Revisiting Concurrency in High-Performance NoSQL Databases

Yuvraj Patel*, Mohit Verma+0, Andrea C. Arpaci-Dusseau*, Remzi H. Arpaci-Dusseau*
*Computer Sciences Department, University of Wisconsin-Madison, +NVIDIA

Abstract

We measure the performance of five popular databases and show that single-node performance does not scale while hosting data on high-performance storage systems (e.g., Flash-based SSDs). We then analyze each system, unveiling techniques each system uses to increase concurrent performance; our taxonomy places said approaches into six different categories (thread architecture, batching, granularity, partitioning, scheduling and low-level efficiency) and thus points towards possible remedies that can scale the system. Finally, we introduce Xyza, a modified version of MongoDB that uses a wide range of classic and novel techniques to improve performance under concurrent, write-heavy workloads. Empirical analysis reveals that Xyza is 2× to 3× faster than MongoDB and scales well (up to 32 processing cores).

1 Introduction

Parallelism is a core technique used to increase distributed system performance. However, using parallelism, both across machines and within a single machine, is challenging; without careful management of concurrent activity, performance can be lost and correctness can be sacrificed.

The goal of *concurrency control* is to ensure correctness among operations that are executing simultaneously in a system. Ideally, concurrency control mechanisms should enable correctness while allowing the system to scale well with the number of cores. Achieving this reality is difficult; decades of efforts have been put forth to understand limitations and propose new techniques that do not limit scaling [12–15, 35, 40, 41, 50, 51, 55].

Technology changes are also afoot in the data center. For example, storage has been a central performance bottleneck in scalable databases and file systems. Researchers and practitioners have proposed systems that can effectively utilize the newly available faster storage media [4, 31–33, 42, 44, 46, 52]. In addition, each node now consists of more cores and it becomes crucial to use them all to achieve high performance; if single-node performance does not scale, more machines must be recruited to tackle the task at hand, thus increasing costs.

Thus, the central question that we address: are standard concurrency control mechanisms effective enough to harness faster storage devices and the large number of cores available in modern systems? In this paper, we answer this question by first analyzing the performance of five popular NoSQL databases – MongoDB [36], Cassandra [1], CouchDB [2], Oracle NoSQL DB [21], and ArangoDB [5]. Our first and most important finding: on a single node with many cores, all five systems do not scale well as the client load increases; as a result, despite the presence of a fast storage device (in this case, a modern SSD [11]), system throughput is notably lower than what could be realized, sometimes by 3× or more.

To understand the concurrency control techniques utilized in these systems, we classify each popular technique into six categories: thread architecture, batching, granularity, partitioning, scheduling, and low-level efficiency. Classifying these techniques helps us understand weaknesses and identify why the databases are not scaling. Based on this analysis, we observe that common scaling techniques (such as partitioning and scheduling) are not well utilized; simply put, these systems, optimized for slow-storage media, are not yet concurrent enough to fully realize the performance of modern storage systems.

To remedy this problem, and to demonstrate how high performance can be attained, we present Xyza – a modified version of MongoDB that uses a wide range of classic and novel techniques to deliver high performance and scalability. In particular, we concentrate our improvements on three techniques that most affect performance: partitioning, scheduling, and low-level efficiency.

Specifically, we partition data structures such as vectors, the journal, and the key-space to improve the performance of the write path. We introduce two novel techniques in scheduling: contention-aware scheduling and semantic-aware scheduling. Contention-aware scheduling considers each lock as a resource and only schedules an operation if there will be no such contention. Semantic-aware scheduling selectively drops conflicting operations when no difference can be realized by clients (due to weakened levels of consistency). Finally, using atomic primitives, we fast-path the common cases allowing them a higher share of system resources, thus increasing performance under load. Overall, when Xyza is subjected to a concurrent, write-heavy workload, we observe that it scales well up to 32 cores and is 2× to 3× faster than MongoDB.

