock implementation they present the same API.

2. A Lock Table which is fed by the Lock Manager, and allows the state of all locks in the database system to be observed in real-time. It also provides facilities for tracing and debugging lock leases, and makes its state available via JMX for integration with 3rd-party monitoring systems.

3. A set of annotations named `EnsureLocked` which can be added to methods in the code base. These annotations form a contract which describes the locks that should be held when the method is invoked. When these annotations are enabled for debugging purposes, they can consult the lock table and eagerly report on violations of the locking contracts.

Example 1. Example Use of Locking Annotations

```
private Collection doCopyCollection(final Txn transaction,
    final DocumentTrigger documentTrigger,
    @EnsureLocked(mode=LockMode.READ_LOCK) final Collection
sourceCollection,
    @EnsureLocked(mode=LockMode.WRITE_LOCK) final Collection
destinationParentCollection,
    @EnsureLocked(mode=LockMode.WRITE_LOCK, type=LockType.COLLECTION)
final XmldbURI destinationCollectionUri,
    final boolean copyCollectionMode,
    final PreserveType preserve) {
```

2.4. UUIDs

In FusionDB every Collection and Document is assigned an immutable and persistent UUID (Universally Unique Identifier). Each UUID is a 128-bit identifier, adhering to the *UUID Version 1* specification as defined by IETF (Internet Engineering Task Force) RFC (Request For Comments) 4122. Instead of using the hosts MAC address, instead as permitted by the UUID specification, we use a random multicast address, which we generate and persist the first time a database host is started. This allows us per-host identification for the UUIDs within a multi-host system, but without leaking information about any hosts network settings.

These UUIDs allow the user to refer to a Collection or Document across systems, and retrieve it by its identifier regardless of its location with the database. When a Collection or Document is moved within the database, its UUID remains unchanged, whilst a copy of a Collection or Document will be allocated a new UUID. UUIDs are also preserved across backup and restore. When restoring a backup, the backup will notify the user of any conflicts between Collections and Documents in the database and the backup that have the same UUID but different database locations.

2.5. Key/Value Metadata

FusionDB offers a key/value metadata store for use with Collections and Documents. Any Collection or Document may have arbitrary metadata in the form of

key/value pairs which are transparently stored alongside it. FusionDB also provides range indexing and search for Collections and Documents based on both the keys and values of their associated Metadata. For example, the user can formulate queries like, "Return me all of the documents which have the metadata keys `town` and `country`, with the respective metadata values, `Wiggaton` and `United Kingdom`. Within FusionDB updates across are atomic and consistent across data-models, so for example, it is impossible for a Document and the Key/Value Metadata associated with that document to be inconsistent with respect to each other even under concurrent updates. Although not yet exposed via XQuery, internally there are also iterator based APIs for efficiently scanning over Collections and Documents metadata.

At present our Key/Value store, is just one of the small ways in which we expose the multi-model potential of FusionDB.

2.6. Online Backup

For backing up a FusionDB database we are able to make use of RocksDB's MVCC facilities to create snapshots and checkpoints. We provide two mechanisms for backing up the database, 1) a Full Document Export, and 2) a Checkpoint Backup. The Checkpoint Backup process is both light-weight and fast, and is the preferred backup mechanism for FusionDB. The Full Document Export process exists only for when the user needs a dump of the database for use with 3rd party systems such as eXist-db.

2.6.1. Full Document Export

The Full Document Export mechanism will be familiar to eXist-db users as it's very similar to the primary backup format for eXist-db, and is in fact compatible. The export results in a directory or zip file, which contains a complete copy of every document from the Collections that the user chose to backup. The output directory or zip file is also structured with directories representing each sub-Collection that is exported. Each directory in the output contains a metadata file named `__contents__.xml` which contains the database metadata for the Collection and its Documents.

The metadata files have been extended from eXist-db for FusionDB to also contain the key/value metadata (see Section 2.5) that a user may have associated with Collection or Documents, and the UUID (see Section 2.4) of each Collection and Document. We should also explicitly state that this Backup is transactional, and as transactions in FusionDB are snapshots of the database, the export has the advantage of being point-in-time consistent unlike in eXist-db. In addition, due to removing read-locks in FusionDB thanks to our snapshot isolation, unlike eXist-db the backup process will not block access for reads or writes to the database.

