XStore: Xcelerated performance in time-series key/value STORagE systems

Lan Nguyen
Illinois Institute of Technology
Department of Computer Science
lnguyen18@hawk.iit.edu

Ioan Raicu
Illinois Institute of Technology
Department of Computer Science
iraicu@iit.edu

ABSTRACT

In recent years, we have seen an unprecedented growth of data in our daily lives ranging from health data from an Apple Watch, financial stock price data, volatile crypto-currency data, to diagnostic data of nuclear/rocket simulations. The increase in high-precision, high-sample-rate time-series data is a challenge to existing database technologies. We have developed a novel technique that utilizes sparse-file support to achieve O(1) time complexity in create, read, update, and delete (CRUD) operations while supporting arbitrarily small time granularity. We designed and implemented XStore[1] to be lightweight and offer high performance without the need to maintain an index of the time-series data. We conducted a detailed evaluation between XStore and existing best-of-breed systems such as MongoDB and InfluxDB using synthetic data spanning 20-years of 1-second granularity data, totaling 662 million rows of data and 170GB of data.

Index terms:
database, NoSQL, data storage, file systems, data analysis, time-series, extreme, performance, high-performance computing, MongoDB, InfluxDB, XStore

1 INTRODUCTION

Time-series is a successive sequence of data points measured over a time interval. In many popular database systems, time series data are often indexed in time order. In other words, an index of a time-series dataset is often times its *timestamp* field. Time series are quite common in our daily lives, more than we might think. It ranges from IoT (Moisture, heart rate, etc.) to scientific researches (Nuclear simulations, diagnostic data of quantum computing, etc.). However, it has come to our attention that current database systems are not optimized enough to deliver the extreme performance that many scientists sought. That is, submillisecond per *insert/query/delete*. To address the performance gap mentioned previously, XStore was born out of the desire for a **NoSQL** database that can deliver extreme performance on an extreme scale. Furthermore, *XStore* is also made with *memory-efficient*. In other words, *XStore* can deliver extreme performance without occupying all available system memory. We hope that with the performance of *XStore* it gives scientists the ability to utilize *XStore* for their time-series analysis tasks and arrive at an actionable conclusion sooner than ever before.

Correspondingly, XStore's research enables the possibility to expand further into other areas where XStore is currently overarching, including:

In data storage, XStore can realize a gain in further performance boost as well as other related topics such as persistency, convenience, ease of use, etc. Currently, there are several notable applications such as *Network File-Systems (NFS)*, *Parallel File-Systems (PFS)*, and *Distributed File-Systems (DFS)*. Consequently, a potential avenue for XStore to branch out is to develop a distributed file-system such that it is latency-optimized coupled with ease-of-use through single namespace across multiple nodes within its network.

Granted there has been a diverse set of solutions designed to support database operations for time series. It seems like most if not all solutions are designed and operated on a technique that was invented in 1970, *B-Tree* and its permutation, B^+*-Tree*. Specifically, an avenue for XStore to branch out is the support of XStore's data organization techniques to operate on in-memory mode such as *MongoDB In-Memory* storage engine.

Similar, if not the same as the previously mentioned domain, *Database*. XStore can explore several opportunities to further the performance of existing products in the data analysis community. One of the most popular tool in this community is *Pandas* in Python. We hope that with XStore's data organization techniques, the iterative process of time-series data using *Pandas* can be massively accelerated in multiple folds.

1.1 Motivations

In recent years, we have seen an un-precedented growth of data in our daily lives ranging from health data from an Apple Watch, financial stock price data to diagnostic data of nuclcear/rocket simulations. The attached [Table 1] and [Figure 1] in this paper are an example of how typical time series data are stored and presented in table format. It has come to our attention that most if not all existing database solutions are not designed to address the performance gap in data retrieval in an ever-growing number of time series data. In other words, existing systems rely on an idea of distributed systems as a mean to scale its efficiency and performance. Furthermore, existing systems are prone to focus its capability in storing a large amount of data but not iterative time-series analytics. Therefore, with results of XStore, we strive to further the efficiency of analyzing such a massive amount of data while maintaining scalability while maintaining a minimal footprint in system usages as well as guarantee a constant time complexity in *CRUD* operations. In summary, our goal is to explore and strive to address the following but not limited to:

(1) Chrono structure – Current's cutting-edge solutions lack the performance acceleration through the adoption of structured time-series data

[1] https://gitlab.com/lvn2007/XStore

(2) Reliance on Tree – Maintenance of *B-Tree/B⁺-Tree* structures require a sizable amount of memory to maintain an $O(log_n)$ time complexity
(3) Scalability – Lack of support for fine-grain time-series data while scalability is a concern due to its reliance on *Tree* structures in which impacts both performance and system's resources
(4) Index – Time-series databases accelerate its performance through indexes. This incurs a heavy tax on CPU and memory, yet remains far from reaching a *O(1)* time complexity performance

Table 1: TS Table Sample – BTCUSDT Historical Tick Data

Timestamp	Price($)	Qty (BTC)	Trade ID	First tradeID	Last tradeID	Buyer marker	...
2017-09-01 00:01:00.493	4689.89	0.053417	61458	69180	69180	True	...
2017-09-01 00:01:00.530	4689.89	0.064732	61459	69181	69182	True	...
2017-09-01 00:01:00.573	4689.90	0.147065	61460	69183	69183	True	...
2017-09-01 00:01:18.120	4689.90	0.042047	61461	69184	69184	True	...
2017-09-01 00:01:18.210	4689.91	0.115178	61462	69185	69185	True	...
...	...	...	...	...	...	...	...

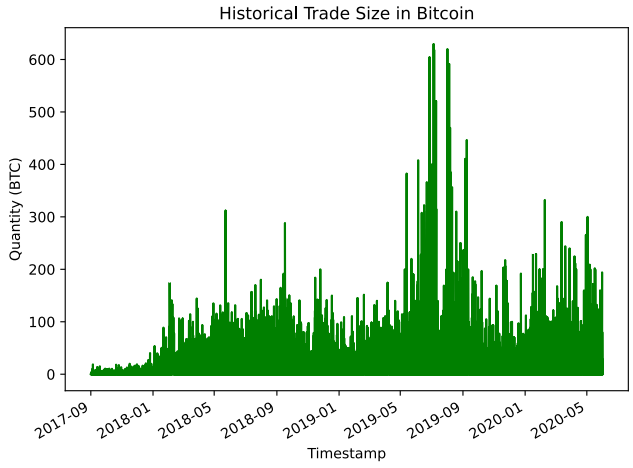

Figure 1: TS Chart Sample – Historical Trade Size in Bitcoin

1.2 Contributions

In this paper, we hope to contribute the following:

- A novel technique that utilizes sparse an order magnitude of time to achieves *O(1)* time complexity in *CRUD* operations
- Design and implementation of XStore, addresses the performance gap in time-series data retrieval without the needs to maintain an index or hash-map while maintaining minimal footprint in system usages
- Enabling the ability to perform iterative time-series analysis at an extreme performance with wide range of use cases ranging from financial backtesting to nuclear/rocket simulations
- Empirical Evaluation results of XStore against the current bleeding edge technologies such as MongoDB and InfluxDB

2 RELATED WORK

InfluxDB is an open-source cross-platform designed and specifically built to handle time series databases. Given that InfluxDB is SQL-enabled, it allows user to effectively query and analyze real-time time series data. While InfluxDB is superior to MongoDB in real-time analytics, its feature-rich functionalities resulted in inferior performance compared to MongoDB. It is imperative to note that InfluxDB is more appropriate for write-heavy workload such as ingesting data from IoT sensors. Moreover, InfluxDB can be scaled horizontally using the existing technique presented in Hadoop [1].

MongoDB is a cross-platform document-oriented database program available from source. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas. It is worth noting MongoDB supports two type of engines: *WiredTiger* and *In-Memory* [2]. The aforementioned storage engines allow MongoDB consumers to have the flexibility to store and consume data that are either performant in space or speed. Additionally, MongoDB can be horizontally scaled and load balancing through the use of *sharding*, a built-in feature of MongoDB. It enables the ability to increase performance while remaining fault-tolerant in a distributed environment.