2 Concurrency Analysis

The simplest way to measure if a system scales vertically is by gradually scaling the resources in a single system and then measuring its performance. A truly scalable system will demonstrate a linear performance increase. How-

[0]Work done while studying at University of Wisconsin-Madison.

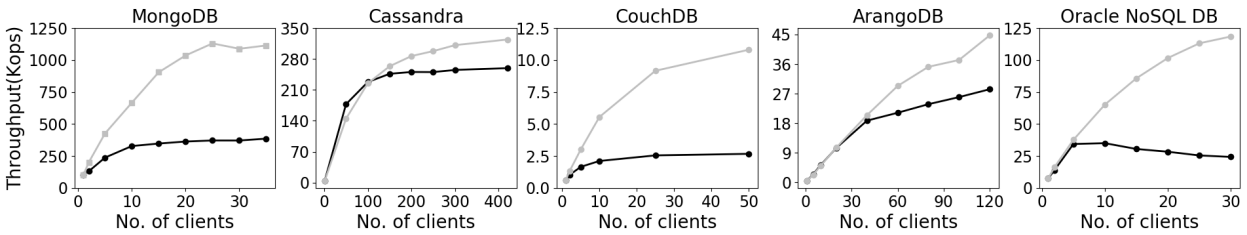

Figure 1: Performance scaling of NoSQL Databases. Dark line represents single instance run. White line represents multiple instances run. Consistency options used (i) MongoDB 3.2.11 - w:0,j=false; (ii) Cassandra 3.11 - commitlog_sync:periodic,replication_factor:0, (iii) CouchDB 1.6.1 - delayed_commits:true, (iv) ArangoDB 3.2 - database.wait-for-sync:false and (v) Oracle NoSQL DB 4.3.11 - master and replica sync policy:KV_SYNC_NONE, acknowledgment_policy:KV_ACK_NONE.

ever, if the system does not scale well, how can one determine whether concurrency control is at the root of the problem? Also, how can one discover if the system cannot utilize available resources and hence limits performance? In this paper, we particularly concentrate on CPU scaling.

To identify if concurrency control is a bottleneck, we conduct experiments in two modes. In the first mode, a single instance of a NoSQL database is run while the amount of load generated on these databases is increased via additional clients. In the second mode, multiple instances are instantiated and each client is configured to interact with a *separate* instance (on the same machine). As there is no interference among client operations, the hypothesis is that the overall cumulative performance will scale in the latter case, and reveal the potential optimal scaling of the system within that particular configuration.

In these experiments, we focus on write performance on a single node as it is more challenging to scale (as compared to reads). Writes modify the global structures more often and hence maintaining integrity poses a challenge under highly concurrent workloads.

We choose the weakest consistency option available for the insert operation for all these databases. The weakest form of consistency is designed to achieve the highest levels of performance. Thus, it is useful in understanding whether concurrency control is a bottleneck.

2.1 Experimental Setup and Workload

We perform our experiments on an 2.4 GHz Intel Xeon E5-2630 v3. It has 2 sockets and each socket has 8 physical cores with hyper-threading enabled. We use two such machines for our experiments where one is used as a server while another is used to run client programs. Both of these machines are connected via a 10Gbps network and has one 480 GB SAS SSD and 128 GB RAM. Both have Ubuntu 14.04 with kernel version 4.9. For ArangoDB, we use five client machines as a single client machine CPU utilization saturates, limiting the load that is generated on the server. Due to memory constraints in the Java environment, we only instantiate two and ten instances of Cassandra and Oracle NoSQL DB respectively.

Our workload consists solely of insert operations. A client program issue inserts in a loop. The client program generates the key and the start and end key range is passed as a parameter to the client program. The value is 100 bytes and consists of random characters. We vary the number of client programs and all client programs execute on the same machine as discussed above. Totally, all the clients combined together insert fifty million records. We measure the time taken to complete the insert operations to calculate throughput and monitor the CPU, disk and network utilization on the server using dstat command [3]. For Cassandra, we use their stress tool [29] and it is configured to mimic the above-mentioned workload. The throughput reported is the average of five runs.

