NDBlaze : Sequentializing Random Writes to a Spatial Database

Kunal Lillaney
Department of Computer Science
Johns Hopkins University
lillaney@jhu.edu

Randal Burns
Department of Computer Science
Johns Hopkins University
randal@jhu.edu

Joshua Vogelstein
Department of Biomedical Engineering
Johns Hopkins University
jovo@jhu.edu

ABSTRACT

We describe a memory based buffer using Redis and Spark that accelerates random writes to a spatial database. Spatial databases are usually read optimized; they use spatial indexes to sequentialize read I/O. As a consequence, random write workloads experience poor I/O performance. We present a system that performs writes asynchronously in memory and then merges them and writes them to the database backend asynchronously. Once in memory the random writes are sorted based on a Z-order space-filling curve and then written to disk sequentially, which maximizes write throughput. Our system improves the user-perceived write throughput by up to 38 times and the I/O throughput to disk by 3.3 times. All interfaces to this system are RESTful web services and they provide a consistent view of the data to the user.

CCS Concepts

•Information systems → Database management system engines; Main memory engines; Spatial-temporal systems;

Keywords

Data-intensive computing, Spatial Databases

1. INTRODUCTION

In the world of Big Data, spatial databases have become increasingly relevant, because of their use in Geographical Informations Systems, like Google Maps [21], and GPS navigation systems [24]. Spatial database are also being used in numerous real-world data-mining applications like land-usage [13], disaster management [10], and data-intensive science domains, such as turbulence simulations [20] and astronomy [27]. As applications and uses of spatial databases become more common, we see a growing need for high performance solutions. The growing performance disparity between compute and storage has left I/O a significant bottleneck [18] and this has affected spatial databases as well.

SSDBM '16 Budapest, Hungary
© 2016 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

Most spatial databases are designed for efficient read-only queries [12]. This aligns well with spatial indexes as it translates into sequential reads and uses disk seeks optimally. In this model, changes to the spatial database should be infrequent, relatively small and largely not time sensitive. A good example is updating location labels in Google Maps, which happens infrequently and represents a fraction of the workload. But there is an increasing number of applications driven by data mining and machine learning [9] for which write performance is critical. Some of these applications generate secondary spatial data in the form of annotations and labels which have to be stored in the same spatial database.

1.1 Motivation

Our motivating application is the *Open Connectome Project* (OCP) [6], a scalable high-throughput database cluster for spatial analysis of neuroscience imaging data. OCP stores seven different modalities of imaging data–from millimeter to nanometer scale—in a spatial database using a Z-order space filling curve. The project currently has more than 70 unique datasets which total more than 150TB. One of the workloads for this project is to run parallel computer vision algorithms at scale on high-performance compute clusters to build neural connectivity maps of the brain [22]. Reconstructing neural circuits at scale is important to understand the inner working of the human brain and solve many challenging problems. These scalable workloads have over a period of 3 days detected more then 19 million synapses in a 4 trillion pixel image volume using a cluster of 3 physical nodes with 186 cores [15].

As imaging data volumes get larger, we are looking at petabytes of data for a mouse brain and exabytes of data for a human brain [3]. Our computer vision workload will generate terabytes of annotations for the peta-scale mouse brain data. Annotation data then is written to the spatial database and co-registered with image data. The compute resources generating these annotations write them instantaneously and synchronously and move on to process more image data. The problem at hand is to frequently write terabytes of data to a spatial database, keeping in mind the time-sensitive nature of the workload. The ability to achieve a high write throughput for spatial database is critical to solve this problem.

For most workloads, the actual write throughput to a spatial database is quite low when compared with read performance. This arises because writes arrive in random order and translate into random writes at disk. Storage in general deals poorly with random writes, preferring sequential writes which avoid seeks for magnetic disk and increase parallelism for solid-state devices. Because of random writes, we are not utilizing the available write throughput. To improve utilization of the available write throughput, there should only be sequential writes to the database. Sequential writes in this case are writes to continuous regions of the database which can be identified by spatial indexes. This means that we can use spatial indexes to reorder the random writes and convert them to sequential writes.

1.2 Contributions

Our system, called NDBlaze, accelerates random writes by using memory to buffer them before they are written to the database. (1) NDBlaze writes data directly to a Redis memory-based key value store and significantly decreases the perceived write latency for compute resources. Random writes to memory have extremely low latency when compared to storage. (2) Our system scales well with the number of writers. Because we are writing to memory, our system avoids I/O interference, which becomes worse with an increase in number of writers. (3) With NDBlaze, we sort and merge multiple writes and reconcile them with data in the database. An in-memory parallel merge, implemented in Spark, allows us to achieve spatial merging quickly and at scale. (4) The system uses multiple writer threads to asynchronously commit the merged writes. The merged writes are written to the database as sequential writes utilizing maximum possible I/O bandwidth. (5) Our system provides a consistent view of the written data, subject to the limitations, of the underlying systems RESTful interface, with a small overhead. This overhead arises because we have to reconcile our buffered writes with the spatial database before data can be sent back to the user on a read. With this approach, we are able to achieve a 38-fold increase in perceived random write performance to the spatial database when compared to direct random writes. There is also up to a 3.3-fold increase in the resultant sequential writes throughput when compared to direct random writes.