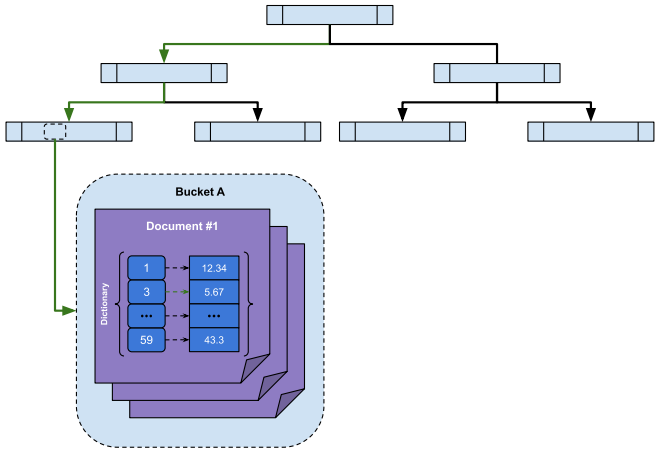

Figure 2: Sample Traverse Path of a B-Tree in MongoDB [3, 4]

Table 2: Bucket Time Span Characteristics[4]

Granularity	Bucket Time Span	Max Elements
Second	1 Hour	3,600
Minutes	24 Hours	86,400
Hours	30 Days	2,592,000

MongoDB and most of other time-series databases employ B-Tree or similar for its storage engine. That is, *CRUD* operations require a tree traversal in which does not only incur logarithmic cost of traversing the tree but also the cost of linearly search through each document to locate a correct target within each document within a bucket. According to [Figure 2] is an example of how a *CRUD* operations can be carried out in MongoDB; a sample traversing path

of a B-Tree can be followed as highlighted in green. Given a target timestamp, MongoDB would need to traverse its B-Tree to locate the appropriate bucket that contains the target timestamp. Once a bucket is located, it would then need to traverse linearly through all of the documents within a bucket to locate the appropriate document that matches the given target timestamp. For example, a time-series collections in MongoDB with a seconds granularity would translate to: A cost of traversing the tree plus the cost of traversing each documents within a bucket. In an event that the data is dense coupled with seconds granularity, it would typically require $\approx 17 - 18$ documents scan within a bucket. In other words, each bucket (seconds granularity) has a maximum amount of 3,600 data points [Table 2], that translates to a scan of $\approx \frac{3,600}{18} = 200$ data points per document to iterate through to conclude a single CRUD operation. Last but not least, as documents are organized in time-ordered bucket, it is important to highlight that the cost of obtaining a particular target with time that resides in the rear of a bucket is much higher than the cost of a target with time that resides in the front of a bucket. Additionally, per MongoDB's manual [4], an incorrect configuration of granularity such as for a data with an arrival interval of minutes, setting it with a granularity of hours would result in a single bucket contains an entire month's worth of data. On the other hand, setting it to seconds would result in a creation of multiple buckets per polling interval. Therefore, such cases of improper configurations lead to a massive performance deterioration.

3 XSTORE SYSTEM ARCHITECTURE

XStore is a time-series specialized key-value store database, a type of a distributed *NoSQL* database program designed to address an ever-growing amount of time-series data. XStore is equipped with a technique that takes advantage of an order of magnitude of time to deliver extreme performance that other *NoSQL* databases lack thereof. The task of performing simulations on time-series data with the goal of deriving an optimized set of values for a target subject is extremely critical. It enables researcher the ability to study and select an optimal parameters for a specific target subject prior to deploy those parameters on a real system. A notorious example of this is the use of backtesting/simulation in finance industry. In summary, the goal of backtesting is to use computing power to perform a series of simulations (or backtesting sessions) to find a set of parameters that historically produce the most profitable trading strategy. In other words, the entire simulation is to try all possible combinations. In each simulation, it digests the input parameter set and runs it against historical time-series data. Finally, at the end of each simulation session, an outcome is exported from each simulation.

According to the example provided above, simulation is a multi-task processes that involve multiple query if data is stored in a database. With that being said, the goal of XStore is designed with the following goal but not limited to:

- Performance – Guarantees *O(1)* complexity for *CRUD* operations
- System's Resource Efficiency – Does not maintain index or hash-map, reduces processor and memory costs
- Scalable – Maintains a constant *O(1)* performance regardless of granularity or size of time-series data
- Persistent – Employs "Store nothing in memory, always on disk" policy

3.1 Data Storage

XStores stores data it ingested as a binary file. Due to the nature of binary files, it allows XStore to effectively perform operations on a file to achieve superior performance in *insert/get/delete*. Importantly, XStore maintains a local key value store for efficiency purposes such that files are opened/closed only once as well as maintaining metadata for XStore's databases.

3.1.1 File and Data Organization

The current implementation of XStore has successfully eliminated the need to invoke *open/close* on each query. Therefore, XStore's binary file operations only require a single *open/close* at *boot/shutdown* respectively.

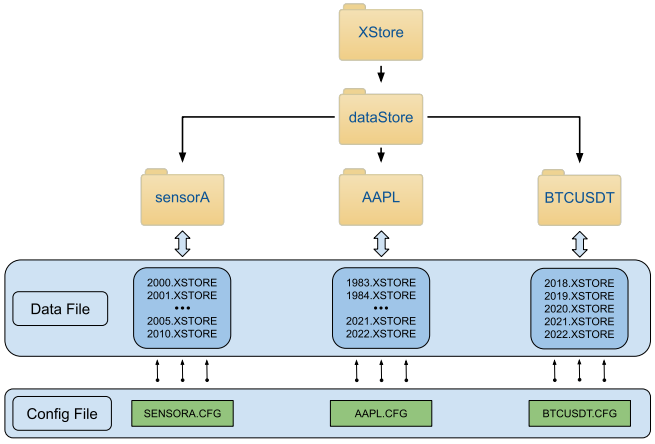

Figure 3: XStore File Organization

An example of how XStore ingests and organizes time series data can be understood using [Figure 3]. As mentioned above, XStore takes advantage of an order of magnitude of time to ensure the scalability and performance of XStore. That is, for each year of data, XStore would perform tasks to organize and store data as a binary file. Note that we will utilize [Figure 3] and [Figure 4] as a visualization aid for our explanation of how XStore stores its data:

(1) Each database created will be stored in a separate directory
(2) Within each directory, it contains all the data file with a naming scheme of **year.XSTORE** i.e., **2018.XSTORE** along with database's metadata as **dbName.XSTORE** i.e., **AAPL.CFG**

Referring to [Figure 4], is an example of how data is being stored by XStore as binary file format. Given that XStore takes advantage of an order of magnitude of time. Therefore, XStore does not need to maintain any form of hash-map. Consequently, this technique enables XStore to guarantee the following:

- No collision due to the nature of time
  i.e., There can't be month 13

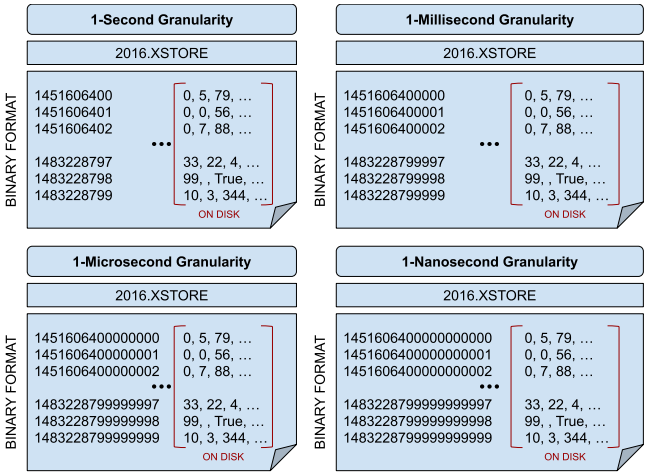

Figure 4: XStore Data Organization

- Logically, each column/item is separated by a comma and associated with a specific timestamp
- Physically, XStore only stores actual inserted data on disk. This is possible due to the support of **sparse file** in many popular file system in Linux. It does not store *timestamp*, represented as **Epoch time** since timestamp can be computed at run-time based on file name and granularity fetched from a database's config file. This has resulted in a drastic reduction in space wasted for storing timestamp
- XStore does not need to maintain an index or map of *Offset* in memory or disk since it is computed at run-time and the computational cost of offset is minimal

3.1.2 *Storage Footprint & Sparse File*

To elucidate the gap in performance outlined in existing database solutions, XStore aims to deliver an extreme performance at the cost of space-efficient. It is important to note that XStore is space in-efficient in a sense that its storage footprint is on par with data stored in a flat file such as *csv* file. It is imperative to note that time-series data tends to have an irregular sequence of time. Therefore, XStore leverages sparse file, a feature that is widely supported by many modern Linux file systems. With sparse file, data stored by XStore appears to be large logically. Physically, only actual data inserted will be stored on disk since holes in a sparse file do not occupy physical disk space [Figure 5] [5]. To further iterate, we argue that given the prices of high performance *SSD*, prices have declined in an increasingly rapid manner over a short period of time [6]. Furthermore, prices for a data center grade *RAM* remain much more expensive, while instituting far smaller capacity compared to a high performance *SSD*. In an event that memory is abundant, XStore's performance can be accelerate beyond the limitation of HDD or SSD by utilizing *RAM disk*. In addition to these arguments, XStore is naturally persistent, given its utilization of persistent storage. Equally important is the fact that XStore leverages persistent storage, which also translates to natural adoption/transition when matched with a machine that is equipped *Intel® Optane DC Persistent Memory* [7] seems to be a Therefore, XStore's choice of leveraging *SSD* to empower its performance is a better trade-off.