2.2 Results and Analysis

The performance comparison of each NoSQL database for single and multiple instances is shown in Figure 1. We observe that for single instance mode, neither the storage nor the network is a bottleneck. This observation is consistent among all the databases; the average disk bandwidth ranges between 5-150 MB/sec. The throughput for single instance mode saturates as the number of clients increase. However, the CPU utilization of the server machine increases as the number of clients increase.

With multiple instances, the throughput either increases linearly or sub-linearly. We observe that the CPU utilization of the server is similar to that of the single instance mode. With multiple instances, unlike the single instance case, each client interacts with a separate instance and all the resources are independent and isolated. As there is no interference from other clients, the concurrency control mechanism does not have to handle contention. With less contention, each instance is able to work independently, thereby increasing the throughput of the system.

The gap between single instance and multiple instances for Cassandra and Oracle NoSQL DB is not the same as that of other databases because the total number of instances compared is smaller. However, even with such a low number of multiple instances, we still see a significant difference in the throughput.

Based on the throughput comparison shown in Figure 1, we show that current concurrency control mechanisms are not able to scale well as the number of clients increases. Given the availability of many-core systems, and newer media such as Flash and NVM technologies, it is critical to design systems keeping in mind these technologies.

2.3 Popular Design Techniques

Modern systems consist of many cores, thus making it extremely important to exploit the high parallelism offered to achieve high performance. Today's systems rely on a variety of techniques to ensure scalability and correctness, many of which have been suggested in the research literature. However, only some of these have been implemented in current systems, whereas others have not. We hypothesize that we can improve the performance scaling of these systems by focusing on a certain class of techniques that are not being currently implemented.

We present a qualitative analysis of these five systems based on the techniques they use in Table 1. This table also discusses our new system (Xyza), an extension of MongoDB; we will present Xyza in the coming section. The table is generated by manually analyzing the code, architecture, and design documents of each system.

Thread Architecture: We focus on one aspect of thread architecture: whether the threads are initialized statically at startup or dynamically created/terminated at run time. Static handling of threads can lead to either under-utilization or over-utilization of resources; it can be challenging to identify an optimal thread-pool size. Dynamic thread creation augurs well, creating the possibility for the system to adjust itself according to system load. All five systems choose the former while MongoDB, Oracle NoSQL DB, and ArangoDB choose the latter.

Batching: Batching is the process of grouping multiple operations to reduce overheads; for example, processing two requests together avoids some locking overhead, as locks only need to be acquired/released once. Similarly, a single larger disk I/O is generally more efficient than smaller individual writes. However, batching can increase the latency of operations; furthermore, it is sometimes difficult to create large batches, depending on the system and workload [47]. All five systems that we study utilize batching to improve the performance of their systems.

Granularity: Granularity refers to how a system handles data access and at what level such data access is allowed. There are two prime issues that need to be addressed: write/write coordination and write/read coordination. A simple coarse-grained approach is to allow exclusive access. However, it is extremely inefficient and limits scaling. To mitigate this, a fine-grained approach of allowing concurrent access to multiple threads is preferred. The coordination is ensured using locks or lock-free data structures. Locks enforce serialization and hence ensures integrity. Many techniques have been proposed to allow concurrent access to readers and writers such as snapshot isolation [12], multi-version concurrency control [14], and optimistic concurrency control [45].

MongoDB, ArangoDB and Oracle NoSQL DB use fine-grained locking to ensure write-write coordination. On the other hand, Cassandra uses last-write-wins strategy and allows two concurrent threads to work on a single record. However, such a strategy impacts read performance as the read operation will have to sweep all the relevant entries to find the latest. From the write-read coordination perspective, the majority of them use MVCC for write/read coordination while Cassandra uses row-level isolation and Oracle NoSQL DB uses read/write locks.