2.6.2. Checkpoint Backup

Checkpoint backups are provided by the underlying RocksDB storage engine and are near-instantaneous. They exploit both the MVCC and append-only nature of the storage engine's database files. When a checkpoint is initiated, a destination directory is created and if the destination is on the same filesystem then the files making up the current state of the live database will be hard-linked into it, otherwise they are copied over to it. Once the checkpoint has completed, a backup process operates directly from the checkpoint to provide a version of the database files that are suitable for archiving. The first part of this process is relatively simple, and really provides just the RocksDB database files holding the data for the backup and some metadata describing the backup. The second part of the backup process makes sure to copy any binary documents present in the database to the backup directory as well, to do this it starts a second read-only database process from the checkpoint, scans the binary documents in the database, making of copy of the filesystem blob of each one. When initiating a Checkpoint Backup, the user can choose either a full or incremental backup. As the backup process operates from a checkpoint, it cannot block concurrent operations accessing the database. The file format of the Checkpoint Backup is proprietary to FusionDB and should be treated as a black-box.

2.7. BLOB Store

Similarly to our work on Locking (see Section 2.3), we recognised that there were a number of problems with the handling of Binary Documents (i.e. non-XML documents), that we had inherited with eXist-db. We again decided to address these by contributing back to the wider Open Source community, as such we developed a new BLOB (Binary Large Object) store component[3] for use within eXist-db. This new BLOB Store is available in FusionDB, and a Pull Request that we prepared will likely be merged into eXist-db in the near future.

Although we have recently reported on the technical detail of the new BLOB Store [47], it is relevant to highlight two key points of its design:

- Lock Free. The entire Blob Store operates without any locks whatsoever, instead it operates as an atomic state machine by exploiting CAS (Compare-And-Swap) CPU instructions. This design decision was taken to try and increase concurrent performance for Binary Document operations.

  Deduplication. The Blob Store only stores each unique Binary Document once. When a copy of a Document is made, a reference counter is incremented, and conversely when a copy is deleted the reference counter is decremen-

[3]RocksDB has recently developed an extension called BlobDB, which is still highly experimental. Like our BLOB Store it also stores Binary Document file content in a series of files on the filesystem, but as far as we are aware does not yet offer deduplication.

ted. The Binary Document's content is only deleted once the reference count reaches zero. This design decision was made to try and reduce disk IO in systems which make use of many Binary Documents.

Importantly with regards to FusionDB, the new BLOB Store was designed not to reuse eXist-db's B+Tree storage engine, but instead persists a simple Hash Table. In FusionDB we have also replaced the BLOB Store's persistent Hash Table with a RocksDB Column Family, which means that updates to Binary Document objects are also transactionally consistent.

3. High-level Architecture

In this section, we detail several of the high-level architectural aspects of FusionDB.

3.1. Programming Language

FusionDB is written in a combination of both Java 8 and C++14 and is available for Windows, macOS, and Linux on x86, x86_64, and PPC64LE CPUs. This came about predominantly because eXist-db 5.0.0 is written in Java 8, whilst RocksDB is written in C++14 and RocksJava (the RocksDB Java API) is written in Java 7. Evolved Binary have been contributors to the RocksJava API since the adoption of RocksDB for FusionDB's storage engine. The RocksJava API is comprehensive, but typically lags behind the RocksDB C++ API, and so Evolved Binary have make many Open Source contributions to add missing features and fix issues.

The RocksJava API makes heavy use of Java's JNI (Java Native Interface) version 1.2 for calling C++ methods in RocksDB. Unfortunately when calling C++ code from Java (or vice-versa) via JNI, there is a performance penalty each time the Java/C++ boundary is traversed. After benchmarking different call mechanisms across the JNI boundary [48], we were able to improve the situation somewhat. However, when callbacks from C++ to Java involve millions of calls, perhaps because of iterating over large datasets, the cost of the JNI boundary accumulates quickly and has a large impact on performance. To avoid such a penalty and ensure good performance, several components of FusionDB which communicate directly and frequently with RocksDB were rewritten in C++. These components are then compiled directly into a custom RocksDB library that is used by FusionDB.