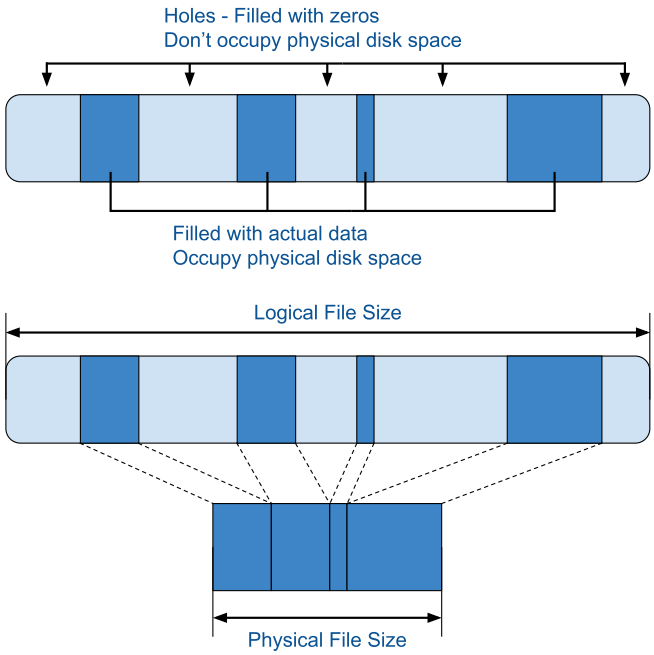

Figure 5: Sparse File [5]

XStore has shown promising results in *CRUD* operations and it is important to note that XStore performs all these operations on persistent storage. Therefore, it is imperative for us to address XStore's storage consumption. According to our analysis, XStore occupies approximately the same amount of storage space and is on par as if data is being stored in a flat file format such as *csv* file. To summarize, by default, XStore posses the following:

- Relies on sparse file – Enable the ability to only physically occupy space on disk for actual data
- 128 bytes per row – Adjustable
- Unlimited column in a row
- Data is stored in binary file format without compression

Despite the benefits of sparse file, there are various trade-offs. One of the major concern of utilizing a sparse file is disk fragmentation. While disk fragmentation pose little to no threat to storage medium such as SSD, it remains a major cause of an increase/reduction in latency and throughput of a spinning hard drive [8]. Consequently, on most if not all HDD, we observe a significant difference in performance between sequential vs. random workload in a sense that sequential workload yields a much higher performance than random workload due to latency cost of moving a disk spindle [9]. With that being said, for XStore case, it is imperative that XStore maintains its data in disk blocks in a contiguous order. In [Figure 6] depicts a scenario of which fragmentation can occurred in a sparse file can pose a major threat to the performance of XStore. Although, it is worth noting that the fragmentation issue does not only happen specifically due to the use of a sparse file but also occur in regular file due to concurrency operations on a file.

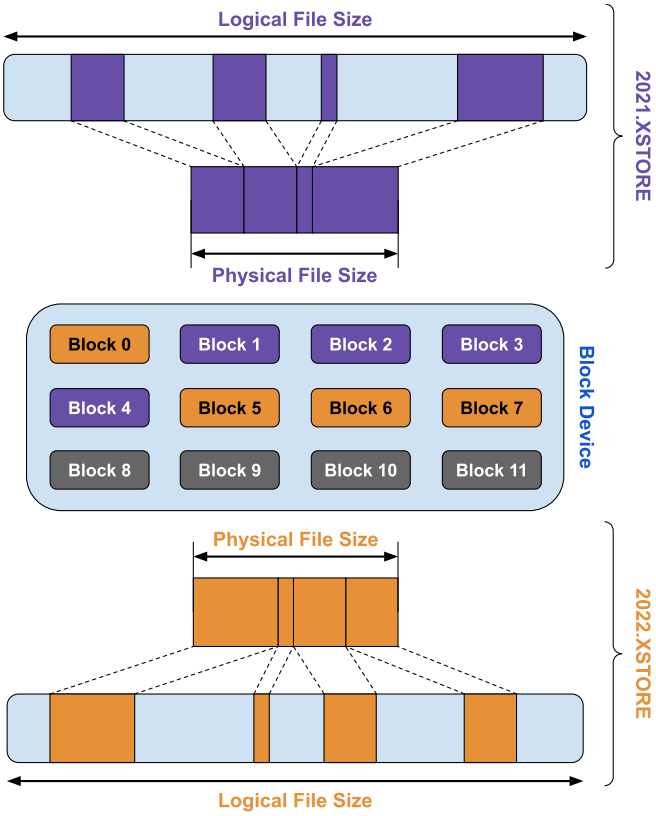

Figure 6: Sparse File – Fragmentation

Notice that for sparse file named 2021.XSTORE (Purple) in [Figure 6] resides in a storage medium. The blocks allocated for 2021.XSTORE is block 1 to block 4 in a contiguous fashion. This indicates that there is no fragmentation for for 2021 with and data is being stored in a chronological-order. Given the contiguous order of blocks allocated, a read operation of an entire data of 2021.XSTORE can be carried out sequentially by having the disk spindle to start from block 1 and keeps reading until the end of block 4. Therefore, in this scenario of file 2021.XSTORE does not suffer from performance degradation caused by disk fragmentation.

On the other hand, for file 2022.XSTORE (Orange) in [Figure 6]. Notice that $\frac{1}{4}$ of the contents of file 2022.XSTORE is being stored in block 0. While the remaining of the block is allocated contiguously from block 5 to block 7. Unlike file 2021.XSTORE, in order to read all contents of file 2022.XSTORE, a disk spindle will need to first start at block 0, read an entire block. Then, proceed with moving the disk spindle again at block 5 then starts reading sequentially from block 5 to block 7.

An obvious strategy of mitigating disk fragmentation is to pre-allocatate a certain amount of file blocks. However, this strategy defeats the purpose of leveraging a sparse file; that is, allocate on as needed-basis such that actual space taken will only be occupied by actual data.

Alternatively, given that time-series data tends to arrive in a chronological order, it helps alleviate some of the concern of data not being stored non-contiguously. However, the fragmentation issue persists in an event of file 2021.XSTORE and 2022.XSTORE are being written concurrently. Even though the OS can delay sync to disk and try to arrange the inserting data in a way such that data stored on disk as a contiguous blocks. The allocated blocks of both files are likely to be interleaved; such as an insert-heavy workload that limit the OS from caching a large chunks of data to delay sync to disk. Therefore, another strategy is to perform disk de-fragmentation. As mentioned in [10], a defragmenter such as *janusd* can be employed to address the impact of disk fragmentation to performance while maintaining a low overhead.

3.1.3 *Offset Algorithm*

In this section, we would like to introduce and discuss an offset computation algorithm that XStore uses to achieve a consistent *O(1)* performance. Our algorithm is rather simple but highly efficient due to XStore leverages an order of magnitude of time to its advantage. It can be understood in the following steps:

(1) Get *yearValue* from file name
yearVal = 2018 because of 2018.XSTORE
(2) Assumptions:
- monthVal, dayVal, hourVal, ... starts at 0
- We have: baseDT = 01/01/2018 00:00:00

(3) Assume that the target is: 01/02/2018 00:00:02
timeDiff = targetDT - baseDT
timeDiff = 01/02/2018 00:00:02 - 01/01/2018 00:00:00
timeDiff = 86,402 (seconds)
(4) Formula:
$Offset = timeDiff \times rowLength$
Note: *rowLength* resides in *config file* of a database

This algorithm enables XStore to deliver performance at an extreme scale regardless of granularity of any time series dataset. According to the above algorithm, it only involves simple arithmetic computations to derive an **Offset**. In other words, with the use of *Offset*, XStore has the ability to *open* a binary file and *seek* directly to a specific location in a binary file; bypassing the needs of maintaining an index or hash-map of the location of stored data. Therefore, *CRUD* operations is guaranteed to be within the constant time complexity of *O(1)* at any given time, regardless of the granularity or size of a time series dataset.

3.2 Architecture Layers

3.2.1 *Outer Layer – Networking*

XStore's networking stack is fully powered by *ZeroMQ*, a high-performance messaging library coupled with *Protobuf* as its serialization protocol. Given that the ZeroMQ library contains an exhausted list of supported programming languages, it has enabled one to consume XStore independent of a specific programming language.

Per [Figure 7], an arbitrary *Client* can connect to XStore using ZeroMQ. Once connected, a client can interact with XStore by sending a specific request. Given that XStore's server is designed according to ZeroMQ's Router/Dealer pattern. This results in the ability of XStore to process any arbitrary amount of clients in a multi-threaded environment. For each client's request, it will be received an intermediary instance called *ZMQ Router*. A *ZMQ Dealer* acts as back-end of XStore such that it attempts to resolve each