Partitioning: Using partitioning, a resource can be broken into multiple parts and then each partition can either be allowed exclusive access or concurrent access. Generally, the number of partitions is not too large, and also depends on the data structure in question. The key aspect is that as the number of partitions increase, more threads can access the data parallelly improving the scalability.

Interestingly, none of the systems we study partition resources within a single node effectively. Oracle NoSQL DB does key-space partitioning and the number of partitions is configurable. MongoDB partitions the locks to scale the common case. We believe that partitioning is a necessary component of any design to scale well and we explore partitioning of resources in our approach.

Scheduling: Scheduling deals with when to schedule operations based on resource availability and data-integrity constraints. Effective scheduling can improve performance by re-ordering requests to minimize contention. Scheduling and partitioning go hand-in-hand, as a system can schedule operations that belong to different partitions in parallel; many systems have relied on data partitioning and scheduling for scalability [25]. Similarly, MongoDB uses multi-granularity locking [34] that allows compatible operations to run in parallel.

Locks help enforce a schedule of how operations are executed. However, as system scales, contention to acquire locks increases limiting the performance. To scale well, we introduce an effective scheduling approach, called *contention-aware scheduling*(Section 3.2), that views locks as a resource and prevents two active threads from accessing the same lock or partition. To prevent the wait queues from growing longer due to contention, we introduce *semantic-aware scheduling*(Section 3.2) that drops the operations having weak consistency option under certain contention scenarios.

Low-level efficiency: With multiple threads, acquiring and releasing locks is costly. Locks are implemented using hardware-provided atomic instructions and many designers choose to use these low-level atomic primitives directly to improve efficiency. These primitives are also used to build concurrent data structures and are more efficient than their lock-based counterparts [11, 16]. Using atomic primitives extensively makes programming hard, complicated, and prone to bugs. All the systems we study use atomic primitives to optimize some of their code paths. However, there is room for further optimization as we believe that low-level efficiency can be used to opti-

			MongoDB	Cassandra	CouchDB	Oracle NoSQL	ArangoDB	Xyza
Thread Architecture	**Static Initialization**		Yes	Yes [30]	Yes	Yes	Yes [8]	Yes
	Dynamic handling		Yes			Yes	Yes [10]	Yes
Batching operations			Yes [37]	Yes [26]	Yes [24]	Yes [20]	Yes [7]	Yes
Granularity	**W/W coordination**	**Concurrent**	Exclusive locks [38]	Last write wins [27]		W/W locks [18, 22]	Exclusive locks [9]	
		Exclusive			Per database [23]			Per partition
	W/R coordination		MVCC [39]	Row-level isolation [27]	MVCC [23]	R/W locks [18, 22]	MVCC [6]	MVCC
Partitioning	**Key-space**					Yes [19]		Yes
	Other data structures			Yes [28]				Yes
	Locks		Yes					
	Journal		Single [39]	Single [17]			Single	Per-client
Scheduling	**Non-overlapping operations**		Yes [38]	Yes [27]		Yes [18]	Yes [9]	Yes
	Contention-aware							Yes
	Semantic-aware							Yes
Low-level efficiency			Yes	Yes	Yes	Yes	Yes	Yes

Table 1: Key design techniques used for concurrency control mechanism. Empty cells denotes that the particular feature is not implemented.

mize common cases.

To summarize the above analysis, we observe that resource contention due to improper partitioning and inefficient scheduling limits scalability. Optimizing common cases needs to be emphasized for scaling systems.

3 Xyza

In this section, we discuss how a combination of old and new techniques can address the fundamental scaling problems. We present Xyza, an extension of MongoDB's existing concurrency control architecture. We choose MongoDB as it is a widely used and important system; it also presents us with ample room for improvement. Xyza inherits everything from MongoDB other than the concurrency control techniques discussed below.