2. RELATED WORK

Buffering writes is not a new concept and many existing systems use buffering to increase write performance. Adding buffers to B-trees and varying the degree of the tree improves the amortized I/O bound for writes [5]. Cache-oblivious streaming B-trees [4] improve B-trees for random writes with shuttle trees, which buffer elements at various levels and use fractal cascading to amortize the cost of writes. The log-structured merge tree (LSM tree) [17] is a data structure that buffers writes in memory. Writes are sorted on their index when they are stored in memory and are stored and merged across multiple levels of the memory hierarchy. When the writes in memory overflow, there is a cascade of writes and they are optimally written to disk in batches, similar to a merge sort. LSM trees are used by many key-value stores, such as Apache Cassandra [16], BigTable [7], LevelDB [8], HBase [1] for amortization of writes. An example of a similar system is BetrFS [14], a write-optimized file system which uses write-optimized indexes and fractal trees in the kernel for random writes to disk.

Our system takes inspiration from the merging of writes in an LSM tree. Both LSM trees and NDBlaze are optimized for writes, but do allow reads on modified data. Both of them are geared towards write intensive applications with infrequent reads. But NDBlaze differs from LSM trees in one important aspect; it performs spatial merging in memory. Similarly, BetrFS does not manage spatial data nor perform spatial merging. Spatial merging allows us to write data to a spatial database in a sequential manner utilizing the storage efficiently.

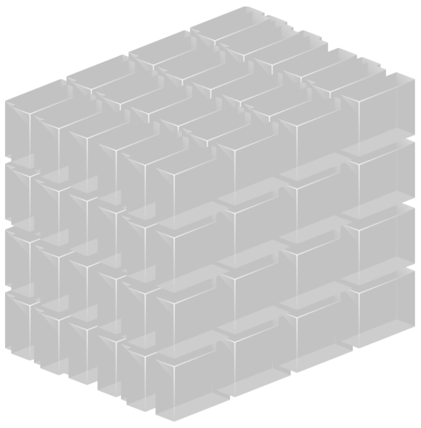


Figure 1: Cuboid structure [6].

3. BACKGROUND

The basic storage structure in our database is a dense multi-dimensional spatial array that is partitioned into rectangular subregions [6]. We call these subregions as "cuboids" and they are similar to chunks in ArrayStore [26]. Each cuboid is assigned an index using a Morton-order space filling curve. Space filling curves organize data recursively such that any power-of-two aligned subregion is wholly continuous in the index [20]. Large aligned regions of data are stored sequentially on disk and can be read in a single streaming I/O. This ensures that the small cuboid size does not have much effect on I/O throughput. Although data are stored as cuboids, we do not restrict the services to cuboid aligned regions. User can read or write arbitrary sub-regions of data comprising of one or more cuboids.

The data stored in our spatial database are organized into different projects. A project contains information about the spatial extent of the data and consists of numerous channels. Channels contain data of different types, e.g. image or annotation, with the same spatial extent. This allows the annotation data to be co-registered with the relevant image data. Typical workloads, read dense image data, find structures in the data, and write back labels or super-pixels to an annotation channel. Annotations can be sparse which means they might be spread across the spatial space with very few of them being adjacent. They can also be dense which means they might be very tightly packed in a spatial region. Annotations are stored in dense cuboids and we store these cuboids lazily and sparsely. This allows us to store cuboids which actually have data and ignore the ones which do not, reducing I/O and storage on the database.

4. ARCHITECTURE AND DESIGN

We describe the NDBlaze architecture in Figure 2 and how it fits into the overall architecture of the OCP data cluster. NDBlaze is located in between the load balanced

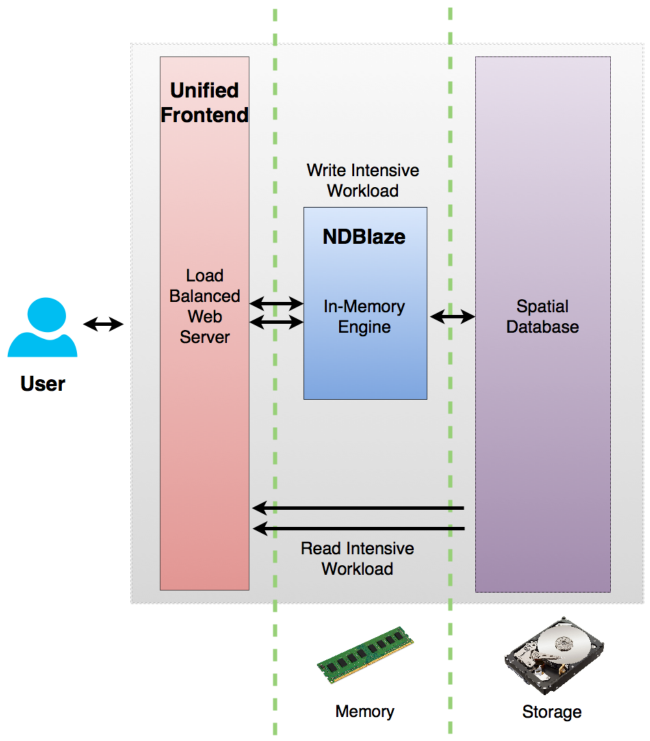


Figure 2: Architecture of NDBlaze with NDStore

web server and the backend spatial database. The load balancer receives read and write requests as Web-service calls from any number of sources: a Web-browser visualization tool, an individual running a script, or a compute cluster running an HPC workflow. Requests are redirected by the load balancer to NDBlaze or the spatial database based on nature of the request. Read requests for immutable datasets are forwarded directly to the spatial database. Write requests for mutable datasets are directed to NDBlaze where they are buffered and eventually committed to the spatial database. If there is a read request on an mutable dataset then it is forwarded to NDBlaze for processing. NDBlaze dynamically determines which data is buffered or committed to the database and merges the data correctly before it is presented to the user.

