RocksDB

A embedded key-value store for fast storage

Monday, May 1, 2017

The birth of RocksDB-Cloud

I have been working on an open source project called RocksDB-Cloud for the last few months. What does it do and why did we build it?

You might be using RocksDB on your application servers or database servers on a public cloud service like AWS or Azure. RocksDB is a high performance embedded storage engine. But when your server machine instance dies and restarts on a different machine in the cloud, then you lose all your data. This is painful and many engineering teams have build own custom replication code around RocksDB to protect against this scenario. RocksDB-Cloud is built to provide a readymade solution to this problem so that you do not have to write any custom code for making RocksDB data durable and available.

RocksDB-Cloud provides **three main advantages** for Cloud environments:

1. A RocksDB-Cloud instance is **durable**. Continuous and automatic replication of db data and metadata to Cloud Storage (e.g. AWS S3). In the event that the RocksDB-Cloud machine dies, another process on any other EC2 machine can reopen the same RocksDB-Cloud database
2. A RocksDB-Cloud instance is **cloneable**. RocksDB-Cloud supports a primitive called zero-copy-clone that allows another instance of RocksDB-Cloud on another machine to clone an existing db. Both master and slave RocksDB-Cloud instance can run in parallel and they share some set of common database files.
3. A RocksDB-Cloud instance automatically places **hot data** in SSD and **cold data** in Cloud Storage. The entire database storage footprint need not be resident on costly SSD. The Cloud storage contains the entire database and the local storage contains only the files that are in the working set.

RocksDB-Cloud is a Open-Source C++ library that brings the power of RocksDB to AWS, Google Cloud and Microsoft Azure. It leverages the power of RocksDB to provide fast key-value access to data stored in Flash and RAM systems. It provides for data durability even in the face of machine failures by integrations with cloud services like AWS-S3. It allows a cost-effective way to utilize the rich hierarchy of storage services (based on RAM, NvMe, SSD, Disk, Cold Storage, etc) that are offered by most cloud providers.