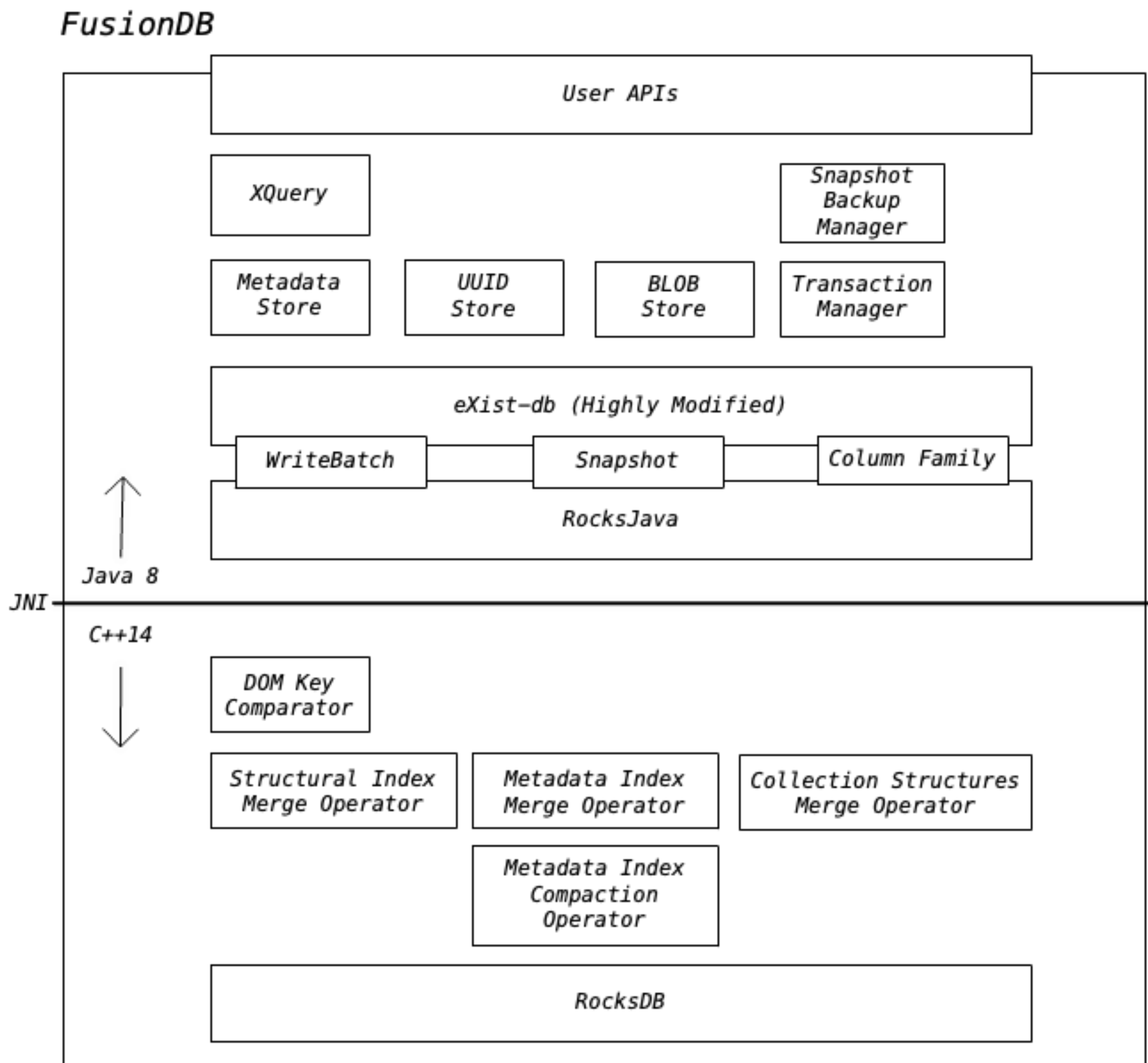

Figure 4. FusionDB - Components by Language

3.2. Column Families

As discussed in Section 2.1.3, RocksDB provides a feature called *Column Families*. These Column Families allow you to separate the key/value pairs that make up the database into arbitrary groupings. By storing homogeneous groups of keys and values into distinct Column Families, we can configure each Column Family in a manner that best reflects the format of those keys and values, and the likely access patterns to them. Modifications across Column Families remain atomic.