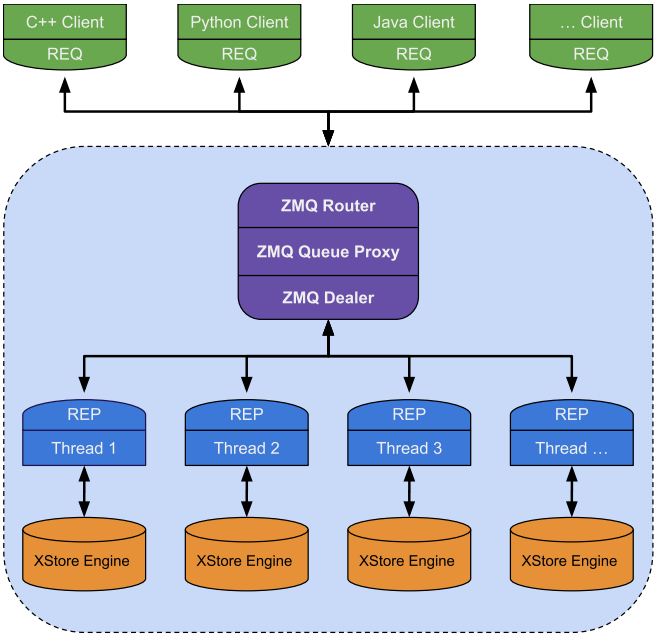

Figure 7: XStore Networking Layer

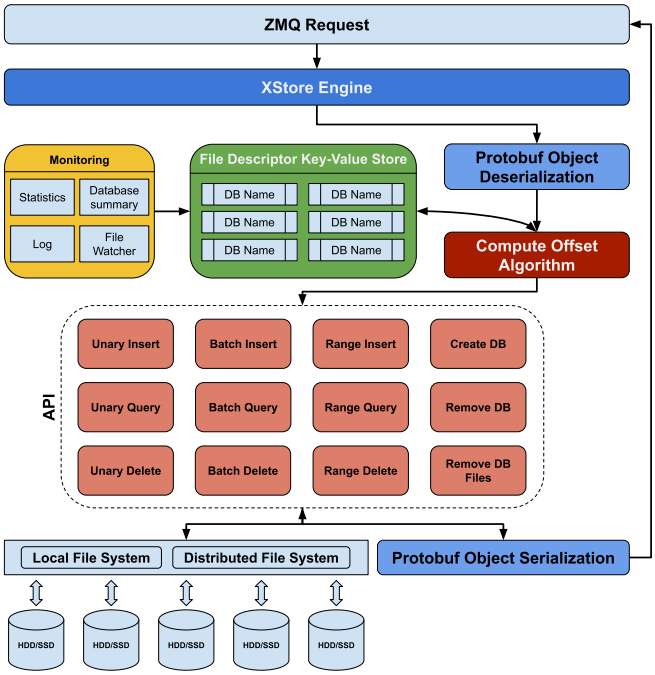

Figure 8: XStore Engine

request in *ZMQ Queue Proxy* as fast as possible. Within XStore's server, each worker thread has the responsibility to process and dequeue requests from ZeroMQ's message queue as fast as possible using *XStore Engine*. Upon each processed request, it will return the response back to the coordinator server to be routed back to the client. Note that each thread in this context has their own *XStore Engine*; which conveniently makes XStore Engine thread-safe without the needs of thread coordination. It is important to note that the communications between *client/server* involve the process of *serializing/deserializing* the Protobuf object.

3.2.2 *Inner Layer – XStore Engine*

In [Figure 8], the inner layer of XStore architecture can be understood as a high-level overview of an entire XStore eco-system. Given that each thread will govern its own *XStore Engine*, therefore, each instance of XStore will first begin with initialization. The initialization step involves administrative operations such as keeping track of statistics, database summary, log, etc. To achieve single open/close, *XStore Engine* will manage its private key value store that contains all available file descriptors of files that XStore manages. Previously mentioned, each request message is serialized and deserialized using *Protobuf*. Upon deserialization of the message from a Protobuf object, XStore will proceed with computing the offset prior to performing actual actions in XStore's database including but not limited to: *unaryInsert, batchInsert, unaryQuery, rangeQuery, etc.*. In our current implementation, we have designed such that the aforementioned operations treat files and directories as if it is stored locally. At this point, it is important to note that in the near future, we might design a distributed file-systems that focuses into managing files in a low-lantecy demanding applications. As soon as data has been performed on disk, *XStore Engine* will then return the result of the operation back to the appropriate *ZMQ request* target.

3.2.3 *API*

XStore is currently a new project at the DataSys laboratory, circulated internally prior to publication to the public. Therefore, it is still immature and would require a tremendous effort to become a production-ready non-relational database. Prior to the below operations, a database must have been initialized through API called **createDB**. This API call will perform the following:

(1) Create a directory for a database
(2) Configure *config file* with the following information:
- Database name
- Granularity
- Row length

With that being said, as of now, XStore is currently supporting the following APIs:

Unary Insert

(1) Locate DB directory & Parse inserting epoch time
(2) Fetch DB's metadata from either internal metadata key-value store in-memory or disk
(3) Enforce time granularity such that configured granularity (Fetched in previous step) should be equal to the inserting granularity
(4) Check if *year*.XSTORE exists. If not, proceed. Otherwise, skip to step (5)
(5) Create sparse file with n number of elements with each element has length of *rowLength* in accordance to database's metadata fetched in step (2)
(6) Compute *Offset* using *Offset Algorithm*

(7) Seek to exact location of target time based on *Offset*
(8) Write data into specific time location to binary file
i.e., 01/01/2018 00:00:01 yields offset = $1 \times rowLength$ in 2018.XSTORE that has a granularity configured as *SECONDS*
(9) Return insert status

Unary Query – In this API call, it is the same as *Unary Insert* from step (1) to (2) then proceeds as following:

(1) Enforce time granularity such that configured granularity should be equal to the target query granularity
(2) Compute *Offset* using *Offset Algorithm*
(3) Seek to exact location of target time based on *Offset*
(4) Read *n* bytes (As specified by *rowLength* in *config file*) worth of data
(5) Return target query data

Batch Insert – In this API call, it is the same as *Unary Insert* from step (1) to (2) then proceeds as following:

(1) Sort collections of inserting data
(2) Enforce time granularity such that configured granularity should be equal to the inserting granularity
(3) Organize inserting data by *Year*
(4) Loop through organized data. On each loop, computes offset then invoke seek and finally write individual inserting data
(5) Return insert status

Batch Query – In this API call, it is the same as *Unary Insert* from step (1) to (2) then proceeds as following:

(1) Sort collections of target timestamp
(2) Enforce time granularity such that configured granularity should be equal to the target query granularity
(3) Organize query target by year for all target timestamps in a collection
(4) Loop through organized year. On each loop, computes offset then invoke seek and finally read individual query target data
(5) Return a collections of query target data with its respective timestamp

Range Insert – In this API call, it is the same as *Batch Insert* with an exception that in range insert, it assumes that the data to be inserted is sorted. Therefore, it does not need a sort routine.

Range Query – In this API call, it is the same as *Batch Query* with an exception that in range query, it performs a sequential read starting from a start timestamp to a given end timestamp in a chronological order. Note that upon this API is called, it performs a query validation to ensure a range is valid i.e., start timestamp $\leq$ end timestamp. Therefore, it does not need a sort routine.

It is imperative to note that we are currently implementing an exhausted list of features that can potentially further the performance and usability of XStore. Several notable additions could be: aggregation (sum, max, min, avg) on range of data, search capability beyond just timestamp, etc.

4 PERFORMANCE EVALUATION

4.1 Testbed

For our experiments, we have configured a cluster of 4x baremetal *compute_haswell_ib* instances on *Chameleon Cloud* [11] with the following hardware configurations for each machine:

- Dual socket Intel® Xeon® E5-2670 v3 Processor – Haswell @2.30GHz (12 cores, 24 threads)
- 8x 16GB (128 GB) of DDR4 2,133 ECC Registered RAM
- 1x Seagate ST9250610NS SATA 7200 RPM HDD Capacity: 250GB with 64MB cache
- Broadcom NetXtreme II BCM57800 1/10 Gigabit Ethernet
- Ubuntu 20.04.5 LTS
- Filesystem: EXT4
- g++ 8.4.0
- C++17
- cppzmq v4.8.1
- Protobuf v3.18.1
- MongoDB v6.0.3

4.2 Microbenchmark – Network Latency

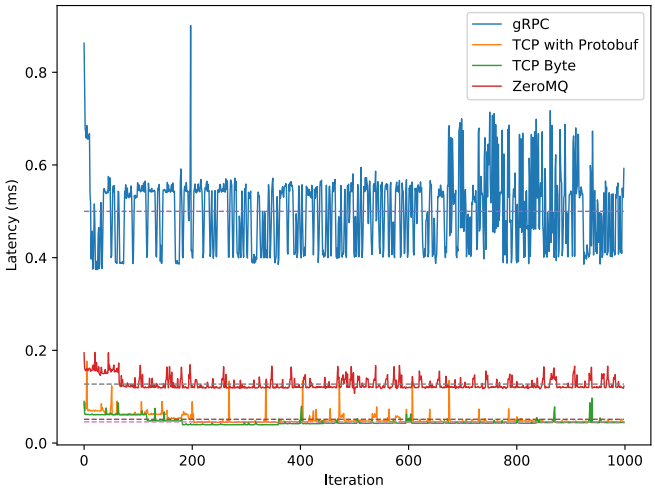

Figure 9: Microbenchmark – Network Latency

Prior to a discussion about how XStore performs against other high-performance database solution. It is imperative that we understand that these results are preliminary results of XStore. Additionally, we would like to perform a network latency benchmark to further familiarize ourselves with the performance characteristics of *ZeroMQ*. In the previous XStore development iteration, we initially opted to enable networking for XStore using *gRPC*. However, we have decided to permanently switch to *ZeroMQ* due to its performance, building blocks as well as its support for various type of networking topologies. In this section, we have performed network latency benchmark on the following:

- gRPC
- Linux TCP Socket transport bytes
- Linux TCP Socket transport Protobuf object
- ZeroMQ (REQ/REP pattern)

According to [Figure 9], we noticed that *ZeroMQ* is far superior compared to *gRPC*. Consequently, the overhead of using *Protobuf* on top of the Linux TCP socket is minimal compared to the benefits of packing messages using *Protobuf*. Additionally, we determined that it is imperative that XStore sacrifice some performance in order to gain additional features and ease of use by employing *ZeroMQ* over *Linux TCP Socket*.

4.3 Evaluation Configurations

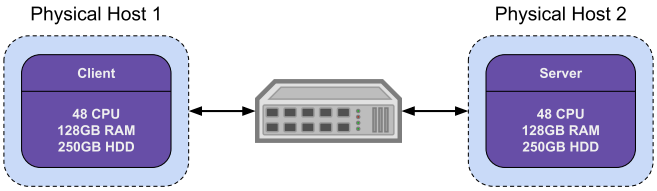

Figure 10: Benchmark Configurations

The evaluation of both *XStore* and *MongoDB* is conducted on a pair of two separate baremetal instances connected via a gigabit ethernet switch [Figure 10]. That is:

XStore:

- Physical host #1: XStore Client in Python
- Physical host #2: XStore Server

MongoDB:

- Physical host #3: MongoDB Client in Python
- Physical host #4: MongoDB Server

In *MongoDB*, we have configured it with the most optimal configurations to operate on time-series data as well as equipped data with a secondary-index [12]. Additionally, caches are cleared in the beginning of each benchmark session of both *XStore* and *MongoDB*.