An important reason to adopt this architecture is the resulting flexibility in the deployment model. NDBlaze can be deployed as a transparent layer over the spatial database and can be located closer to the user or the database based on the user requirements. Our system is accessible via RESTful web services and also utilizes them to write data back to the spatial database. The NDBlaze architecture is scalable depending on the nature of the workload. It can be deployed on a single node, or with dedicated Redis and Spark clusters spanning across multiple nodes.

4.1 Write Processing

Figure 3 presents an overview of write processing in NDBlaze and the data flow path. The flow path can be divided into two sections: synchronous writes from the user to memory and asynchronous writes from cluster memory to the spatial database backend. NDBlaze uses Nginx and Python Django to run a web-service which accepts writes from the user. When a data blob is posted to our service, we synchronously insert this data blob to a Redis (Section 4.2.1) database and update the respective indexes. A data blob here consists of multiple cuboids and need not be aligned. This process of synchronous data insertion continues and we use a daemon to monitor the consumption of memory by Redis. When the memory consumed by Redis breaches our threshold, the daemon launches a Spark (Section 4.2.2) job to flush this data back to the spatial database. The Spark job reads all the data blobs from the Redis database and breaks the blobs into smaller cuboids and their respective Z-indexes. Simultaneously, it fetches the corresponding cuboids present in the spatial database. We now sort and merge these data cuboids in memory based on their Z-indexes. The merged cuboids are inserted into a write queue and asynchronous workers (Celery; Section 4.2.3) post them to the spatial database. The celery workers run in the background and write these cuboids asynchronously. Spark may accept more merge tasks while existing I/Os are pending. When specific cuboids are committed to the database, we evict the related data blobs from Redis and update the indexes. We choose to write back in cuboid form and not larger subregions to avoid further processing at the spatial database, i.e. the spatial database may write cuboids directly to their store and need not de-serialize and then serialize data.

The data flow path significantly decreases the number of I/O operations to the spatial database for rewrites to the same spatial region. I/O operations to the spatial database are expensive because of high write latency for storage. When a cuboid is posted directly to the spatial database, we read the corresponding cuboid existing in the spatial database and merge it with the posted cuboid. This whole operation comprises single read and single write operation, $\mathcal{O}(2)$ I/O operations for a single cuboid post request. For n data post requests to the same cuboid, we perform $\mathcal{O}(n)$ reads and $\mathcal{O}(n)$ writes to the spatial database. A total of $\mathcal{O}(2n)$ I/O operations for n post requests. Now, let us consider the case when we post these n cuboids to NDBlaze. We replace the $\mathcal{O}(2n)$ I/O operations to the spatial database with $\mathcal{O}(2n)$ I/O operations to memory which is many orders of magnitude faster in terms of latency. We perform a single read operation from the spatial database which is required to merge the buffered data with committed data. There is also only a single write operation to commit this merged data back to the database. In total, we perform $\mathcal{O}(2)$ I/O operations to the spatial database, reducing the number of operations to the database by a factor of n.

NDBlaze also allows the user to read the buffered data in memory, which occurs infrequently, but is a possibility that we address. We do this by triggering a Spark job when there is a read request to a buffered spatial region. All the steps of the data flow path described above remain the same with an additional step. We return the merged data region back to the user via the web-service before we insert the cuboids into the write queue. The overhead here, compared to reading it directly from a spatial database, is the time taken to merge the cuboids in memory. The writes to the database continue to occur in a background process and do not affect the latency in the modified data flow path.

4.2 Software Design

NDBlaze consists of many diverse components, each of which has a specific role. We detail the components in the order of their occurrence in the data flow path.