3.1 Partitioning

MongoDB does not effectively utilize partitioning despite the presence of partitionable resources like the journal, key-space, global vectors, and maps, that Xyza partitions.

Per client partitioning: For each new client connection, MongoDB creates a separate thread to handle their requests. Consequently, the thread terminates once the connection closes. Xyza takes advantage of this design aspect and partitions many resources per client thread. Due to exclusive access, there is no need for locking.

Xyza partitions the system-wide journal used by MongoDB into a per-client journal. As the client threads now have exclusive access to their journal, they do not have to find/allocate active slots in the single global journal and hence avoid the use of locks/atomic primitives as would be necessary for coordination among threads.

MongoDB associates each write operation with a session object and maintains a vector of session objects and protects it using a lock. In Xyza, we partition the session vector so that each client will have its own session vector and have exclusive access to it.

Additionally, effective partitioning helps in pinning the client threads to a particular core. Without pinning, threads can move between cores leading to frequent cache invalidations impacting the performance.

Key-space: MongoDB uses sharding to partition its key-space horizontally across multiple nodes; however, within a single node, it does not partition the key-space. It prevents write-write conflicts using locks. Xyza extends the partitioning approach to the key-space ranges and partitions the key-space accordingly. Thus, instead of relying on the locks for concurrent page accesses, more operations can execute in parallel as long as the number of partitions is more than the cores available in the node.

3.2 Scheduling

Contention-aware scheduling: As the locks are within the application domain, the operating system does not have a view of how they are used and hence cannot schedule operations based on their availability. However, the application can detect the lock availability and accordingly schedule operations that are not likely to contend. Taking advantage of this fact, we propose *contention-aware scheduling*, where no two operations that require the same set of locks will be scheduled simultaneously. Overlapping operations wait until the current operations complete. We extend this idea to partitions too. Using such scheduling helps in avoiding locks and thus one can view scheduling as a form of primitive synchronization. This ensures that correctness is not compromised. Moreover, Xyza does not allow a single operation to span across two partitions and hence avoids deadlocks.

The hierarchical locking approach [38] taken by MongoDB is not enough to ensure scaling. Thus, we include contention-aware scheduling in addition to hierarchical locking. Contention-aware scheduling relies on partitioning to mitigate the problem of false conflicts by precisely knowing which key ranges are busy.

Semantic-aware scheduling: As part of scheduling the operations, the system should decide what happens to operations that are not scheduled due to lack of resources. The most obvious choice of waiting until the resources needed are available comes from the fact that the execution of the operation is absolutely necessary [14]. However, another option arises due to the consistency options available within many of today's NoSQL databases. Specifically, Xyza uses this semantic knowledge, particularly in cases where consistency is weakened, to decide what should be done in case the operation cannot be scheduled. Such operations can be dropped without wait-

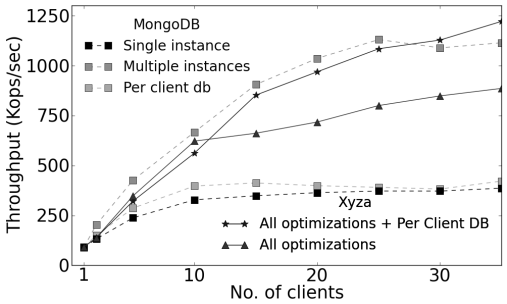

Figure 2: Xyza performance with various optimization techniques

ing as there is no guarantee associated with the weaker consistency option. Operations having stronger consistency will wait as the guarantees are stronger too.

3.3 Low-level Efficiency

We believe low-level efficiency should also target common cases and designers should use atomic primitives wherever possible to optimize performance. Basic read and write operations are one of the most common operations for any NoSQL database. To speed up this common case, we replace the complex lock manager with a simple wait-signal mechanism where each resource has its own set of mutex and condition variables that can be efficiently used for waiting and signaling. To optimize the common case further, we use atomic primitives instead of traversing the hierarchy tree that acquires the mutex to update the state information of each resource.