4.4 Baseline Measurement

```
--- . (ext4 /dev/sda3) ioping statistics ---
99 requests completed in 574.5 ms, 396 KiB read, 172 iops, 689.3 KiB/s
generated 100 requests in 1.65 min, 400 KiB, 1 iops, 4.04 KiB/s
min/avg/max/mdev = 256.6 us / 5.80 ms / 29.0 ms / 4.66 ms
```

Figure 11: Read Latency using O_DIRECT

Throughout the subsequent experiments, to demonstrate the performance of XStore remains resilient even under an unfavorable hardware configurations such as HDD while operating on a dataset that is $\geq$ 170 (GB) in size; far exceed the test machine's memory capacity. We have acquired a baseline measurement of disk latency such that it averages around 5.80 (ms) per read operation [Figure 11]. This baseline measurement was acquired using a Linux tool called *ioping* with *O_DIRECT* flag to bypass Linux kernel's caches.

4.5 Benchmark – Unary Insert

In this first benchmark, we compare the insertion latency performance including networking cost between XStore and MongoDB in unary insert. That is, for each of the 1,000 iterations, client invokes a single insert operation from client to server with a unique inserting data of $\geq$ 264 bytes. Upon each successful insert, a server will return insert status to client as a notification indicates the result of that particular insert operation. It is important to note that the randomized timestamp is ranging from the start timestamp to the end timestamp across the entire 170 (GB) dataset.

Table 3: Summary Statistics – Unary Insert (ms)

	Min	Max	Mean	Std Dev	Avg. Throughput
XStore (SEQ)	0.55	0.87	0.74	0.04	1,352.65
XStore (RAND)	0.59	0.93	0.76	0.05	1,307.44
XStore (Ram Disk)	0.55	0.96	0.75	0.05	1,336.26
MongoDB (SEQ)	0.78	1.95	1.34	0.24	744.98
MongoDB (RAND)	0.77	1.10	0.93	0.06	1,070.60
MongoDB (2^{nd} Index)	0.86	19.06	1.45	0.86	688.02
MongoDB (Ram Disk)	0.85	31.27	1.41	1.03	711.08

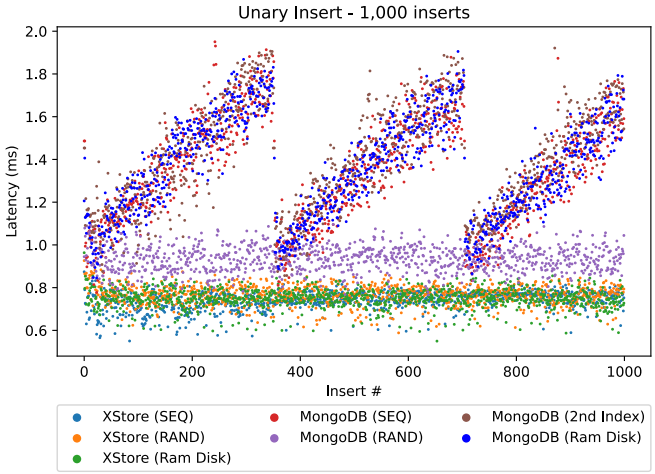

Figure 12: Benchmark – Unary Insert
Note to MongoDB: Replaced 25 observations with latency $\geq$ 2 (ms) with its respective average

According to [Figure 12] along with [Table 3], we observed that compare to MongoDB, XStore took relatively less amount of time to establish a connection between client and server. Although in MongoDB's tests, we observed a multiple occasions of spike in latency that are well above 20 (ms). However, this is likely caused by the initial handshake between client and server and/or the initialization required from MongoDB. In the entire test iterations, XStore excels in performance for both sequential and random insert workloads, notably, a consistent average latency of 0.74 (ms) and 0.76 (ms) per round-trip across the wire; translates to an average throughput of

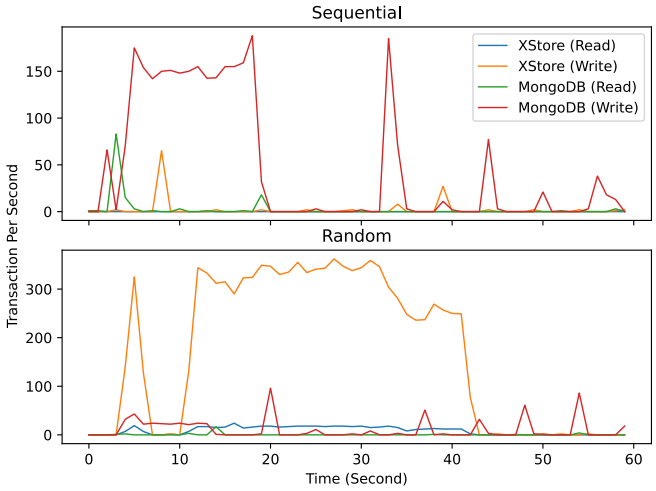

Figure 13: I/O Activity (Unary Insert)

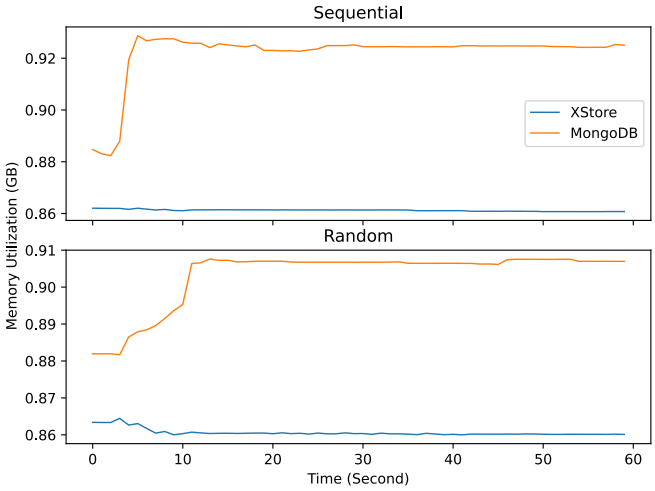

Figure 15: Memory Activity (Unary Insert)

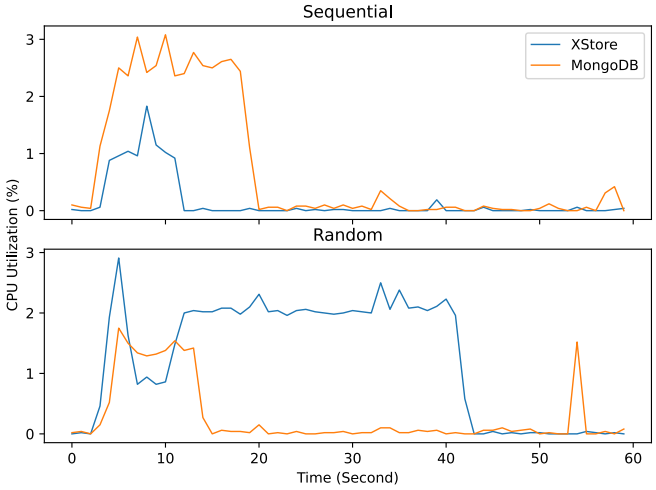

Figure 14: Processors Activity (Unary Insert)

1, 352.65 and 1, 307.44 rows inserted per second respectively. While in MongoDB, yields an average of 744.98 and 1, 070.60 rows inserted per second for sequential and random workload respectively. It is interesting to observe that both XStore and MongoDB are able to deliver such low latency on average. This is stemming from the fact that both solutions employ delaying write to its advantage. That is, instead of sync each individual insert to disk, it delays the sync activity and sync to disk at a later time as an attempt to improve performance/efficiency. Given that XStore defers the sync to disk responsibility to the OS and MongoDB delays its write. Especially in MongoDB, it delivers a better performance in random insert compare to sequential insert. Consequently, it explains how both XStore and MongoDB is able to deliver such low latency that is on-par with latency performance of ram disk.

Per [Figure 12], MongoDB's latency for sequential workload is worsened upon each subsequent inserts. We observed that in sequential workload, MongoDB performance deterioration ends on every 380^{th} insert. It hinted that MongoDB organizes each time series data into a bucket and on the $\approx 380^{th}$ insert, MongoDB flushes data resides in a bucket to disk as batches and the actual writing to disk is carried out asynchronously a batch has been filled-up to a certain level as a way to improve efficiency. Nevertheless, per [Figure 12], we observe that with the use of bucket, as a bucket is filled with data, it deteriorates performance as the bucket becomes more dense with data. It coincides well with our discussion of [Figure 2] as the cost of traversing bucket rises in conjunction with rise in documents per bucket. Therefore, MongoDB suffers from time taken to locate a specific location of where to insert a time series data within a bucket. Moreover, MongoDB (2^{nd} Index)'s performance meets our expectation. That is, it will take longer for MongoDB (2^{nd} Index) to conclude a single insert since there is an overhead of maintaining a secondary index. In the current implementation of XStore, it involves the use of both fopen/fclose which ensure the data remains in buffer to be written to disk. Upon further investigation, per [Figure 13] and [Figure 14], we notice that for sequential workload, MongoDB made almost twice as much write requests to the hard drive as well as consuming twice as much processor power than XStore. Coupling with [Figure 15] shows that XStore uses little to no memory compare to MongoDB. It enables us to conclude that XStore is superior in performance than MongoDB in sequential workload while efficiently utilize system's processor and memory.