When we replaced eXist-db's B+Tree based storage engine with RocksDB, we carefully ported each distinct sub-system in eXist-db that used a B+Tree to one or more RocksDB Column Families. We also replaced eXist-db's non-B+Tree based Symbol Table and its indexes with a newly designed highly concurrent Symbol

Table which is persisted to several Column Families. In addition the new sub-systems that we added for Key/Value Metadata (see Section 2.5) and UUID locators (see Section 2.4) also make use of Column Families for storing data and indexes.

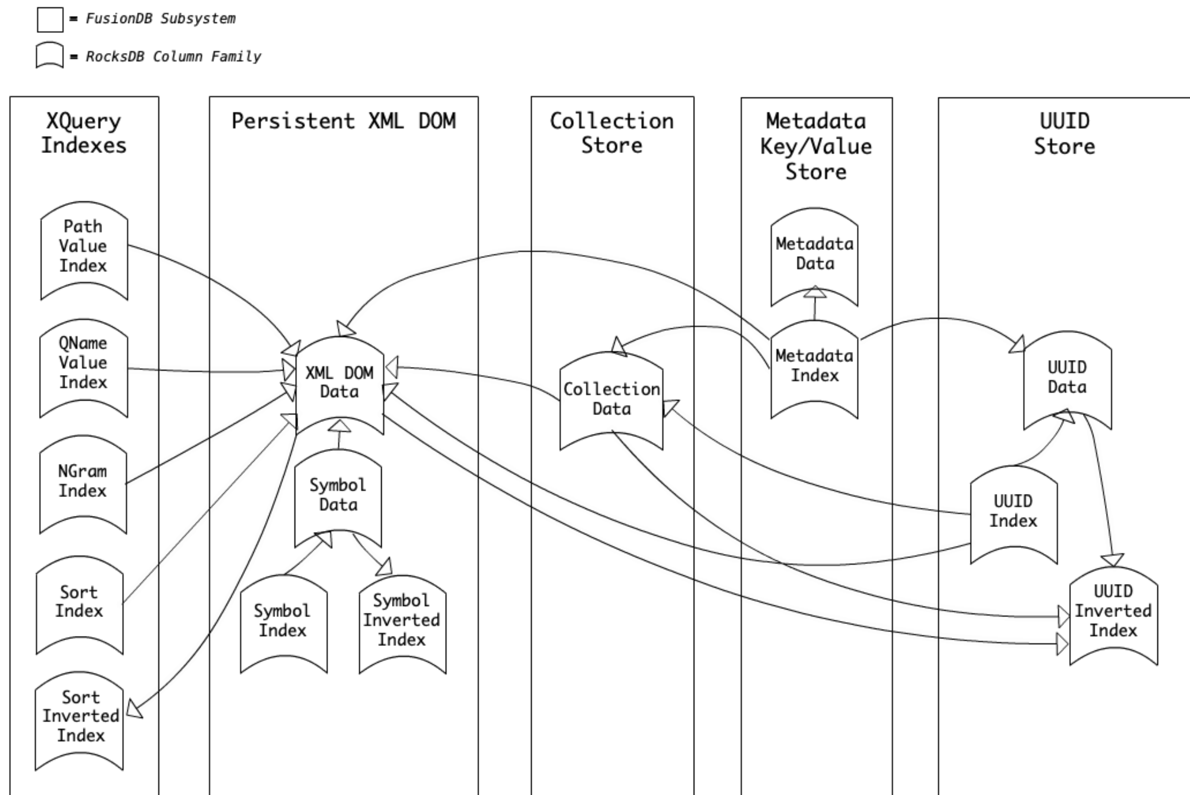

Figure 5. Main Column Families used by FusionDB

4. Conclusion

In 2014, the inception of this project came about as a response to a number of issues that we had identified over a period of several years from both users of Open Source NXDs and developers. We therefore set out to build a better modern Open Source database system which could handle XML Documents as well as other data models.

Initially we opted to fork and modify eXist-db as the base for FusionDB. Our plan was to firstly, fork eXist-db and replace its storage engine with RocksDB, before undertaking further radical changes to our fork of eXist-db. Then secondly, to build out new features, both alongside and with our fork of eXist-db. At that time, we believed that forking the eXist-db codebase with which we were familiar, would result in a faster route to delivery for FusionDB. Our initial estimate was calculated at between six and twelve months of development time. Unfortunately, this proved not to be the case, in fact quite the opposite! eXist-db has evolved over 18 years, and as such is a feature rich and complex product. As an

Open Source project, many developers have contributed and moved on, and there are several areas that are undocumented, untested, and no longer well understood by its current development team. Our goal, was to pass 100% of eXist-db's test suite with FusionDB. This proved to be a tough although not insurmountable challenge, however, achieving this revealed many bugs and incorrect tests in eXist-db which we also had to fix (and chose to contribute back to the eXist-db Open Source project). Whilst we have gained much by forking eXist-db, with hindsight we believe that it would have been quicker to develop a new eXist-db compatible system from scratch without forking eXist-db.

