2012 IEEE International Conference on Cluster Computing Workshops

Sedna: A Memory Based Key-Value Storage System for Realtime Processing in Cloud

Dong Dai
Computer Science College
University of Science and Technology of China
Suzhou Institution of USTC
daidong@mail.ustc.edu.cn

Xi Li
Computer Science College
University of Science and Technology of China
Suzhou Institution of USTC
llxx@ustc.edu.cn

Chao Wang, Mingming Sun
Computer Science College
University of Science and Technology of China
Suzhou Institution of USTC
saintwc,mmsun@mail.ustc.edu.cn

Xuehai Zhou
Computer Science College
University of Science and Technology of China
Suzhou Institution of USTC
xhzhou@ustc.edu.cn

***Abstract*—Comparing with the traditional disk based distributed storage system, RAM based storage has been proven to be an effective way to accelerate realtime applications processing speed. In this paper, we propose a memory based distributed Cloud storage system called Sedna. Managing 'big data' across lots of commodity servers, Sedna provides high scalability, simple effective data access APIs with data consistency and persistency, and a new trigger based APIs for realtime applications. To guarantee the scalability with low latency, we design and implement a hierarchical structure to manage huge size data center which is simple and effective. Except the high speed provided by memory based storage, Sedna absorbs advantages from state-of-art cloud programming frameworks, and gives programmers a new way to write massive data realtime applications. These data read/write triggers APIs are necessary but are missing parts of modern distributed storage system. Experiments and examples show Sedna achieve comparable speed to widely-used distributed cache system, and provide a more efficient way to use distributed storage.**

I. Introduction

Realtime processing has been a new wave in cloud computing along with the rapid development of large scale Internet service. The speed of data generation is much faster, and the requirement for timely analysis is also more exigent. As the world's largest social network site, Facebook has been facing the realtime trend for years [12]. These applications require high speed concurrent write/read access to a very large amount of realtime data. For example, the Realtime Analytics [8] need to read and analysis data generated in realtime by Scribe [10]. Facebook Message [3], which would index the rapidly growing data set like messages and emails, dynamically and incrementally for fast random lookups.

Even the realtime trends has emerged for years, cloud storage system has not been able to catch up with it. Realtime applications basically need to store large scale data in higher throughput and randomly read them back, however, this is still a big challenge for disk based cloud storage system due to the latency of disk I/O operations. In 2009, RAMCloud [23] proposed to use RAM as the main storage material instead of disk in cloud. It showed the possibility and the reasonability of using RAM or other high speed random access device to substitute disk in data centers. RAMCloud concentrates on the data persistency and recovery strategy, However, we believe the scalability is another important problem when changing from disk storage to memory storage. Considering the storage capacity of RAM(32-64GB) and Disk(16-32T) in a standalone server, to build the competitive storage system, we need at least hundredfold servers in a RAM cloud. Managing so many servers without delaying the RAM speed is a great challenge for researchers. In Sedna, we focus on the scalability issues and design a hierarchical structure to manage large-scale cluster.

This paper is supported by RFP-07-999 Cisco Research Foundation, Natinal Science Foundation NSFC-60873221, and Jiangsu Research Project-BY128

With the high data read/write speed, it is still complex and difficult to program a realtime application in cloud platform. Current realtime programming models in cloud like MapReduce Online [21], S4 [20], Spark [24] have some limitations facing different kinds of realtime applications. It is unavoidable because the storage layers they are running on only provides simple data read and write APIs. This means if applications want to get the newest data from storage layer, it has to continuously read it over and over again. These simple read/write Apis are acceptable for batch based applications but do not fit for realtime ones. In Sedna, besides the straightforward data access Apis, we provides trigger based Apis which are useful for upper layer realtime programming models. According our use case in realtime search engine, we find it is easy to implement a programming framework for different kinds of realtime applications based on Sedna.

This article is decomposed as fellows: section 2 list some problems facing realtime era in cloud environment. After that we will describe the storage layer of Sedna especially the hierarchical structure of Sedna detailed in section 3. In section 4, we present the trigger based read/write apis Sedna provides for realtime applications, and show a use case based on it. In section 5, we show experiments on Sedna's performance comparing with current popular distributed memory cache system. We also use this performance analysis to demonstrate the efficiency of Sedna's hierarchical structure. In section 6, we list some related works and show the difference between them and Sedna. Finally, this paper is concluded in section 7.

II. Realtime for Storage

Realtime applications have emerged and been popular in the state-of-art internet services for years. It call for numerous abilities which most of current storage solutions in cloud have not obtained. Fig. 1 illustrates a typical realtime cloud platform. The under layer storage system needs to response to the write and read requests from clients and needs to support the realtime application scheduler.