About Me

Dhruba Borthakur

Connect to me at http://www.facebook.com/dhruba

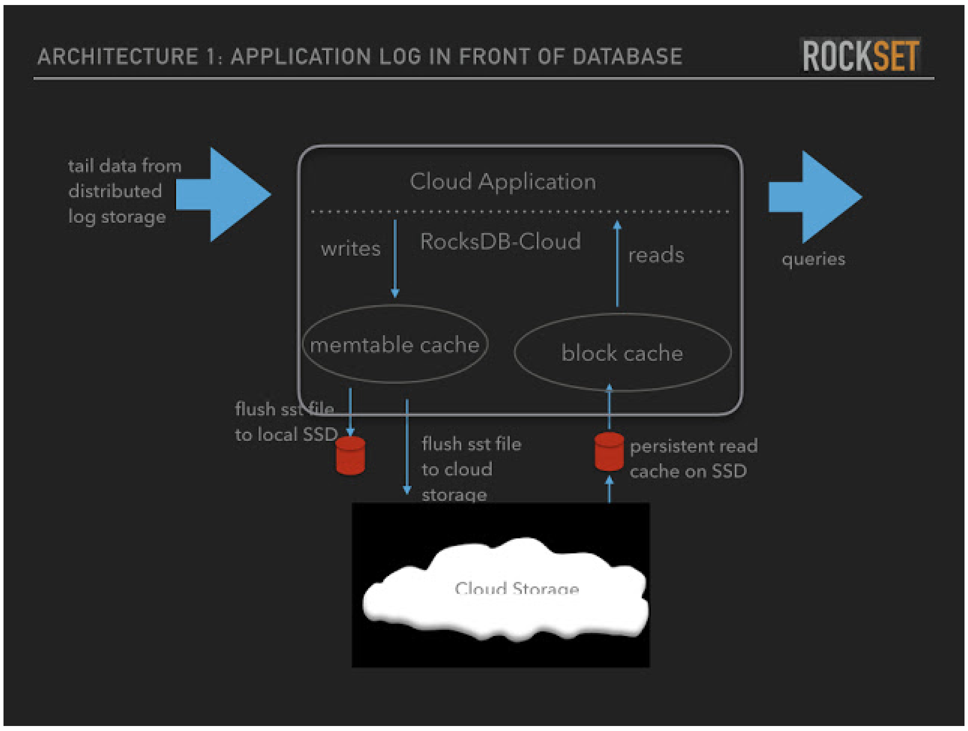
View my complete profile

Compatibility with RocksDB

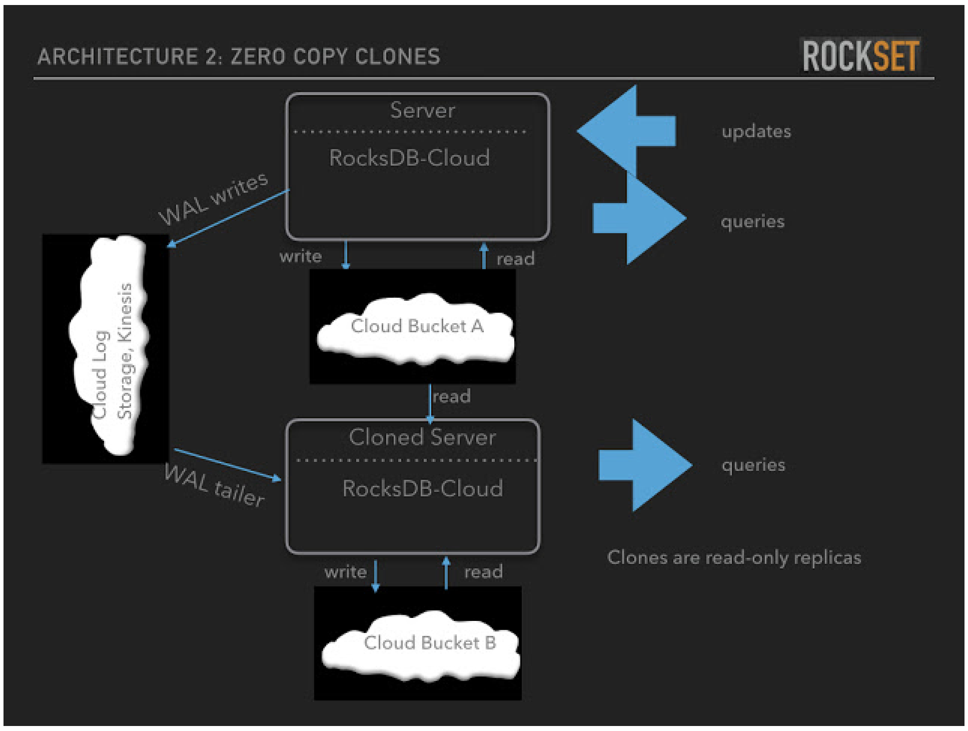
RocksDB-Cloud is API compatible, data format compatible and license-compatible with RocksDB which means that your applications do not have to change if you move from RocksDB to RocksDB-Cloud.

Workload categories

There is a category of workload where RocksDB is used to tail data from a distributed log storage system. In this case, the RocksDB write-ahead-log is switched off and the application-log is in front of the database. The RocksDB-Cloud library persists every new sst file to the cloud-storage. Reads occur by demand-paging-in relevant data blocks from cloud-storage into the locally attached SSD-based persistent cache. This is shown in the following picture.


ARCHITECTURE 1: APPLICATION LOG IN FRONT OF DATABASE
ROCKSET
tail data from distributed log storage
Cloud Application
RocksDB-Cloud
writes
reads
queries
memtable cache
block cache
flush sst file to local SSD
flush sst file to cloud storage
persistent read cache on SSD
Cloud Storage

There is another category of workload where RocksDB is used as a read/write datastore. Applications can issue gets/puts to the datastore. In this case, RocksDB-Cloud persists the write-ahead-log into a cloud-based logging system like AWS-Kinesis. A slave RocksDB-Cloud instance can be configured to tail this write-ahead-log and keep itself updated. The following picture shows how a clone instance keeps itself upto-date by cloning a base-image from the cloud storage and then keeping itself upto-date by tailing the write-ahead-logs.



These two workloads listed above are described in greater detail in a set of architecture slides that I recently delivered at PerconaLive2017 .

In the current implementation, RocksDB-Cloud support only AWS-S3 but won't it be cool if we can have precisely the same api on Microsoft Azure? That is precisely what Min Wei, an engineer from Microsoft, is working on. We are working together to build something useful. Here is a set of slides that I recently delivered at PerconaLive2017 that describes the high-level architecture of RocksDB-Cloud.