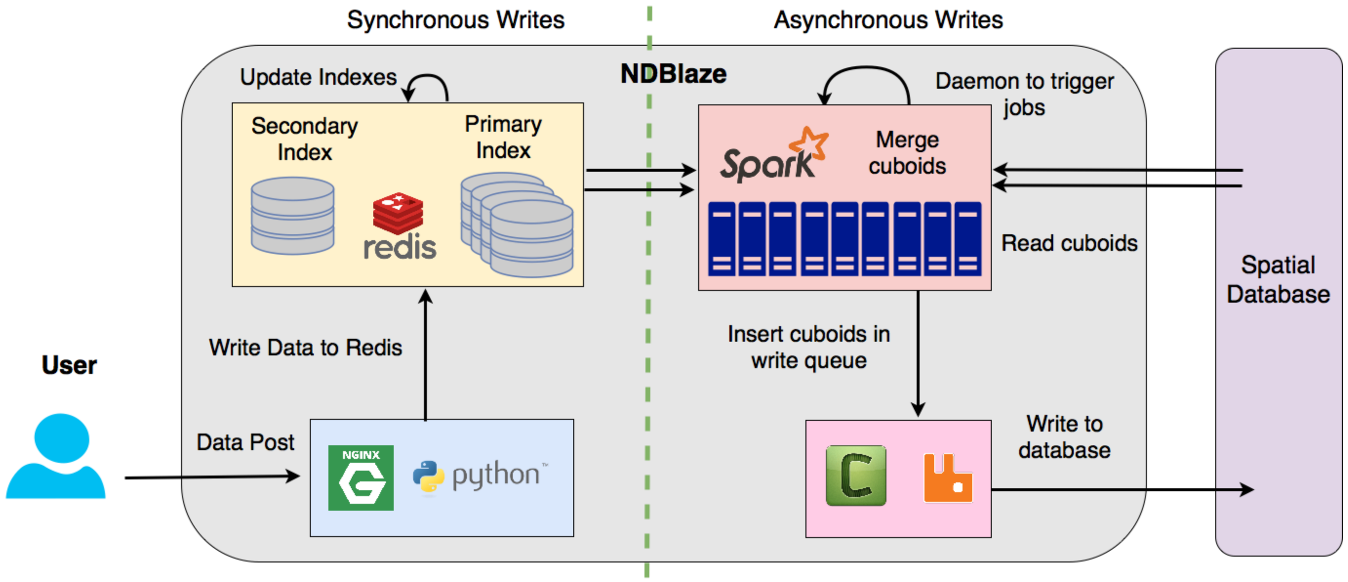


Figure 3: Write processing in NDBlaze

4.2.1 *Redis*

We choose Redis [23], which is an open-source in-memory data structure store, to buffer the random writes. We explored many other systems such as Aerospike [2], MemSQL [25] and Spark [28] for this role. Some of them are not fast enough or not able to scale well for multiple writers, which is a strong prerequisite for our work flow. There were other academic projects, such as RAMCloud [19], which might be a good replacement for this component.

Redis is a good choice from the currently available commercial grade systems for multiple reasons. It is entirely in-memory, which offers us low latency to random writes and low write contention in case of multiple writers. It has a cluster mode of operation, which allows us to scale out our memory buffer and not get confined by the physical memory limitations of a single node. Finally, Redis has support to store sets and perform set operations such as union, difference, and intersection, which we use for managing the indexes.

The organization of data blobs in Redis has a big impact on performance and write optimizations. Each data blob can cover a spatial region consisting of multiple Z-indexes. We considered and rejected, breaking a data blob into smaller cuboids before they are inserted into Redis. This involves a simple indexing scheme of key-values and makes inserts or fetches from memory a single lookup operation. But there is a down-side to this option, because the breakup process involves unpacking the large data blob into a numpy array and repacking them into smaller numpy arrays. The process of packing and unpacking numpy arrays is time consuming and relatively slow when compared to memory write latency. Plus, doing one large write as opposed to many cumulative smaller writes is generally preferred. We chose a more efficient option. We perform a single write to the Redis database when a data blob is posted to NDBlaze, The cuboid breakup occurs in parallel within Spark as we merge the cuboids. This choice is made to avoid any latency resulting from serialization and de-serialization of data at the time of synchronous writes.

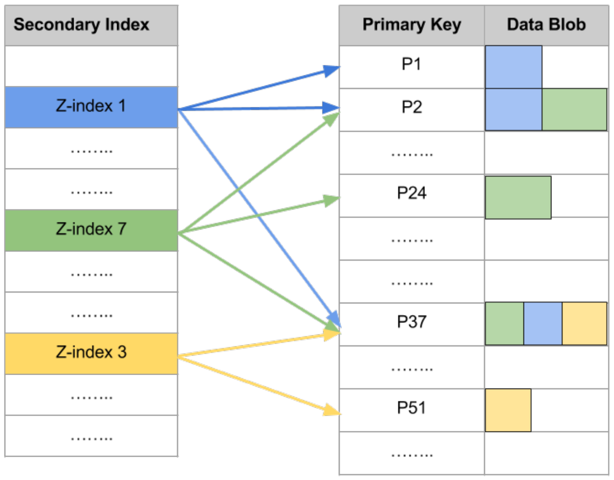


Figure 4: Organization of Data Blobs in Redis.

A post operation consists of the data blob itself and URL parameters describing the data blob. We generate a primary key using the URL parameters to uniquely identify this data blob. The primary key is self describing and consists of dataset, channel name, resolution, x, y, z bounds and the current time-stamp. A dataset is unique across the system and this allows us to store meta-data for the data blob in the primary key itself. We append the time-stamp at the end to determine the order of writes with the same x, y, z bounds. The time-stamp argument is also used at the time of data blob eviction to decide which write was merged last. In Figure 4, primary key P37 maps to a data blob which has Z-indexes 1, 2 and 7.

We maintain a secondary index over the primary key to identify data blobs for a given Z-index and use this during a spatial merge. This problem arises because we defer the