4 Evaluation

To highlight Xyza's performance, we conduct experiments using workloads as described in Section 2. As seen in Figure 2, the performance of Xyza is more than 2× faster than MongoDB. Xyza's performance is roughly 80-90% of that of the multiple instances performance of MongoDB. As the machine has 16 physical cores, Xyza's performance improves slowly after 15 clients.

We perform another experiment where each client writes to its own database and collection. We expect the performance to be on par with that of multiple instances of MongoDB as each client is accessing a different database and collection. The results shown in Figure 2 confirms our expectations.

To test how Xyza performs when subjected to conflicting operations where multiple clients write in the same keyspace partition for the same workload mentioned earlier; we observe around 8-12% operations are dropped due to semantic-aware scheduling for all clients(1-35). We also observe a decrease in the throughput by 3-11% along with a drop in CPU utilization around 4-9%.

5 Related Work

Key-value (KV) stores such as FAWN [4], WiscKey [44], NVMKV [46], SILT [42], etc., proposed new datastructures or optimizations from the storage perspective to improve performance. Instead, our work concentrates on concurrency control techniques that are general and can likely be used in any system.

To improve throughput, several in-memory KV stores have been proposed such as Masstree [45], MICA [43], Memcached [47], RAMCloud [48], Hstore [49], VoltDB [54], and Silo [53]. These systems are carefully designed to take advantage of DRAM and cache characteristics; our work shows that it is time to consider aggressive concurrency mechanisms even in persistent KV stores.

Most of these stores, including Masstree, RAMCloud, Memcached, and Silo, use a single partition in a single node. Masstree chooses fine-grained concurrency and optimistic concurrency control in their design. Similarly, Silo also uses a variant of optimistic concurrency control. Xyza chooses a coarse-grained approach to identify potential conflicts and accordingly schedule operations and avoids the traditional concurrency control approaches.

In contrast, Hstore, MICA, and VoltDB partition data among multiple cores and thus ensure that exclusive access to these partitions can avoid concurrency. Hstore assigns one thread per partition and thus avoids traditional concurrency control. Xyza never ties any thread to a partition but relies on the efficient scheduling of operations to ensure that only one thread is accessing the partition.

6 Conclusion

We have shown that today's databases do not scale well when the data is hosted on high-performing storage media, and thus fail to fully utilize the large number of cores common in today's systems. We presented a categorization of various concurrency control techniques into six categories: thread architecture, batching, granularity, partitioning, scheduling and low-level efficiency. A thorough analysis of these techniques indicated that further optimization was needed; today's concurrency control mechanisms are not effective in scaling and thus do not obtain peak performance. Finally, we present Xyza, an extension of MongoDB, that uses a combination of classic and novel techniques to achieve high performance and scaling. We believe that the proposed techniques can be applied in many other systems, and look forward to testing this hypothesis in the future. We look forward to analyzing the read path and revisiting concurrency control for read operations in future work.

7 Acknowledgments

We thank the anonymous reviewers and the members of ADSL for their valuable input. This material was supported by funding from NSF grants CNS-1421033 and DOE grant DESC0014935. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE,or any other institutions.

References

[1] Apache Cassandra. http://cassandra.apache.org/.

[2] Apache CouchDB. http://couchdb.apache.org/.

[3] Dstat - Linux man page. https://linux.die.net/man/1/dstat.

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[5] ArangoDB. ArangoDB. https://www.arangodb.com/.

[6] ArangoDB. Basics and Terminology. https://docs.arangodb.com/3.2/Manual/DataModeling/Documents/DocumentAddress.html.

[7] ArangoDB. Batch Requests. https://docs.arangodb.com/3.2/HTTP/BatchRequest/.

[8] ArangoDB. General Options. https://docs.arangodb.com/3.2/Manual/Administration/Configuration/GeneralArangod.html.

[9] ArangoDB. Locking and Isolation. https://docs.arangodb.com/3.2/Manual/Transactions/LockingAndIsolation.html.