If you plan to tinker with RocksDB-Cloud, please start with this example. Come join us in our google group and we can all hack together!

Posted by Dhruba Borthakur at 11:36 PM 4 comments: G+

Labels: cloud storage, flash storage, rocksdb, rocksdb-cloud, tiered storage

Wednesday, April 26, 2017

Research topics in RocksDB

I have been asked by various students and professors about research-and-development-ideas with RocksDB. I remember the first time I was asked about this was by a student at UC Berkeley when I presented at the AMP-Lab-Seminar in 2013. I have been accumulating a list of these topics in my mind since that time, and here it is:

- **Time Series**: RocksDB stores keys-and-values in the database. RocksDB uses delta-encoding of keys to reduce the storage footprint. Can we get better compressibility if the key-prefix is of a specific type? If the key-prefix is a timestamp, can we use sophisticated methods to achieve better storage efficiency? Algorithms in the lines of the delta-of-delta encoding schemes as described in the Gorilla paper are possible candidates to consider. That paper also describes how a an XOR scheme can be used to compress values that are floating points.
- **Byte Addressable Storage**: There are storage devices that are byte address-able and RocksDB's PlainTable format is supposed to be used for these types of storage media. However, supporting delta-encoding of keys and supporting compression (e.g. gzip, zlib, etc) are a challenge in this data format. Design and

implement delta-encoding and compression for PlainTable format.

- **Read-triggered-compactions**: New writes to the db causes new files to be created, which in turn, triggers a compaction process. In the current implementation, all compactions are triggered by these writes. However, there are scenarios that can benefit when read/scan requests trigger compaction. For example, if a majority of read-requests are needing to check for that key in multiple files, then it might be better to compact all those files so that succeeding read-requests can be served with fewer reads from storage. Design and implement read-triggered compactions.
- **Tiered storage**: RocksDB has a feature that allows caching data in persistent storage cache. This means that sst files that reside on slow remote storage can be brought in on demand and cached on the specified persistent storage cache. The current implementation considers the persistent storage cache as a single homogeneous tier. However, there are scenarios when multiple storage systems (with different latency of access) need to be configured for caching. For example, suppose all your sst files are on network storage. And you want to use locally attached spinning disks as well as locally attached SSD to be part of your persistent storage cache. Enhance RocksDB to support more than one tier of storage for its persistent storage cache.
- **Versioned store**: RocksDB always keeps the latest value for a key. Enhance RocksDB so that it can store and allow access to n most recent updates to a key.

In a future blog post, I will describe some of the advantages of running RocksDB on the cloud (e.g. AWS, Azure, etc). Stay tuned.

Posted by Dhruba Borthakur at 12:02 AM No comments: G+

Labels: research, rocksdb, tiered storage

Sunday, November 24, 2013

The History of RocksDB

The Past
It was mid 2011, I had been developing HDFS/HBase for five years and I was in love with the Hadoop ecosystem. Here is a system that can store and query hundreds of petabytes of data without blinking an eye. Hadoop has always focussed on scalability first and Hadoop's scalability was increasing by leaps and bounds. It was apparent that we will be able to store an exabyte of data in a single Hadoop cluster in a few years. I wanted to take on a new challenge...to see if we can extend HDFS's success story from Data Analytics to Query Serving workloads as well.

A Query Serving workload mostly consists of point lookups, random short range scans and point updates. The primary requirement of this workload is *low-latency* queries. On the other hand, a Big Data Analytics query typically involves large sequential scans and joins of two or more data sets with a very low volume of updates if any. Thus, the following year I spent comparing HBase/HDFS and MySQL for a Query Serving workload. The advantage of using HBase is that we can have multiple copies of data within a single data center. I wanted to determine what is needed to migrate a very large Query Serving workload from a cluster of MySQL servers to an HBase/HDFS cluster. This multi-petabyte dataset was stored on spinning disks. After multiple rounds of enhancements to HBase and were able to make it such that HBase latencies were only twice as slow as a MySQL server and HBase was using only three times more IOPs to serve the same workload on the same hardware. We were making steady progress towards our goal... but then something changed!

Flash storage became a reality. The data set migrated from spinning disks to flash. Now, the million dollar question that came up is whether HBase is capable of efficiently using flash hardware. I benchmarked HDFS and HBase with data on flash storage and posted the results in an earlier post.The short story is that the HDFS/HBase of 2012 had a few software bottlenecks because of which it was not able to use flash storage efficiently. It became clear that if data is stored in flash storage, then we need a new storage engine to be able to serve a random workload on that data efficiently. I started to look out for techniques to build the next generation key-value store, especially designed to serve data from fast storage.