With FusionDB, in the first instance, we have now addressed many of the issues identified by users and developers. RocksDB has given us much greater stability, crash resilience, recovery, and performance. We have created an ACID transactional database system which offers Snapshot Isolation and ensures consistency and atomicity. We have invested a great deal of time and effort in to improving the locking situation of eXist-db, which when coupled with our Snapshot Isolation has also reduced the number and duration of locks that we require in FusionDB. Reduced locking has improved performance generally and decreased contention, thus also improving vertical scalability. Likewise, it has also prevented system maintenance tasks such as Backup or Reindexing from blocking access to the database, as these now have their own dedicated snapshot on which to act. Additionally we have also replaced and contributed new subsystems which add Key/Value Metadata, UUID locators, and improved Binary Document storage. This paper has both described much of the design rationale behind these technical achievements, and revealed the architecture of their implementation.

Whilst we have not yet completed work on the issues of complex graph-like cross-reference queries, or clustering to deliver horizontal scalability, they are firmly on our roadmap. The construction of FusionDB has not been an easy path to navigate, but ultimately we felt that the work was both justified and deserved by the XML community. We believe that FusionDB has a huge amount of potential, and we are excited to both share it with the world and develop it further.

5. Future Work

A great deal of research and development work has gone into the development of FusionDB. Some of this work, such as that around Locking and Binary Storage, has been contributed back to eXist-db, whilst more modest improvements have been contributed back to RocksJava.

We are approaching the stage where we believe we could release a first version of FusionDB, however before that becomes a reality, there is still work to complete in the short-term concerning:

- Collection Caching.

  The Collection Cache of eXist-db is shared between multiple transaction but is yet version aware. The Collection cache was originally designed to reduce Java Object creation and memory-use by ensuring only one Object for each distinct Collection, and reduce disk I/O by avoiding the need to frequently re-read Collections from disk. To cope with transactional updates upon cached in-memory Collection objects, the Collection Cache needs to be either, removed entirely, or revised to be MVCC aware so that a transaction is accessing the correct in-memory version of the on-disk Collection.
- Locking and Transactions.

  Whilst we have hugely improved the current state of locking in eXist-db, for FusionDB there are many more opportunities for reducing the number of locks whilst preserving transaction isolation and consistency. One obvious area of work, would be looking for any advantages in replacing FusionDB's transactions with the new facilities recently available in RocksDB for pessimistic and optimistic transactions.
- Performance.

  Ultimately our goal is to push FusionDB to out-perform any other Document Database. Our first performance goal is to have FusionDB out-perform eXist-db on any operation or query. At present, FusionDB is faster than eXist-db for many operations, but slower than eXist-db for several others. Our changes so far to the fork of eXist-db used by FusionDB have been far-reaching but somewhat conservative, we have focused on carefully reproducing existing behaviour. We are looking forward to drastically improving performance, by exploiting the many untapped optimisations that our new technology stack affords.
- Licensing.

  Our goal has always been to release FusionDB as Open Source software, and this has not changed. However, we want to ensure that we choose an appropriate license or licenses, that enable us to build the best possible community and software. We were previously considering a dual-licensed approach, whereby users could choose AGPLv3 or a Commercially licensed exemption to AGPLv3.

  Recent developments in software licensing require us to properly reevaluate this choice. Concrete examples of this in the database market are:

  - Redis previously licensed its open source version under a BSD 3-clause, and the modules for that under AGPLv3. Its Enterprise edition is only available under a commercial license without source code. Around the August 22nd 2018, Redis relicenced its modules from AGPLv3 to Apache 2.0 modified with Commons Clause, changing them from Open Source to Source Available.