[10] ArangoDB. Scalability. https://docs.arangodb.com/3.2/Manual/Scalability/.

[11] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[12] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[13] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1295–1309, New York, NY, USA, 2015. ACM.

[14] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[15] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[16] Anastasia Braginsky and Erez Petrank. A lock-free b+tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 58–67, New York, NY, USA, 2012. ACM.

[17] Apache Cassandra. Apache Cassandra - Storage Engine. http://cassandra.apache.org/doc/latest/architecture/storage_engine.html.

[18] Oracle Corporation. Concurrent Processing in Berkeley DB Java Edition. https://docs.oracle.com/cd/E17277_02/html/GettingStartedGuide/concurrentProcessing.html.

[19] Oracle Corporation. Identify the number of partitions. https://docs.oracle.com/cd/E26161_02/html/AdminGuide/store-config.html#num-partitions.

[20] Oracle Corporation. Oracle NoSQL BulkPut. https://blogs.oracle.com/nosql/oracle-nosql-bulkput.

[21] Oracle Corporation. Oracle NoSQL Database. https://www.oracle.com/database/nosql/index.html.

[22] Oracle Corporation. Oracle NoSQL Database vs. Cassandra. http://www.oracle.com/technetwork/products/nosqldb/documentation/nosql-vs-cassandra-1961717.pdf.

[23] Apache CouchDB. Apache CouchDB - Technical Overview. http://docs.couchdb.org/en/1.6.1/intro/overview.html#acid-properties.

[24] Apache CouchDB. Inserting documents in bulk. http://docs.couchdb.org/en/1.6.1/api/database/bulk-api.html#inserting-documents-in-bulk.

[25] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings*

of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, pages 419–434, Berkeley, CA, USA, 2016. USENIX Association.

[26] Datastax. Cassandra - Batching inserts, updates and deletes. https://docs.datastax.com/en/cql/3.3/cql/cql_using/useBatch.html.

[27] DataStax. How are Cassandra transactions different from RDBMS transactions? http://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlTransactionsDiffer.html?hl=row\%2Clevel\%2Cisolation.

[28] Datastax. How is data read? http://docs.datastax.com/en/archived/cassandra/3.x/cassandra/dml/dmlAboutReads.html.

[29] DataStax. The cassandra-stress tool. http://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsCStress.html?hl=cas.

[30] DataStax. Thread pool and read/write latency statistics. http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsThreadPoolStats.html?hl=thread\%2Cpool.

[31] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[32] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 25–36, New York, NY, USA, 2011. ACM.

[33] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[34] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Readings in database systems. chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base, pages 94–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[35] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, January 2017.

[36] MongoDB Inc. MongoDB. https://www.mongodb.com/.

[37] MongoDB Inc. MongoDB - Bulk Write Operations. https://docs.mongodb.com/v3.2/core/bulk-write-operations/.

[38] MongoDB Inc. MongoDB 3.2 FAQ: Concurrency. https://docs.mongodb.com/v3.2/faq/concurrency/.

[39] MongoDB Inc. MongoDB 3.2 WiredTiger Storage Engine. https://docs.mongodb.com/v3.2/core/wiredtiger/.

[40] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[41] Per-
AAke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011.

[42] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011. ACM.

[43] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association.

[44] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 133–148, Berkeley, CA, USA, 2016. USENIX Association.

[45] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*,

EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[46] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 207–219, Berkeley, CA, USA, 2015. USENIX Association.

[47] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[48] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[49] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[50] Dixin Tang, Hao Jiang, and Aaron J. Elmore. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *CIDR*, 2017.

[51] Alexander Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Comput. Surv.*, 30(1):70–119, March 1998.

[52] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 59–72, New York, NY, USA, 2009. ACM.

[53] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[54] Inc. VoltDB. VoltDB. `https://www.voltdb.com/`.

[55] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, 2016. ACM.