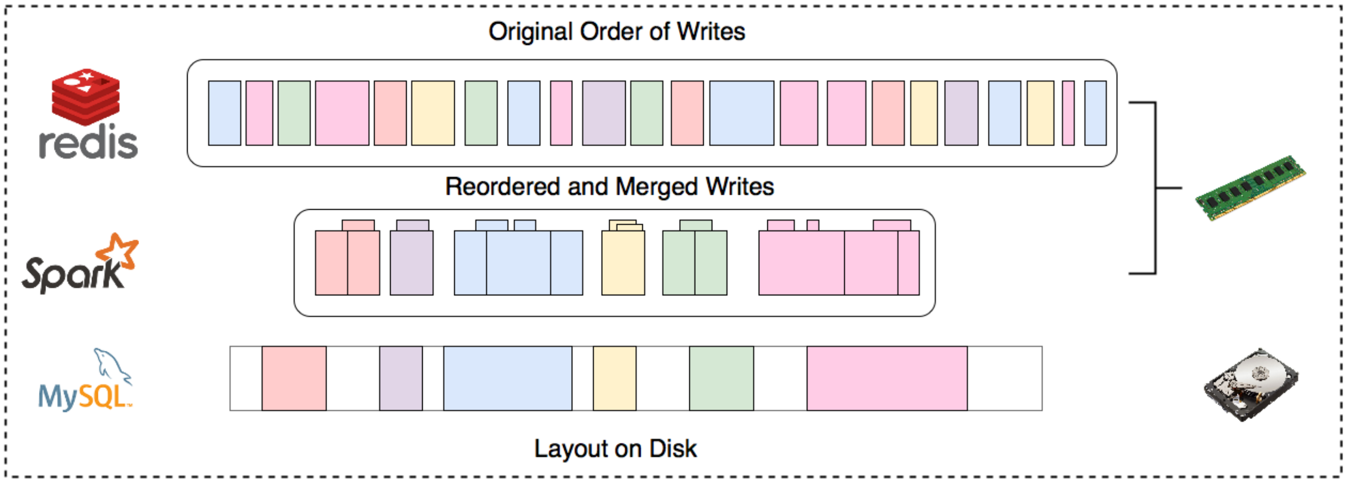


Figure 5: Order of writes in NDBlaze.

cuboid breakup process at insertion time and do not map them based on their Z-indexes. But we do need quick access to all data blobs for a given Z-index without having to iterate over the entire memory buffer. A secondary index maps to a set of primary keys and there is one for every Z-index present in all the data blobs. It consists of the dataset, channel name, resolution and Z-index. An insert or update occurs whenever a data blob is written. Using this index, we can merge all data blobs with a specific Z-index in two lookups. In Figure 4, Z-index 7 exists in three data blobs of P2, P24 and P37. We merge the respective data blobs when Z-index 7 is committed back to the database.

NDBlaze has two different Redis configurations based on the deployment scenario. (1) Standalone Redis instance which exists on a single node and contains data blobs, primary keys, secondary indexes on the same node. (2) Redis cluster where the secondary index is stored on a standalone node while the primary keys and data blobs are distributed across multiple nodes. The two different configurations exists because Redis in cluster mode does not support all operations we need to maintain the secondary index. This deployment can be done on a per dataset basis where some larger datasets can be buffered using the cluster mode and smaller datasets using the standalone mode.

In order to provide a consistent view of the database to the user, we choose not to evict the data blobs from Redis when they are fetched into Spark. Instead, we evict them after they are written to the database. If there is a request by the user for a spatial region which is currently being merged in Spark, we launch another job which returns the same view. We evict the data blobs using the secondary index and use the time-stamp value of the last write merged to remove only the merged data blobs. With this approach, we can continuously accept writes even when we are flushing cuboids to the database.

4.2.2 *Spark*

The second important component of our system is Apache Spark [28], which is a fast engine for large-scale data processing. With Spark, we are able to keep our data within memory at all times and are not limited by the memory capacity of a single node.

Serialization in Spark proves to be a major bottleneck for running massive workloads. Spark serializes all of its data before it can perform any operation and this leads to a problem if the data is of a non-standard datatype. All our data blobs are numpy arrays and the default serializer used by PySpark is cPickle which is extremely slow for serializing numpy arrays. PySpark offers other sterilizers as well, such as Marshal and UTF8, both of which perform much better than cPickle but do not offer any support for numpy arrays. For serializing numpy data, we choose Bloscpack [11] or Blosc, which is an extremely fast and lightweight serialization format for numpy arrays based on the Blosc codec. Blosc spits out byte arrays that can then be serialized using the fastest serializer available for Spark. Also, Blosc compresses the data with serialization and thus the data has a smaller memory footprint in Redis. One caveat of using Blosc is the maximum limit on the size of file which it can compress. This number is the value of `INT_MAX` and translates into approximately 2048MB. None of our cube sizes in practice exceed this value and this limitation of Blosc is not an issue for our workload.

A Spark job performs multiple tasks in the work flow, ranging from reading data to posting it in the write queue. The job first fetches all the secondary indexes present in the Redis database. It then fetches from Redis the data blobs mapped by the secondary indexes. In the next stage, the data blobs are broken into smaller cuboids and mapped to their respective Z-indexes. The process of breaking a data blob involves iterating over a large numpy array and extracting cuboids, essentially smaller numpy arrays, from it. We de-serialize the data blobs in Spark in order to decompose them and then serialize the cuboids once again to avoid paying the serialization penalty for Spark. A key is assigned to each cuboid based on the dataset and resolution that the cuboid belongs to. This information is extracted from the primary keys of the data blobs when they are decomposed into cuboids, preventing any more lookups from Redis. Simultaneously, we fetch the corresponding cuboids from the spatial database using the Z-index in the secondary indexes. Again, the secondary indexes are self describing which allows Spark to fetch them directly from the backend without any additional lookups. These two lists of cuboids are now combined in Spark, sorted based on Z-index, and merged. We sort the list of cuboids to ensure the correct write order which is essential for sequential writes to the database.

Figure 5 depicts the various levels of NDBlaze and how the writes are sorted and merged in Spark to match their layout on disk.

The merge function combines various versions of the same cuboid and orders the writes correctly. It essentially compares each pixel value in two numpy arrays. Because we need the pixel values, the merge process cannot be done with packed arrays and they are unpacked once again. A non-zero pixel value always take precedence over a zero pixel value. This ensures that objects detected in a particular work flow do not overwrite existing objects if they are not overlapping. In case there is a conflict, i.e both pixel values are non-zero then we consider the last write as correct and overwrite the other pixel value. The time-stamp in the primary key allows us to identify the correct order of writes. This function has exactly the same behavior as in the spatial database and does not affect the consistency of the data. We initially used a numpy vectorize function to merge cuboids within Spark. This proved to be too slow for our purpose and we implement an optimized routine in C bound with python ctypes to accelerate this process. Python ctypes are python wrappers around C functions and are an order of magnitude faster then their pure python counter parts. There were some challenges in using ctype functions for Spark, because Sparks requires a shared C library. This is simple for a single node deployment, but Spark has multiple workers across many nodes that do not share the same python environment. This library has to be copied to all nodes at the time of deployment.