Why do we need an Embedded Database?
Flash is fundamentally different from spinning storage in performance. For the purpose of this discussion, let's assume that a read or write to spinning disk takes about 10 milliseconds while a read or write to flash storage takes about 100 microseconds. Network network-latency between two machines remains around 50 microseconds. These numbers are not cast in stone and your hardware could be very different from this one, but these numbers demonstrate the relative differences between two scenarios. What does this have anything to do with application-systems architecture? A client wants to store and access data from a database. There are two alternatives, it can store data on locally attached

disks or it can store data over the network on a remote server that have disks attached to it. If we consider latency, then the locally attached disks can serve a read request in about 10 milliseconds. And in the client-server architecture, accessing the same data over a network results in a latency of 10.05 milliseconds, the overhead imposed by the network being only a miniscule 0.5%. Given this fact, it is easy to understand why a majority of currently-deployed systems use the client-server model of accessing data. (For the purpose of the discussion, I am ignoring network bandwidth limitations).

Now, lets consider the same scenario but with disks replaced by flash drives. A data access in the case of locally attached flash storage is 100 microseconds whereas accessing the same data via the network is 150 micros. Network data access is 50% higher overhead than local data access and 50% is a pretty big number. This means that databases that run embedded within an application could have much lower latency than applications that access data over a network. Thus, the necessity of an Embedded Database.

The above hypothesis does not state that client-server models will become extinct. The client-server-data-access-model has inherent advantages in the areas of data management and will continue to remain prominent in application deployment scenarios.

Aren't there any existing embedded databases?
Of course there are many existing embedded databases: BerkeleyDB, KyotoDB, SQLite3, leveldb, etc. Open-source benchmarks seem to state that leveldb is the fastest of the lot. But not all of them are suitable for storing data on Flash storage. Flash has limited write-endurance; updates to a block of data on Flash typically introduces write-amplification within the Flash driver. Given that we want to run our database on flash, we focussed on measuring write-amplification for evaluating databases technologies.

HBase and Cassandra are Log Structured Merge (LSM) style databases, but it will take a lot of engineering work to make HBase and Cassandra be an embeddable library. Both of them are servers with an entire ecosystem of built-in management, configuration and deployments. I was looking for a simple c/c++ library: leveldb was the apparent first choice for our benchmarking.

Why was leveldb insufficient for our purpose?
I started to benchmark leveldb and found that it was unsuitable for our server workloads. Leveldb was a cool piece of work but was not designed for server workloads. The open-source benchmark results looks awesome at first sight, but I quickly realized that those results were for a database whose size was smaller than the size of RAM on the test machine, i.e. the entire database has to fit in the OS page cache. When I performed the same benchmarks on a database that was at least 5 times larger than main memory, the performance results was dismal.

Leveldb's single-threaded compaction process was insufficient to drive server workloads. Frequent write-stalls caused 99-percentile latency to be tremendously large. Mmap-ing a file into the OS cache introduced performance bottlenecks for reads. Leveldb was unable to consume all the IOs offered by the underlying flash storage.

On the other hand, I was seeing server storage hardware evolve fast in different dimensions, For example, an experimental system where a storage volume is striped across 10 flash cards can provide upwards of a million IOs per second. A NVRAM based storage can support a few million data accesses per second. I would like to use a database that can drive these types of fast storage hardware. A natural evolution of flash storage could lead us to storage that has a very limited erase cycles and I envisioned that a database that can allow elegant tradeoffs of read amplification, write amplification and space amplification would be a dire necessity for these type of storage. Leveldb was not designed to achieve these goals. The best path was to fork the leveldb code and change its architecture to suit these needs. So, RocksDB was born!

The vision for RocksDB
1. An embedded key-value store with point lookups and range scans
2. Optimized for fast storage, e.g. Flash and RAM
3. Server Side database with full production support
4. Scale linearly with number of CPU cores and with storage IOPs

RocksDB is not a distributed database. It does not have fault-tolerance or replication built into it. It does not know anything about data-sharding. It is upto the application that is using RocksDB to implement replication, fault-tolerance and sharding if needed.