While for random workload, per [Figure 13], XStore made a significant more write requests compare to MongoDB. It is because XStore organizes time series data by year i.e., 2017.XSTORE, 2018.XSTORE, etc. Thus, a random insert would cause a new sparse file to be create if a given year is not yet existed. An involvement of multiple large sparse file would also cause the OS to trigger additional calls as the costs of maintaining a complex sparse file such as filesystem's metadata updates to maintain logical and physical offset of a sparse file, block allocation, etc. In constrast to XStore's I/O pattern, MongoDB made little to no write calls to the disk drive.

At the same time, [Figure 15] shows that XStore's memory remains flat for the entire duration of the experiment while MongoDB memory does not only increase but also remain flat after the experiment has been concluded. Given the data provided in [Figure 13] and [Figure 15], we realized that MongoDB does not sync the inserting data to disk but keep them in memory. This means that MongoDB is less persistent than XStore. Thus, in an event a power outage, the inserting data will be irrecoverable. According to [Table 3], even though both XStore and MongoDB delay sync to disk and the average latency is 0.76 (ms) and 0.93 (ms) in XStore (RAND) and MongoDB (RAND) respectively. In other words, we can safely assume that both solution temporarily stores its data in memory and sync with disk at a later time. It is evidently clear that our technique is more efficient in figuring out a specific location to insert without the needs to use bucket. Therefore, we conclude that XStore is more effective than MongoDB in random workload while utilize minimal system's resources.

4.6 Benchmark – Unary Query

Table 4: Summary Statistics – Unary Query (ms)

	Min	Max	Mean	Std Dev	Avg. Throughput
XStore (SEQ)	0.50	20.70	0.76	0.63	1,309.49
XStore (RAND)	3.29	77.52	13.89	5.57	71.97
XStore (Ram Disk)	0.50	1.53	0.67	0.06	1,494.94
MongoDB (SEQ)	1.04	74.71	1.55	2.32	644.86
MongoDB (RAND)	2.69	163.24	37.91	11.78	26.38
MongoDB (2^{nd} Index)	2.12	364.71	75.27	34.36	13.29
MongoDB (Ram Disk)	1.04	7.90	2.93	1.39	340.97

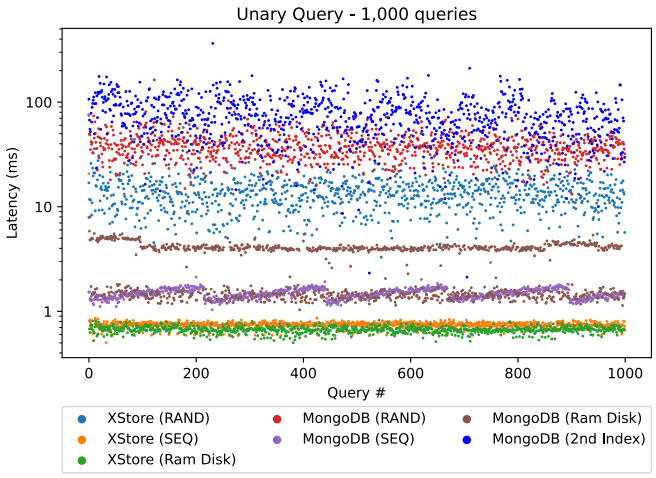

Figure 16: Benchmark – Unary Query

Similarly, in this second test, we performed a query latency performance test including round-trip networking cost between XStore and MongoDB. In other words, how long does it take for each solution to complete a single query given a random and sequential timestamp. Throughout this benchmark, we have performed a total of 1,000 unary query requests from client to server with each query request contains a timestamp that is ≈ 8 bytes long indicates client's target timestamp. Upon each unary query request, a server will return to client a pair of *timestamp* (≈ 8 bytes) along with data associated with that particular timestamp (≥ 256) bytes. It is imperative to note that for the random workload, the randomized timestamp is ranging from the start timestamp to the end timestamp across the entire 170 (GB) dataset. While in sequential workload, a target timestamp is incremented by one second upon each subsequent query.

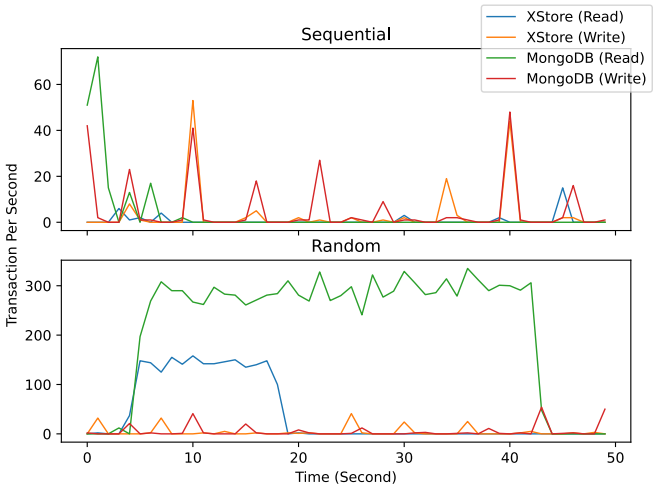

Figure 17: I/O Activity (Unary Query)

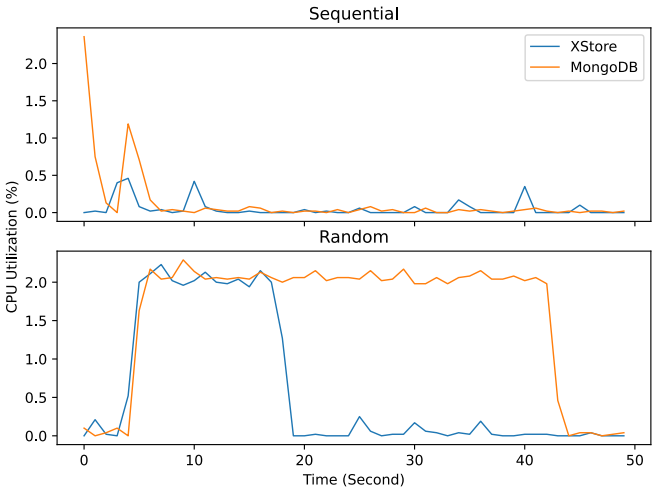

Figure 18: Processors Activity (Unary Query)

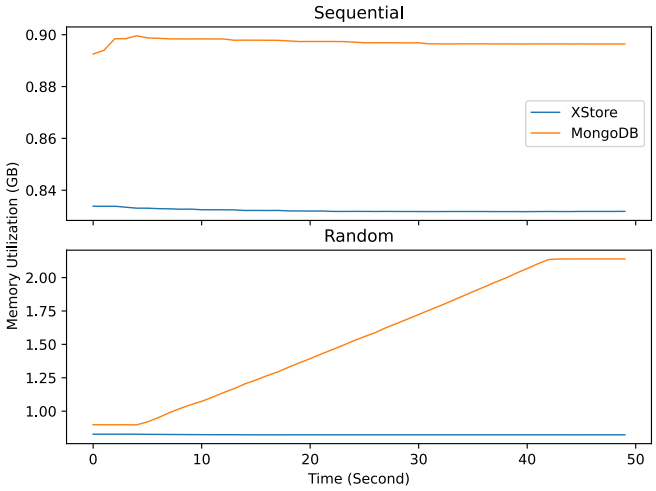

Figure 19: Memory Activity (Unary Query)

According to [Figure 16], we observed that in sequential workload, XStore outperforms MongoDB by ≥ 2x; translates to a throughput of 1, 309.49 rows per second. On average, XStore took 0.76 (*ms*) to complete a single query, while MongoDB took 1.55 (*ms*). Interestingly, MongoDB took almost 3x longer to establish a connection between client and server; indicates by 20.70 (*ms*) and 74.71 (*ms*) in [Table 4]. On the other hand, given such low average latency of the two solutions, we realize that any subsequent queries after the initial query must have been hot-queries and its file descriptor is already resided in OS caches. Most importantly, per [Figure 16], we notice that MongoDB (SEQ) latency increases in a very seasonal fashion. This enables us to empirically validate our analysis MongoDB's bucket usage (200 documents per bucket). That is, it is much faster to locate a single item that is in the front of a bucket than in the back. Upon further investigations of the behavior of the underlying two solutions under sequential workload. According to [Figure 17], while it is not obvious due to noise from other activity within the OS, we observed that both solutions made a relatively same amount of I/O calls. At the same time, it is interesting to see that prior to the benchmark, [Figure 19] depicts a clear conclusion that MongoDB consumes more memory than XStore at idle. Additionally, in [Figure 18], the data also points out that MongoDB consumes more than twice the CPU resource of the host machine.