4.2.3 Celery

The merged cuboids are posted to a write queue and deferred to background processes. Here the write queue is a RabbitMQ message queue, while the background processes are celery workers. The celery workers use RESTful calls to post cuboids back to the spatial database. We choose to use a write queue instead of posting data directly from Spark for two reasons: (1) writing data to the database is relatively slow, which will tie up the Spark workers and create a backlog of other jobs; and, (2) the write queue allows us to control the flow of writes and manage it for different storage media. For example, we use single thread for magnetic disk drives to avoid disk contention arising from multiple writers. Alternately, we can use a preset number of writers for SSDs, so that we can utilize their write parallelism. Using celery we can easily scale and use processes across multiple nodes for workers. When the merge is finalized and written to the spatial database, we delete the indexes and the data blobs from the Redis database.

5. RESULTS

Write throughput is the principal performance measure for our system, determining how much data can be written by a parallel computer-vision work flow. We conducted the experiments against the web-services of NDBlaze. Experiments show the data blob size, not the size of the data transferred or written to disk. Both data blobs and cuboids are compressed when written to disk and uncompressed when read from disk. Neuroscience imaging data in general has high entropy and tends to compress by between 10-20% with the Blosc compression used for these experiments.

Experiments are conducted on Amazon AWS, because our deployment requires memory optimized nodes for the Re-

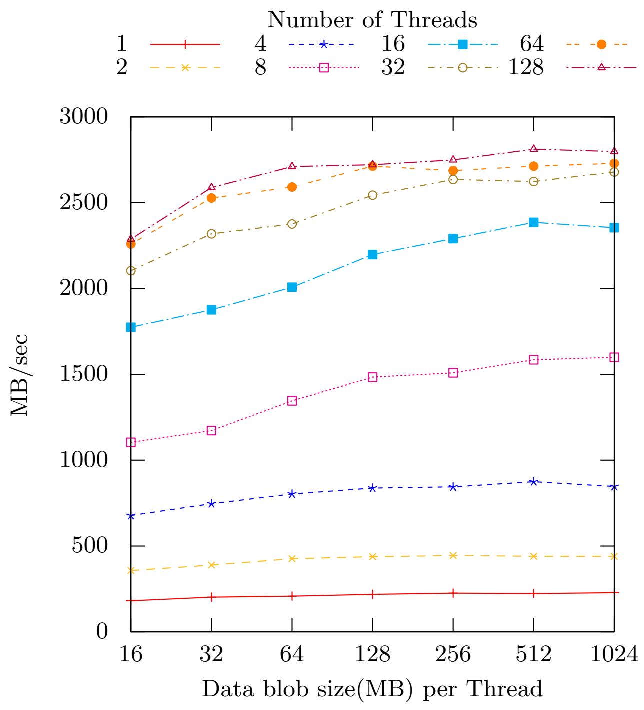
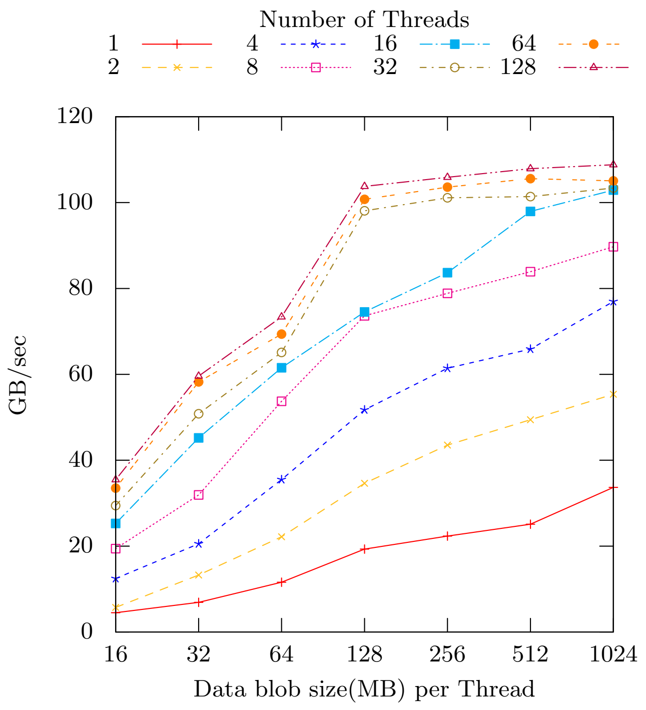