Architecture of RocksDB
RocksDB is a C++ library that can be used to persistently store keys and values. Keys and values are arbitrary byte streams. Keys are stored in sorted runs. New writes occur to new places in the storage and a background compaction process eliminates duplicates and processes delete markers. There is support for atomically writing a set of keys into the database. Backward and forward iteration over the keys are supported.

RockDB is built using a "pluggable" architecture. This makes it easy to replace parts of it without impacting the overall architecture of the system. This architecture makes me confident that RocksDB will be easily tunable for different workloads and on different hardware.

For example, one can plug in various compression modules (snappy, zlib, bzip, etc) without changing any RocksDB code. Similarly, an application can plug in their own compaction filter to process keys during compaction; an example application can use it to implement a *expiry-time* for keys in the database. RocksDB has pluggable memtables so that an application can design a custom data structure to cache their writes, one example is prefix-hash-memtable where a portion of the key is hashed and the remainder of the key is arranged in the form of a btree. The implementation of a sst file is pluggable too and an application can design their own format for sst files. RocksDB supports a MergeType record that allows applications to build higher level constructs (Lists, Counters, etc) by avoiding a read-modify-write.

RocksDB currently supports two styles of compaction: the level style compaction and the universal stye compaction. These two styles offers flexible performance tradeoffs. Compactions are inherently multi-threaded so that large databases can sustain high update rates. I will write a separate post on the pros and cons of these two styles of compaction.

RocksDB exposes interfaces for incremental online backup which is needed for any kind of production usage. It supports setting bloom filters on a sub-part of the key, which is one possible technique to reduce iops needed for range-scans.

RocksDB's apis are stackable, which means that you can wrap lower level apis with higher level easy-to-use wrappers. This is the topic of a future post.

Potential Use-cases of RocksDB
RocksDB can be used by applications that need low latency database accesses. A user-facing application that stores the viewing history and state of users of a website can potentially store this content on RocksDB. A spam detection application that needs fast access to big data sets can use RocksDB. A graph-search query that needs to scan a data set in realtime can use RocksDB. RocksDB can be used to cache data from Hadoop, thereby allowing applications to query Hadoop data in realtime. A message-queue that supports a high number of inserts and deletes can use RocksDB.

The Road Ahead
There is work-in-progress to make RocksDB be able to serve data at memory speeds when the entire database fits in RAM. RocksDB would evolve to be compatible with highly multi-core machines. With the advent of super-low-endurance-flash storage, we expect RocksDB to store data with minimal write amplification. Maybe sometime in the future, someone will port RocksDB to the Android and iOS Platform. Another nice feature would be to support Column Families to support better clustering of related data.

I am hoping that software programmers and database developers would use, enhance and customize RocksDB for their use-cases. The code base is in http://github.com/facebook/rocksdb. You can join the Facebook Group https://www.facebook.com/groups/rocksdb.dev to participate in the engineering design discussions about RocksDB.

Posted by Dhruba Borthakur at 8:13 PM 7 comments: G+

Labels: database, flash storage, hbase, lsm, rocksdb
Location: Sunnyvale, CA, USA

Wednesday, October 16, 2013

The SPARROW Theorem for performance of storage systems

Introduction
I have been measuring the performance of a number of storage systems and here is a post of what I have learnt so far. I do not claim that any of this is new, it is written from my perspective and is a very opinionated piece.

Database and Storage Systems have evolved over the years, starting from storing data in spinning-disk based storage system to solid-state-storage (SSD) and random-access-memory (RAM) storage. These storage hardware have very different performance characteristics. The difference in the physical constraints of these storage hardware means that we might need to use different methodologies to measure their performance. In the following discussions, I refer to OLTP workloads where each record is small (smaller than 4K) and they are read randomly.

Spinning Disks

Spinning disks can typically sustain 100 to 200 IO per sec. Random accesses are typically bottlenecked by the number of IOs that the disk can service. The sequential access speeds of disks can reach upto 80 MBytes/sec. Most database system that run on spinning disks are optimized to reduce the number of random reads to storage that it consumes to service a specified workload. Btree implementations (like Innodb, Berkeley-DB, etc) tries to reduce the number of random reads by caching some part of the working set in RAM.
LSM databases (like Apache HBase) converts random writes to sequential writes to get better database performance. For all these systems, reducing the number of random disk access directly improves performance, especially when the workload is mostly small random reads or short scans.