- MongoDB's database was previously licensed under AGPLv3. On October 16th 2018, MongoDB relicensed its software under a new license SSPL (Server Side Public License) it created to solve their perceived problems with AGPL. Whether SSPL is an Open Source or Source Available license is still to be determined by its submission to the OSI.
- Neo4j previously licensed its Community Edition under GPLv3, and its Enterprise edition under AGPLv3. On November 15th 2018, Neo4j changed to an Open Core model, whereby their Enterprise Edition is now only available under a commercial license without source code.

  The reason for these licensing changes have often been cited as protecting commercial interests of Open Source companies. However it is clear that each of these companies is taking a different route to achieve a similar outcome. We believe that further study of the outcomes of these changes is merited, with a focus on their acceptance or rejection by their respective previous open source users.

In the medium term, areas of future work that are already under consideration are:

- Further Document Models

  Of primary concern is support for natively storing JSON documents. Several options exist for querying across XML and JSON document models, including XQuery 3.1 Maps/Arrays and JSONiq, further work is needed to identify the best approach for FusionDB. In addition, some research has already been undertaken into also storing Markdown and HTML5 documents natively, and will likely be further expanded upon.
- Distributed Database

  After correctness and single-node performance, we consider this to be the most important feature of modern database that are designed to scale. Further research into establishing a multi-node shared-nothing instance of FusionDB is highly desirable.
- Graph Database

  As discussed (see Section 1.1) many users have complex cross document query requirements that would likely benefit from more complex graph based linking and querying. The key/value facilities of RocksDB have been previously been demonstrated by ArangoDB and DGraph as a feasible base for building Graph database models. Integrating a graph model into FusionDB, and the subsequent opportunities for cross-model querying is an interesting topic for further research in FusionDB.

Bibliography

[1] Hal Berenson. Phil Bernstein. Jim Gray. Jim Melton. Elizabeth O'Neil. Patrick O'Neil. June 1995. *A Critique of ANSI SQL Isolation Levels*. Association for

Computing Machinery, Inc. http://research.microsoft.com/pubs/69541/tr-95-51.pdf .

[2] Wolfgang Meier. eXist-db. 2006-10-13T09:35:53Z. *Re: [Exist-open] Lock (exclusive-lock, etc) does anybody have a book, tutorial, or helpful explanation?*. exist-open mailing list. https://sourceforge.net/p/exist/mailman/message/14675343/ .

[3] Wolfgang Meier. eXist-db. 2005-07-26T07:15:39Z. *Initial checkin of logging&recovery code.*. GitHub. https://github.com/eXist-db/exist/commit/1ed4d47f01c9ee2ede#diff-16915756b76d37e10eba8b939a1e2f40R1648.

[4] Wolfgang Meier. Adam Retter. eXist-db. 2018-04-23T20:21:37+02:00. *[bugfix] Fix failing recovery for overflow DOM nodes (>4k). Addresses #1838*. GitHub. https://github.com/exist-db/exist/commit/6467898.

[5] Pierrick Brihaye. eXist-db. 2007-02-13T16:13:31Z. *Made a broader use of transaction.registerLock(). Left 2 methods with the old design (multiple collections are involved).*. GitHub. https://github.com/eXist-db/exist/commit/cd29fef34f91d471d79966b963d1657fd9186f89.

[6] Dmitriy Shabanov. eXist-db. 2010-12-14T17:44:07Z. *[bugfix] The validateXMLResourceInternal lock document, but do not unlock with a hope that it will happen up somewhere. Link lock with transaction, so document lock will be released after transaction completed or canceled.*. GitHub. https://github.com/eXist-db/exist/commit/5af077cd441039d9b8125a9149f399b9bd8ee95c.

[7] Adam Retter. eXist-db. 2018-10-31T19:12:22+08:00. *[bugfix] Locks do not need to be acquired for the transaction life-time for permission changes*. GitHub. https://github.com/eXist-db/exist/commit/0faee22bc792629d625807319459a164d00691c1.

[8] eXist-db. *eXist-db B+ tree*. GitHub. https://github.com/eXist-db/exist/blob/eXist-4.5.0/src/org/exist/storage/btree/BTree.java#L26.

[9] dbXML. *dbXML 1.0b4 B+Tree*. SourceForge. http://sourceforge.net/projects/dbxml-core/files/OldFiles/dbXML-Core-1.0b4.tar.gz/download.

[10] Christian Grün. BaseX. 2014-05-23T17:54:00Z. *Transaction Management - Concurrency Control*. BaseX. http://docs.basex.org/index.php?title=Transaction_Management&oldid=10646#Concurrency_Control.