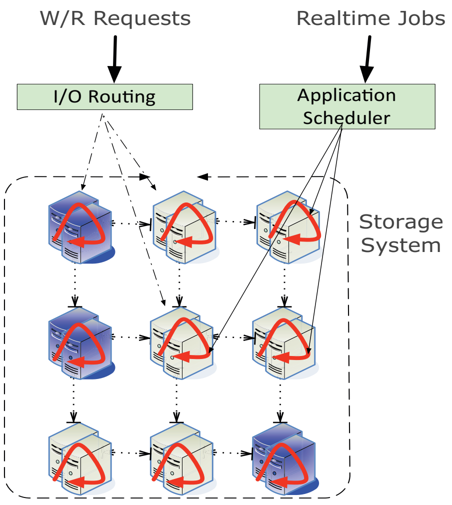


Fig. 1. Storage Layer in Realtime Cluster

Knowing this architecture of realtime cluster, and critical characters of realtime applications summarized below, we can get more clues about what a modern storage is required to meet the demands of the realtime applications.

- Need to store large amount of small data. Small and fragile raw data pieces have been the most common data type in current cloud applications. According to the statistics provided by Twitter, 350 billion tweets(each tweet is limited less than 140 bytes) were posted in Twitter from near 400 million users per day.
- Need to store data in high throughput. Still take Twitter as an example, with a rate of billions of messages and millions of relationship changes every day. To store or process these newly generated data requires high write throughput ability. Further more, the situation becomes worse when each message need to be written several times for reliability.
- Need to random reads in high speed. Once the newly generated data was written into the storage system, we need corresponding handlers to process it immediately. Take a realtime search engine as example, the time interval between the newly data sprawled and indexed should be short.
- Need to help users process data in realtime. Typical realtime applications response to users within seconds. This response was generated through a serial complex computing.

A. Modern Storage Model

Storage model includes several aspects including data representation, data storage strategy, data retrieving and other merits like performance, availability, scalability etc. In following sections, we describe a storage model that Sedna uses to adapt real time applications.

1) Representing Data: Modern cloud applications typically store a large amount of small data pieces, and usually access them randomly, or at least, in a way that is different from how these data was collected. Key-Value fashion is quite fit for small data piece, and supports random read/write well. Though original key-value is a flatten data model, we can add extra information in the 'key' part to represent hierarchical data space. Or, divide data into different tables like Bigtable does. Finally, key-value pair is naturally fit for current popular MapReduce [14] programming model.

2) Programming Model: MapReduce is a popular distributed programming model introduced by Google. It simplifies large scale distribution applications by dividing them into map stage and reduce stage. MapReduce was widely used in batching jobs, however, it can not support realtime processing well. In realtime applications, the interactions between users and cloud platform became continuous. Once data arrived, we need to process it immediately and generate new results, which will affect what users see in clients or begin next round processing. To support this programming model, the storage system should provide applications the ability to watch on their interesting data pieces, store the intermediate results produced in processing safely even failure happens, and send these results to next phase immediately instead of waiting for writing into local disk.

3) Speed Requirement: Realtime applications running on cloud platform require high read/write speed which only can be provided by memory storage and high speed network connection. 10Gb network has been popular in modern data centers, so the max transmission speed of network has exceeded the disk speed. To accommodate this situation, higher speed storage material began to noticeable for system architecture. Memory is one of the fastest and its price has been more reasonable now. We can easily put the whole data set into a cluster's main memory. Also, memory is fit for small data chunks that are typically less than 4KB(MB).

4) Scalability Requirement: Once we store most data into memory, comparing the disk size per server, we will need more servers to provide competitive storage capacity. Besides, the internet scale service needs storage system changes drastically according to its popularity. In a word, modern storage system needs the ability of managing more servers to provide scalable storage and computing power.

B. Sedna Design Idea

Sedna is designed following the storage model described in section 2.A. Here we list some basic design principles:

- Sedna stores all the recent data in memory instead of disk. All data stored in Sedna is arranged in key value

TABLE I
SUMMARY OF SEDNA

Problems	Technique	Advantages
Partitioning	Consistent Hashing	Incremental Scalability
Replication	Eventually Consistency	Higher R/W Speed Flexible policy
Node Management	ZooKeeper Sub cluster	Avoid the single node failure
Read&Write	Read Trigger and Lock-Free Writes	Speed up the Processing by data push and low latency
Failure Detection and Handling	Heart-beat protocol and Active detection	Reduce the failure detection time
Persistency Strategy	Periodically flush or write-ahead logs	Different speed and availability according users' needs

fashion, and the key was extended implicitly by Sedna to provide hierarchical data space.

- Sedna uses a subset of cluster to manage overall consistent status. This way avoids the single failure problem which often draws criticism on the availability of Hadoop [4] or other GFS [15] like storage system, and provides the potential to extend data center into huge size without conspicuous penalty on speed.
- Sedna provides read/write trigger apis to help programmers write realtime applications. By pushing recently changed data to corresponding clients and triggering specify user-define actions, Sedna can reduce the effort to write complex realtime applications dramatically. Besides, Sedna provides distributed data structure to help users write realtime applications or streaming processing applications.