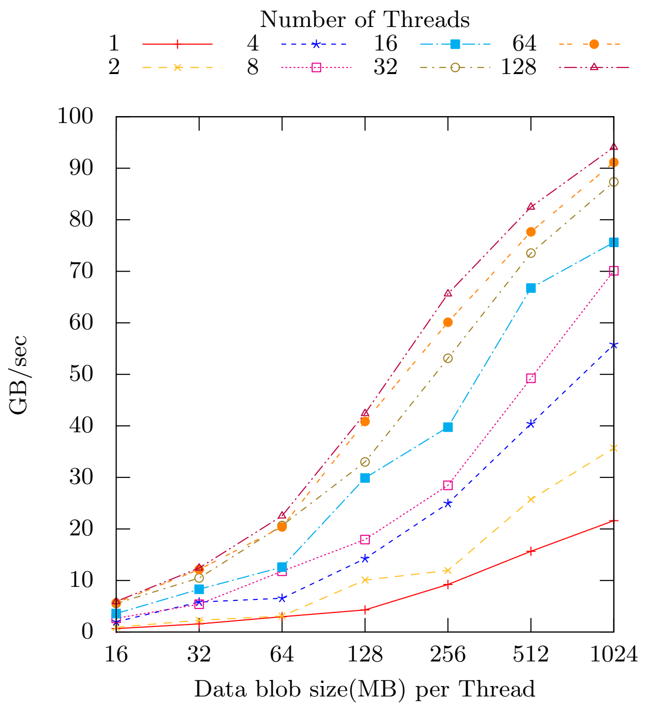
Figure 6: Direct Spatial Database Write Throughput: without NDBlaze.

dis cluster, compute optimized nodes for the Spark cluster, and storage optimized nodes for the spatial database. The spatial database backend is a single AWS i2.8xlarge instance: 32 vCPU's, 244GB of memory,a 10GBps connection and a RAID 0 array of 8 800GB dedicated SSD's for a 4.5TB TB XFS file system. The spatial database runs on MySQL. NDBlaze is a single AWS r3.4xlarge instance: 16vCPU's and 122GB of memory. For the Redis cluster we have two configurations: a single node local instance and a 3 node cluster. The Redis cluster is represented by three AWS r3.4xlarge instances: each with 16vCPU's, 122GB of memory. The nodes are deployed without any replicas and act as 3 independent masters. These Redis nodes store data blobs. We also run an instance of Redis in the memory of the NDBlaze node to store secondary indexes. It is important to note that we choose to use our own deployment of Redis instead of using the AWS Redis cache, because the AWS cache is only available for a single node and not very tunable. The Spark cluster consists of a single master and four slave AWS m4.4xlarge instances: each with 16vCPU's and 64GB of memory. We run a total of 16 Spark workers each with a single executor: 4 vCPU's and 16GB of memory. We use the default PySpark serializer and the serialization buffer is allocated 1GB.

The performance of the OCP spatial database approaches 3 GB/s for larger block sizes high degrees of parallelism. Figure 6 shows the write throughput to the spatial database as a function of data blob size for different number of threads. In this experiment we perform random writes directly to the database. One can the see that write throughput remains pretty steady with some marginal increases as we increase the data blob size. We observe a close to linear in-



(a) Standalone Redis



(b) 3 node Redis cluster

Figure 7: Perceived Write Throughput with NDBlaze

crease in writes up to 32 threads. The SSD hardware of the AWS nodes benefit from parallel writers as there are multiple channels in each device. We are able achieve a peak write throughput of 2.8GB/sec for 128 threads and 512MB data blob size.

By completing writes asynchronously in memory, NDBlaze increases the perceived write throughput by a factor of 40. Figure 7 shows the write throughput of the NDBlaze as a function of data blob size for different number of threads. We refer to these results as the perceived write throughput, because the user witnesses these data completion rates for I/O bursts. Sustained I/O will fill the buffer and eventually performance will fall to the true I/O rate. We test this for two different Redis deployments and benchmark them. In Figure 7a, we deploy NDBlaze on a standalone Redis Node with 122GB of memory. Write throughput scales linearly with increase in data blob size up to 16 threads. Here, we are limited by maximum Blosc size and cannot test the data blob size beyond 1GB data. We achieve a peak write throughput of 108GB/sec for 128 threads and 1GB data blob size. As we increase the number of threads, our write throughput plateaus, because we are limited by memory bandwidth of the the single Redis node. In Figure 7b, we benchmark NDBlaze with the 3 node Redis cluster. Here, we can achieve a peak of 94GB/sec for 128 threads and 1GB data blob size. Notably, this throughput is lower than the peak for a single Redis node. This arises from the overhead of cluster coordination and routing data among nodes. Because the single node is more efficient, we recommend a deployment in which we deploy multiple instances of NDBlaze, each with a single node Redis cluster. Each of these instances will be dedicated for a single dataset and redirections will be managed by the web-services load balancer. This allows us to scale without the performance penalty of writing to cluster.

Beyond burst performance, NDBlaze improves the sustained I/O performance to the spatial database, because it makes I/O more sequential. Figure 8 shows the write throughput to NDBlaze from the write queue. For this benchmark, we pre-fill the queue with 16GB of merged and spatial index sorted data, which is then written by celery workers. This experiment shows the performance of the asynchronous I/O processes when clearing the write queue. We experiment over multiple celery workers ranging from 1 to 16 and receive a peak at 8 workers of 9.5GB/sec which is about 3.3 times faster then 2.8GB/sec peak for random writes with 32 workers. Again, the AWS SSDs benefit from parallel I/O, because they have multiple channels. When running on disk systems (not shown), a single writer performs best, because it eliminates write contention.