Solid State Devices (SSD)
SSDs have physical constraints that are different from spinning disks. An SSD can typically perform about 60K to 100K IO per second. This is orders of magnitude larger than what a spinning disk can possibly do. The throughout of an SSD can very from 300MBytes/sec to 600 MBytes/sec depending on the manufacturer and the percentage of free space on the SSD.

If a storage software issues 100K IO per second and each IO is 4 K, then the total data bandwidth needed is 400 MBytes/sec which is close to the maximum throughput of the SSD. If you run a OLTP benchmark on SSDs, it is likely that you are bottlenecked because of one of two reasons:
(1) you have used up all the random IOs per second offered by the device or
(2) you have maxed out the SSD data bandwidth.
It could be either of these two constraints. If you are reaching the maximum data throughput of this device, then reducing the number of random accesses to storage might not improve performance for your system. You need to reduce the total storage bandwidth too.

So, what consumes the storage bandwidth of the device? This bandwidth is consumed by reads and writes done by the storage software on behalf of the application.

Read Amplification (RA)
Read Amplification is the ratio of the number of storage bytes accessed to satisfy a single read request of 1 byte. For example, if a btree-based storage system has to read a single 4K size page from storage (assuming that the index pages are cached in RAM) for every read request of 1 byte, then it has a read amplification of 4K. On the other hand, if an LSM based database needs to consult three storage files to satisfy a single read request of size 1 byte and the block size of each file is 4K, then its read amplification is 12K (ignoring the existence of bloom filters). My colleague Mark Callaghan has an awesome post about Read Amplification.

Write Amplification (WA)
Write Amplification is the ratio of the number of storage bytes accessed to satisfy a single 1 byte write request to the database. **WA** includes the write amplification caused by the database software as well as the write amplification generated by the SSD itself.

For example, if a btree-based storage system has to write 1 byte into the database, it has to read in the 4K page into memory, update it and then write it back, thereby incurring a write amplification of 8K. A btree-based storage-system typically overwrites the SSD page in place thereby incurring additional write amplification in the SSD layer too.

On the other hand, an LSM storage engine typically writes the new 1 byte to a new place on SSD and has a write amplification of 1 (not including compaction).

For transaction logging, a btree-based system needs a redo log and an undo log which means that **WA** is further increased by 2 for these systems. An LSM based system needs only an redo log which causes **WA** to increase by 1.

But the above does not mean that LSM systems are better. Before I try to explain why it is so, please allow me to write about Space Amplification.

Space Amplification (SA)
Space Amplification is the number of storage bytes needed to store 1 byte of information. Storing multiple indices of the same data increase the storage requirements but could decrease read latencies. Space Amplification is also caused by the internal fragmentation and padding. An LSM database can have the same key with older versions of the data in multiple files... this too can cause Space Amplification.

The SPAce, Read Or Write theorem (SPARROW)

The SPARROW Theorem states that:
1. **RA** is inversely related to **WA**
2. **WA** is inversely related to **SA**

If you want to reduce the Read Amplification caused by a specific workload, then you can achieve it only if you incur higher Write Amplification. Conversely, if you want to reduce the Write Amplification caused by a specific workload, then you have to incur higher Read Amplification to achieve that goal. (This assumes that we maintain Space Amplification constant at all times)

Similarly, the only way to reduce Space Amplification caused by a specific workload is to tune the system in such a way that it can sustain a higher Write Amplification. Conversely, if you want to reduce Write Amplification, then you have to incur higher Space Amplification for the same workload.

Implications of the SPARROW Theorem
There isn't any storage system that can escape from the clutches of the SPARROW Theorem. A single system architecture CANNOT reduce all three SA, RA and WA. I am talking about the architecture and not of implementations.

A practical Database implementations would always try to reduce its current SA, WA and RA by optimizing its code and algorithms. But once all it's code and algorithms are optimized, then it won't be able to improve all three dimensions at the same time. Its performance will be confined by the walls outlined by the SPARROW Theorem.

Given the above fact, it would be great to have most database system be configurable so that an administrator can tune each of these three dimensions based on the workload and the hardware that it is running on. This will result in a highly flexible database system architecture that can sustain a myriad of workloads.

Posted by Dhruba Borthakur at 11:29 PM No comments: G+

Labels: cassandra, hbase, rocksdb, storage software

Home

Subscribe to: Posts (Atom)

Simple theme. Powered by Blogger.