[11] J. Chris Anderson, Jan Lehnardt, and Noah Slater. Apache. 2010-02-05. *CouchDB - The Definitive Guide. Storing Documents.* 1st. O'Reilly. http://guide.couchdb.org/draft/documents.html.

[12] MongoDB, Inc.. 2018-11-29. *What is MongoDB?*. https://www.mongodb.com/what-is-mongodb.

[13] Marklogic Corporation. 2018-11-29. *MarkLogic Application Developers Guide. Working with JSON.* https://docs.marklogic.com/guide/app-dev/json.

[14] Marklogic Corporation. 2018-11-29. *MarkLogic Application Developers Guide. Understanding Transactions in MarkLogic Server.* https://docs.marklogic.com/guide/app-dev/transactions.

[15] David Gorbet. Marklogic Corporation. 2018-11-30. *I is for Isolation, That's Good Enough for Me! - MarkLogic*. https://www.marklogic.com/blog/isolation/.

[16] Sinclair Target. Two-Bit History. 2017-09-21. *The Rise and Rise of JSON*. https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html.

[17] Christian Grün. BaseX. 2014-11-20T14:02:00Z. *SQL Module*. BaseX. http://docs.basex.org/index.php?title=SQL_Module&oldid=11101.

[18] Dan McCreary. XQuery Examples Collection Wikibook. 2011-04-14T16:42:00Z. *XQuery SQL Module*. Wikibooks. https://en.wikibooks.org/wiki/XQuery/XQuery_SQL_Module.

[19] Oracle. 2015-07-10. *Introduction to Berkeley DB XML. Database Features.* Oracle. https://docs.oracle.com/cd/E17276_01/html/intro_xml/dbfeatures.html.

[20] Pete Aven. Diane Burley. MarkLogic. 2017-05-11. *Building on Multi-Model Databases. How to Manage Multiple Schemas Using a Single Platform.* 1st. 24-26. O'Reilly. http://info.marklogic.com/rs/371-XVQ-609/images/building-on-multi-model-databases.pdf.

[21] Philip Lehman. S BING YAO. 1981. *Efficient Locking for Concurrent Operations on B-Trees*. ACM. *ACM Transactions on Database Systems*. 6. 4. 650-670.

[22] Anastasia Braginsky. Erez Petrank. Technion - Israel Institute of Technology. 2012. *A Lock-Free B+tree. Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures. SPAA '12*. ACM. 58-67. 978-1-4503-1213-4. 10.1145/2312005.2312016.

[23] Justin Levandoski. David Lomet. Sudipta Sengupta. Microsoft Research. 2014-04-08. *The Bw-Tree: A B-tree for new hardware platforms*. IEEE. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 978-1-4673-4910-9. 10.1109/ICDE.2013.6544834.

[24] Lars Arge. 1995. *The Buffer Tree: A New Technique for Optimal I/O Algorithms*. BRICS, Department of Computer Science, University of Aarhus. *Lecture Notes in Computer Science*. 995. *WADS 1995*. Springer.

[25] Gerth Stølting Brodal. Rolf Fagerberg. 2003. *Lower Bounds for External Memory Dictionaries*. BRICS, Department of Computer Science, University of Aarhus. Society for Industrial and Applied Mathematics. *Proceedings of the Fourteenth*

Annual ACM-SIAM Symposium on Discrete Algorithms. *SODA '03*. 546-554. 0-89871-538-5.

[26] brandur. 2017-05-07. *The long road to Mongo's durability*. https://brandur.org/fragments/mongo-durability.

[27] Nassyam Basha. 2018-04-01. *Database Lost write and corruption detections made easy with dbcomp - 12.2*. Oracle User Group Community. https://community.oracle.com/docs/DOC-1023009.

[28] Robert Newson. 2017-01-19. *[COUCHDB-3274] eof in couch_file can be incorrect after error - ASF JIRA*. Apache Software Foundation. https://issues.apache.org/jira/browse/COUCHDB-3274.

[29] Jeffrey Aguilera. 2005-10-07T08:08:00Z. *[DERBY-606] SYSCS_UTIL.SYSCS_INPLACE_COMPRESS_TABLE fails on (very) large tables - ASF JIRA*. Apache Software Foundation. https://issues.apache.org/jira/browse/DERBY-606.