6. CONCLUSIONS AND FUTURE WORK

We have developed a system that captures random writes in memory, improving the application perceived performance by a factor of 40 for I/O bursts. Furthermore, the system improves the sustained I/O throughput by more than a factor of three, from 2.8 GB/s to 9.7 GB/s, because it delays random writes and performs them sequentially. We have deployed this platform for internal use and are in the process of rolling it out to all users of *NeuroData* (http://neurodata.io) and the *Open Connectome Project* (http://openconnecto.me). All software are open-source and available for collaborative development and reuse at our github repository (http:

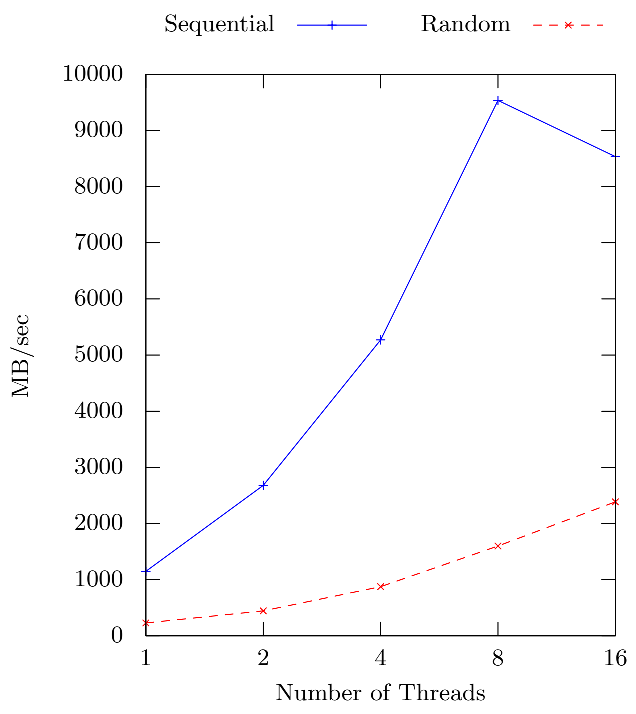


Figure 8: Actual Write Throughput

//github.com/neurodata/ndblaze). Random write I/O has been the performance limiting factor in our computer vision workloads and NDBlaze will transform our ability to run parallel machine learning and computer vision algorithms on neuroscience imaging data.

Additional engineering will allow NDBlaze to extend its performance benefits to arbitrarily longer bursts. As implemented, when the Redis cluster memory fills, the I/O throughput will degrade to the 9.5 GB/s that the asynchronous writing processes can support. In AWS, we can scale both the Redis cluster, by adding more nodes, and the I/O throughput, with additional spatial database nodes, based on auto-scaling components on demand to meet an I/O surge.

7. REFERENCES

[1] Apache HBase. http://hbase.apache.org, 2010.

[2] Aerospike. http://www.aerospike.com, 2012.

[3] A. Abbott et al. Solving the brain. *Nature*, 499(7458):272–274, 2013.

[4] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92. ACM, 2007.

[5] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 546–554. Society for Industrial and Applied Mathematics, 2003.

[6] R. Burns, K. Lillaney, D. R. Berger, L. Grosenick, K. Deisseroth, R. C. Reid, W. G. Roncal, P. Manavalan, D. D. Bock, N. Kasthuri, M. Kazhdan, S. J. Smith, D. Kleissas, E. Perlman, K. Chung, N. C. Weiler, J. Lichtman, A. S. Szalay, J. T. Vogelstein, and R. J. Vogelstein. The Open Connectome Project Data Cluster: Scalable Analysis and Vision for High-throughput Neuroscience. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[8] J. Dean and S. Ghemawat. leveldb–a fast and lightweight key/value database library by Google, 2011.

[9] M. Ester, H.-P. Kriegel, and J. Sander. Algorithms and applications for spatial data mining. *Geographic Data Mining and Knowledge Discovery*, 5(6), 2001.

[10] A. E. Gunes and J. P. Kovel. Using GIS in emergency management operations. *Journal of Urban Planning and Development*, 126(3):136–149, 2000.

[11] V. Haenel. Bloscpack: a compressed lightweight serialization format for numerical data. *arXiv preprint arXiv:1404.6383*, 2014.

[12] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd international conference on Very large data bases*, pages 487–498. VLDB Endowment, 2006.

[13] C. Homer, C. Huang, L. Yang, B. Wylie, and M. Coan. Development of a 2001 national land-cover database for the United States. *Photogrammetric Engineering & Remote Sensing*, 70(7):829–840, 2004.

[14] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, et al. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, 2015.

[15] D. M. Kleissas, W. Gray Roncal, P. Manavalan, J. T. Vogelstein, D. D. Bock, R. Burns, and R. J. Vogelstein. Large-Scale Synapse Detection Using CAJAL3D. *Neuroinformatics*, 2013.

[16] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[17] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[18] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23(1):11–28, 1989.

[19] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[20] E. Perlman, R. Burns, Y. Li, and C. Meneveau. Data Exploration of Turbulence Simulations using a Database Cluster. In *Supercomputing*, 2007.

[21] P. Rigaux, M. Scholl, and A. Voisard. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2001.

[22] W. R. G. Roncal, D. M. Kleissas, J. T. Vogelstein, P. Manavalan, K. Lillaney, M. Pekala, R. Burns, R. J. Vogelstein, C. E. Priebe, M. A. Chevillet, et al. An automated images-to-graphs framework for high resolution connectomics. *Frontiers in neuroinformatics*, 9, 2015.

[23] S. Sanfilippo and P. Noordhuis. Redis. http://redis.io, 2009.

[24] J. Schiller and A. Voisard. *Location-based services*. Elsevier, 2004.

[25] N. Shamgunov. The MemSQL In-Memory Database System. In *IMDM@ VLDB*, 2014.

[26] E. Soroush, M. Balazinska, and D. Wang. Arraystore: a storage manager for complex parallel array processing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 253–264. ACM, 2011.

[27] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, and P. Kukol. Indexing the Sphere with the Hierarchical Triangular Mesh. Technical Report MSR-TR-2005-123, Microsoft Research, 2005.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10:10–10, 2010.