Prior to generating a secondary index for MongoDB, we expect the query result will improve since there is a secondary index that could further the performance of MongoDB [13]. Surprisingly, the benchmark results for MongoDB (2^{nd} Index) (Random workload), indicates a performance deterioration when compare with regular MongoDB without secondary index. Upon further analysis of [Figure 16] for random workload, despite the fact that there occurred several spikes in latency for MongoDB, its performance is relatively stable across 1, 000 random unary queries. Additionally, we observed a significant latency increase in establishing connection between client and server for both XStore and MongoDB comparing to sequential workload. This is stemming from the fact that in this test, client side of both solution is querying against a server that maintains a dataset of ≥ 662 million rows. Despite the fact that with the current implementation of XStore requiring fopen/fclose for each individual query, we affirm that the performance can accelerate a bit further. On the contrary, the performance of XStore shown in [Figure 16] and [Table 4] indicates an average look-up latency of 13.89 (*ms*) which is a magnitude worst than its sequential workload. This is due to the fact that in random workload, the movement cost of a disk spindle is extremely expensive; the seek cost (read) for our testbed is 8.5 (*ms*) according to the advertised seek time published by Seagate, the hard drive manufacturer of our testbed [14]. At the same time, it is worth noting that for random workload, XStore outperforms MongoDB by over 2.5x in an average latency to conclude a single random query. Per [Figure 17] shows that XStore does not only made twice as less of I/O calls as MongoDB but also was able to deliver the queried data in a much shorter period of time. This results in a drastic difference in I/O calls of the entire duration of the benchmark between XStore and MongoDB. Similarly, per [Figure 14], we observe a similar pattern in which MongoDB occupies the testbed's CPU for a longer duration. Despite MongoDB occupies system's resources for a longer period of time, in [Figure 19] depicts a much clearer picture. According to data in the figure, we observed that the memory usage of MongoDB keeps increasing linearly upon each subsequent query in the entirety of 1, 000 unary queries. While, XStore remains unchanged in memory usage. This figure coupled with previous figures enable us to conclude that even though MongoDB attempts to harness the memory at its disposal, due to its internal structure of storing data, it is inferior to XStore in both system's resources as well as raw performance. With that being said, we have concluded that with our novel technique, it has enabled us to achieve a performance on a hard disk while delivering a major performance improvement in latency when comparing to other state-of-the-art system such as *GorillaDB on flash* [15].

4.7 Benchmark – Range Insert

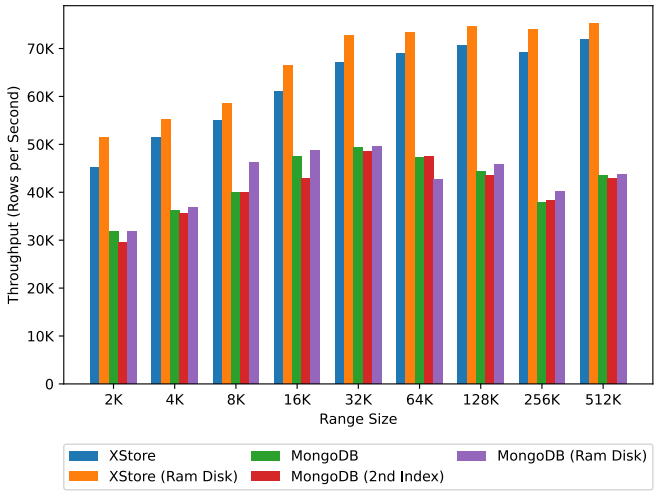

Figure 20: Benchmark – Range Insert (Average Throughput)

In this test, we have performed and averaged 10 runs of range insert with various range sizes ranging from 2, 000 to 512, 000 per batch. That is, instead of performing 2, 000 individual unary inserts to complete a 2, 000 inserts. In range insert, we would pack all 2, 000 inserting items together in a single *ZMQ* message then dispatch it from client to server. It is important to note that range insert assumes the inserting data is already organized in a chronological order. Within each of the aforementioned range size in [Figure 20], it contains N amount of elements to be inserted where each individual element has a timestamp ($\approx$ 8 bytes) along with $\geq$ 256 bytes worth of data associated with it. Doing this would allow us to re-coup losses in performance due to transporting messages over the wire. Additionally, given XStore's implementation of range insert, it writes sequentially to disk for any given range of timestamps along with its data. Therefore, allowing XStore to harness the performance of sequential write.

According to [Figure 20], we observe that XStore delivers a much higher throughput than MongoDB and MongoDB (2^{nd} Index) by $\approx$ 48% and $\approx$ 52% respectively. Similarly, as observed in unary insert, we noticed a commonality that MongoDB (2^{nd} Index) delivers a lower throughput than regular MongoDB. It is interesting to notice that the performance of MongoDB starts to deteriorated once range size is above 32, 000. On the other hand, XStore's performance is able to grow in-conjunction with growth in range size. Nevertheless, XStore's throughput seems to plateaued once range size reaches 128, 000. The performance difference is largely contributed by the fact that our technique enables XStore to only require a minimal effort to locate and insert elements into a storage medium while harnessing the efficiency of both sequential write workload as well as transporting packet across the network.

4.8 Benchmark – Range Query

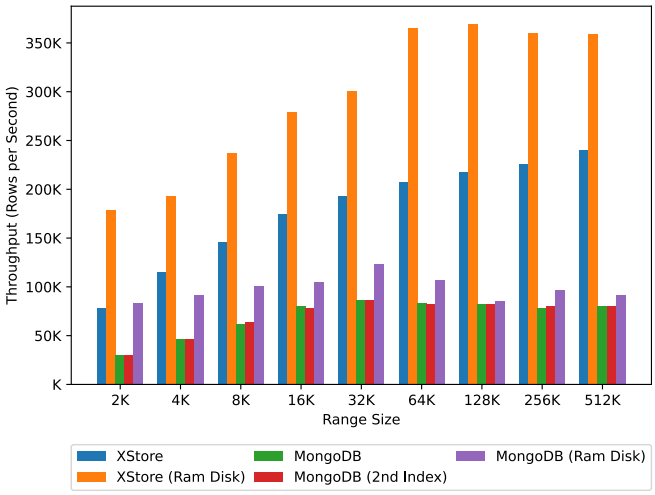

Figure 21: Benchmark – Range Query (Average Throughput)

In contrast to unary query, the range query test demonstrate a performance of querying a range from any given start timestamp to end timestamp. In other words, instead of performing 2, 000 individual query, range query would allow us to query a range of data. This is achieve by dispatching a single request from client to server where client provides a start and end timestamp of a range along with its desired database name to be queried from. Similar to range insert, the range query call enables the ability to re-coup performance losses in transporting messages back and forth multiple times while harnessing the benefit of sequential read. The test was performed on various range sizes ranging from 2, 000 to 512, 000. That is, 2, 000 indicates range from a given start timestamp to an end timestamp that sums up to 2, 000 i.e., range of timestamp from 1577844623 to 1577846623 contains 2, 000 individual data points where each data point ($\geq$ 256 bytes) is associated with a unique timestamp.

Per [Figure 21], we noticed the gap in throughput between *XStore* and *MongoDB* ranges from $\approx$ 2x to $\approx$ 3x. Comparably to previous tests, we observe that *MongoDB* (2^{nd} *Index)* delivers a performance that's on-par with regular *MongoDB*. Simultaneously, we observe a similar in throughput performance as observed in range insert. That is, the performance of *MongoDB* and *MongoDB* (2^{nd} *Index)* deteriorate once range size is above 32, 000. The fact that MongoDB (2^{nd} Index) delivers the same amount of performance for read workload is quite interesting given our initial expectation is that with a secondary index, we should at least observe some improvement. Unlike in previous test, insert throughput plateaus once range sizes is above 128, 000, *XStore* is able to demonstrate its ability to consistently deliver a higher query throughput as range size increases. Considering *XStore's* query throughput, it has exemplified the capability of conducting time-series analyses at a rapid rate where the effectiveness of fetching a large range of data then apply analyses on client's end is a concern.

4.9 Benchmark – Batch Insert

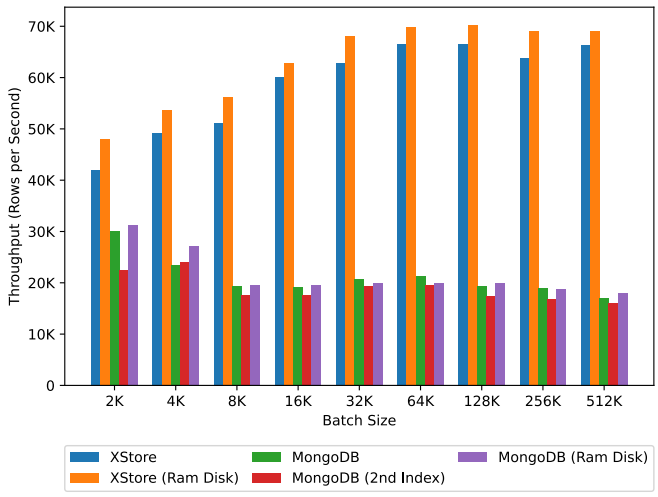

Figure 22: Benchmark – Batch Insert (Average Throughput)

Comparably to range insert, batch insert is very much alike. Nevertheless, a major difference is that in this test, we introduce random workload as a batch. In other words, for every batch size mentioned in [Figure 22], we would pack a single *ZMQ* message that consists of N elements corresponding to N batch size where

each element in a batch is not organized in a chronological order. Therefore, it is a batch of random write workload.

According to [Figure 22], we observe a significant impact in performance between sequential vs. random workload. Notably, as batch size increases, *MongoDB* average throughput decreases. Concurrently, the performance gap between *XStore* and *MongoDB* also increase in conjunction with increase in batch size. In addition, we also observe that *MongoDB* (2^{nd} *Index)* does not only fail to improve performance but also worsen it. Comparing this test with range insert, the performance gap between *XStore* and *MongoDB* is immensely apparent. Across all batch sizes, *XStore's* delivers $\geq$ 2x more throughput on average than its counterpart. Such a large gap in performance between the aforementioned solutions is stemming from the fact that *XStore* attempts to sort its inserting data prior to perform write operations to disk. The sorting routine prior to performing write is crucial since it allows *XStore* to attempt to organize and convert random write workload into a sequential write workload. Per [Figure 20] and [Figure 22], we observe a similarity in terms of performance. However, sort routine in batch insert incurs a cost, which we deem is a good trade-off in regards to the performance gain from it.