[30] *Issues · jankotek/mapdb*. 2019-01-20. GitHub. https://github.com/jankotek/mapdb/issues?utf8=%E2%9C%93&q=is%3Aissue+corrupt.

[31] Ryan S. Dancey. *License-discuss Mailing List*. *OpenLDAP license.* 2001-04-09T23:22:48Z. http://lists.opensource.org/pipermail/license-discuss_lists.opensource.org/2001-April/003156.html.

[32] Jung-Sang Ahn. Chiyoung Seo. Ravi Mayuram. Rahim Yaseen. Jin-Soo Kim. Seung Ryoul Maeng. 2016-03-01. 2015-05-20. *ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys*. IEEE. *Published in: IEEE Transactions on Computers*. 65. 3. 902-915. 10.1109/TC.2015.2435779.

[33] Patrick O'Neil. Edward Cheng. Dieter Gawlick. Elizabeth O'Neil. 1996. *The Log-structured Merge-tree (LSM-tree)*. *Acta Informatica*. 33. Springer-Verlag New York, Inc.. 351-385. 10.1007/s002360050048.

[34] Ilya Grigorik. 2012-02-06. *SSTable and Log Structured Storage: LevelDB*. https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/.

[35] Siying Dong. 2015-02-27. *WriteBatchWithIndex: Utility for Implementing Read-Your-Own-Writes*. https://rocksdb.org/blog/2015/02/27/write-batch-with-index.html.

[36] Dhruba Borthakur. Facebook. 2013. *The Story of RocksDB*. *Embedded Key-Value Store for Flash and RAM.* https://github.com/facebook/rocksdb/blob/gh-pages-old/intro.pdf?raw=true.

[37] Symas Corporation. 2019. *LMDB TECHNICAL INFORMATION*. *Other Projects.* https://symas.com/lmdb/technical/#projects.

[38] Facebook. GitHub. 2019. *rocksdb/USERS.md at v5.17.2 · facebook/rocksdb. Users of RocksDB and their use cases. Other Projects.* https://github.com/facebook/rocksdb/blob/v5.17.2/USERS.md.

[39] Facebook. GitHub. 2019. *rocksdb/block_builder.cc at v5.17.2 · facebook/rocksdb.* https://github.com/facebook/rocksdb/blob/v5.17.2/table/block_builder.cc#L10.

[40] Facebook. GitHub. 2019. *Prefix Seek API Changes · facebook/rocksdb Wiki.* https://github.com/facebook/rocksdb/wiki/Prefix-Seek-API-Changes.

[41] Facebook. GitHub. 2019-01-23. 2019-01-21. *RocksDB - Wikipedia. Integration.* https://en.wikipedia.org/wiki/RocksDB#Integration.

[42] Dan Ports. Kevin Grittner. 2012-08. *Serializable Snapshot Isolation in PostgreSQL.* VLDB Endowment. *Proc. VLDB Endow..* 5. August 2012. 1850-1861. 10.14778/2367502.2367523.

[43] Adam Retter. *Locking and Cache Improvements for eXist-db.* 2018-02-05. https://www.evolvedbinary.com/technical-reports/exist-db/locking-and-cache-improvements/locking-and-cache-improvements-20180205.pdf.

[44] Adam Retter. *Asymmetrical Locking for eXist-db.* 2018-02-05. https://www.evolvedbinary.com/technical-reports/exist-db/asymmetrical-locking/asymmetrical-locking-20180205.pdf.

[45] Gudka Khilan. Susan Eisenbach. *Fast Multi-Level Locks for Java. A Preliminary Performance Evaluation.* 2010. *EC^2 2010: Workshop on Exploiting Concurrency Efficiently and Correctly.* https://www.cl.cam.ac.uk/~kg365/pubs/ec2-fastlocks.pdf.

[46] Gudka Khilan. 1975. *Granularity of Locks in a Shared Data Base. Proceedings of the 1st International Conference on Very Large Data Bases.* VLDB '75. 10.1145/1282480.1282513. 978-1-4503-3920-9. ACM. 428-451.

[47] Adam Retter. *BLOB Deduplication in eXist-db.* 2018-11-27. https://blog.adamretter.org.uk/blob-deduplication/.

[48] Adam Retter. *JNI Construction Benchmark. Results.* 2016-01-18. https://github.com/adamretter/jni-construction-benchmark#results.