4.10 Benchmark – Batch Query

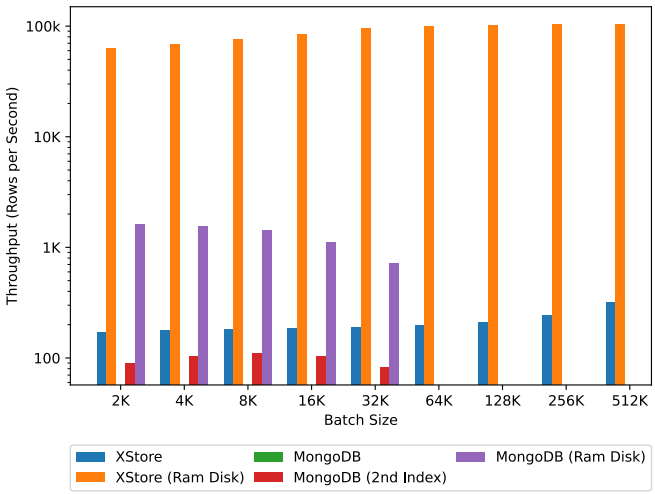

Figure 23: Benchmark – Batch Query (Average Throughput)

In this last test, it is almost identical to range query but with an exception that the workload in this test is a batch of random query. According to [Figure 23], among MongoDB and MongoDB ($2^{nd}Index$), it is interesting to notice that without a secondary index, MongoDB is inoperable for this workload. While equipped with a secondary index, we observed a performance degradation once batch size grows beyond 8,000 rows. More importantly, MongoDB ($2^{nd}Index$) becomes inoperable once the batch size is above 32,000 rows. This is due to the limitation in BSON file size of MongoDB (Codename: BSONObjectTooLarge).

Upon further investigation of MongoDB performance's behavior on ram disk, [Figure 23] depicts a major improvement of both XStore and MongoDB in average throughput. To be specific, operating on ram disk portray a performance gap such that MongoDB is only able to deliver at most $\approx$ 1,500 rows per second; while, XStore delivers a throughput of at most 90k rows per second. To further re-iterate, XStore poses a slight growth in performance from 2k batch size and upper is due to the benefit from minimization of network communications.

Initially, XStore consistently outperforms MongoDB by more than 2x. Nonetheless, the performance gaps broaden as the batch size grows. In particular, XStore overall throughput persistently increase along side with batch size. This indicates that given an undesirable workload such as batch of random queries coupled with operating on a spinning hard drive, XStore is able to maintain its performance as data scales. While it's unfortunate that we did not perform batch query with size larger than 512*K*. The throughput performance gap between XStore and its previous batch size grows larger i.e., 4*K* vs. 2*K*, 8*K* vs. 4*K*, etc. It indicates that a reasonable expectation of it would be at least on-par with the performance of 512*K* batch size or even better.

4.11 Benchmark – Storage Footprint

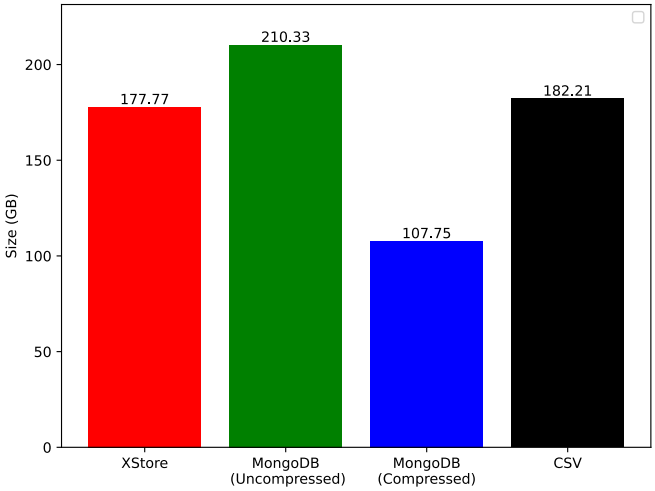

Figure 24: Benchmark – Storage Footprint

The current implementation of XStore does not involve any compression. Therefore, it is expected for XStore to have a larger storage footprint than MongoDB. As part of XStore's next development iteration, XStore currently interprets and stores data as string i.e., An integer with value 123,456 costs 6 bytes instead of 4 bytes. According to [Figure 24], size of MongoDB uncompressed data is in fact larger than XStore. This is due to the fact that MongoDB heavily relies on *BSON* format which introduces overheads such as padding that cause the extra space taken. One of the major contributor to XStore storage size is its inefficiency in storing its underlying data.

4.12 Benchmark – Summary

On the contrary, there's few studies presented a performance that is comparable with XStore. An example of this could derive from a work presented by a team of researchers at *Facebook*. **Gorilla** paper,

a fast, scalable, in-memory time series database. In Gorilla paper, their database system was able to achieve a query performance of [15]:

- Percentile 50^{th}: 2-3 (ms)
- Percentile 90^{th}: 10 (ms)
- Percentile 99^{th}: 105 (ms)

Given the above results presented above, it is interesting that the XStore's technique of leveraging time structure enables superior performance that does not only outperforms *GorillaDB's* on flash performance but almost in-memory. Although *GorillaDB* employs memory to enable such high-performance database, its performance remains inferior than XStore due to the needs of performing scan over all in memory data [15]. Additionally, XStore does not rely on system memory, enabling XStore to be naturally persistent than Gorilla while matching the performance expected as if data is being stored and fetched from memory.

In summary, we have concluded that XStore has persistently outperformed MongoDB predominantly due to the use chronological structure of time series. That is, MongoDB employs *B-Tree* which took $O(log_n)$. While, XStore guarantees a constant time for a CRUD operation since it only requires a simple arithmetic operations to locate an exact location of where to conduct its operation on any given timestamp.

5 CONCLUSIONS

According to our analysis based on several benchmarking sessions discussed in this paper, we can confidently conclude that in the near future, XStore would be able to deliver a superior performance over the aforementioned database storage systems. Nevertheless, there are a lot of work ahead of us and we are excited of the promising solution that XStore brings. Last but not least, it is imperative for us to benchmark and provide an answer to "Which file system is best suited for XStore?" and "How does XStore perform compare to other SQL time-series databases under a distributed, multi-client workload?".

6 FUTURE WORKS

XStore is a young and an on-going project that is around 1 year old as of 02/11/2023. Therefore, there is a monumental amount of work awaits ahead of us. We are optimistic and humble that it will be an exciting journey ahead. Therefore, we have prepared an exhaustive list of future work that could be implemented to XStore in the future:

- Perform benchmark on a real use-case such as conducting a backtest and parameter optimization in financial engineering
- Expand benchmark to more comprehensive benchmark such as benchmark on multiple physical machines, data stored on DFS, include additional TSDB, etc.
- Research low-latency distributed filesystem
  Perhaps, another exciting research topic?
- Branch-out XStore's techniques to *Pandas – Python*
- Integrate XSearch/SCANNS [16] into XStore
  Enabling the ability to search data within time-series data
- More functionality:
  - Enable XStore to support in-memory mode

ACKNOWLEDGMENTS

This work would not have been possible without the support of the Illinois Institute of Technology. I am thankful for invaluable experience and knowledge from Dr. Ioan Raicu, Professor and Director of the Data-Intensive Distributed Systems Laboratory at IIT.

I am also grateful and would like to thank Alexandru Iulian Orhean. As a fellow Ph.D. student, he has taught me more than I could ever give him credit for. Throughout this project, Alexandru has provided me with an extensive guide on how to navigate through C++ as well as asking the right questions to improve XStore.

REFERENCES

[1] InfluxData. Influxdb.
[2] MongoDB Inc. Mongodb wiki.
[3] Carlos Garcia Calatrava, Yolanda Becerra Fontal, Fernando M Cucchietti, and Carla Diví Cuesta. Nagaredb: A resource-efficient document-oriented time-series database. *Data*, 6(8):91, 2021.
[4] MongoDB Inc. Set granularity for time series data – mongodb manual.
[5] Sven. Sparse file - wikipedia.
[6] Vera (Mini Tool). Ssd prices continue to fall, now upgrade your hard drive!
[7] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.
[8] Feng Chen, David A Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):181–192, 2009.
[9] Milo Polte, Jiri Simsa, and Garth Gibson. Comparing performance of solid state devices and mechanical disks. In *2008 3rd Petascale Data Storage Workshop*, pages 1–7. IEEE, 2008.
[10] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, Jihong Kim, et al. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *USENIX Annual Technical Conference*, pages 759–771, 2017.
[11] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.
[12] MongoDB Inc. Time series – mongodb manual.
[13] MongoDB Inc. Add secondary indexes to time series collections – mongodb manual.
[14] Seagate Technology LLC. Constellation.2 data sheet.
[15] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.
[16] Alexandru Iulian Orhean, Anna Giannakou, Lavanya Ramakrishnan, Kyle Chard, and Ioan Raicu. Scanns: Towards scalable and concurrent data indexing and searching in high-end computing system. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 51–60. IEEE, 2022.