III. SEDNA ARCHITECTURE

The architecture of a distributed storage system in production environment is quite complex which includes data persistence component, scalable and robust solution for load balancing, servers management, failure detection, handling and recovery, replica consistency, state transfer, concurrent mechanisms. In this section, we focus on several core conceptions and implementation in Senda: partitioning, replication, data read and write, as well as Zookeeper usage. In next section, we introduce how Sedna provide a new perspective to program realtime applications. Table 1 presents a summary of techniques employed by Sedna and their advantages.

A. Overall Architecture

Fig. 2 shows the overall architecture of the Sedna system. As a cloud storage layer for distributed applications, the requests from clients usually was directly routed to a server in data center. Servers in Sedna cluster are divided into two categories: the upper layer sub-cluster which we call ZooKeeper [11] cluster and ordinary servers. Each server in data center runs nearly the same components except for the ZooKeeper parts. A complete Sedna instance on one server was logically divided into local part and distributes part. Local part includes the local memory management layer and persistency layer. The distribute part was implemented based on ZooKeeper, which will help us maintain a consistent status information of the whole cluster. The top layer up cluster status manager layer in Fig. 2 contains components which are pluggable modules providing different functionalities, like replica management, nodes management, data balance, etc.

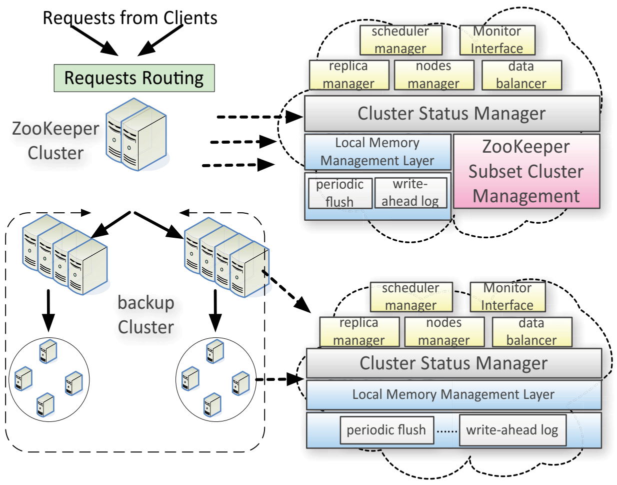


Fig. 2. Overall Architecture of Sedna

B. Partitioning

Sedna uses distributed hash table to separate data across the cluster. The distributed consistent hash ring looks like Fig. 3. It was equally divided into millions of slices, so every slice represents a sub-range of INTEGER(which indicates the key range). What's more, each sub-range is called a virtual node, which consists sequential numbers. When data arrives, its key will be hashed to an integer, then mod to a virtual node. Every data in a virtual node will be stored in one server(**r1**), and replicated in other two servers(**r2**, **r3** as the figure shows). The real servers that store data were named real nodes comparing with virtual nodes.

The simplest consistent hash algorithm is problematic as its non-uniformity when servers join and leave the cluster frequently. Virtual nodes [2] strategy was proposed to reduce the imbalance in a cluster. In Sedna, virtual node is the minimal continuous block for data storage. We record all the virtual nodes' status including its capacity, read/write frequency. Besides, we also maintain a imbalance table for all the real nodes computed from the virtual nodes' status. This information is calculated and stored locally, and periodically updated to ZooKeeper cluster. It is only necessary to update the imbalance table, which is a quite small comparing with the virtual nodes number.

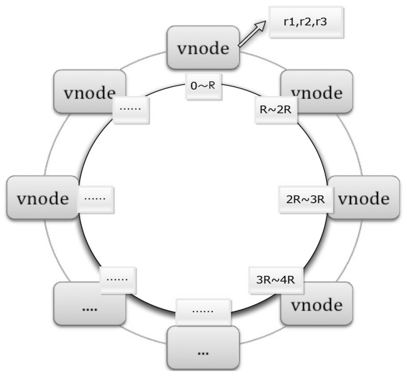


Fig. 3. Virtual Node Ring of Sedna

C. Replication

Every data stored in Sedna system have at least other two copies to guarantee the data availability. We only provide the eventually consistency among different copies using quorum algorithm. Use **R** denotes the minimal number of nodes that must be connected when read data from **N** replicas, **W** denotes the minimal number of nodes that must be connected when write data from **N** replicas. The values of **R** and **W** are subject to following two constrains:

$$R + W > N$$

$$W > \frac{N}{2}$$

This two formulas guarantee the consistency eventually. For example, if there are 3 copies for each data, and **R** equals 2, **W** equals 2. These two formulas are satisfied. In this situation, once any two writes among these three copies success, the read operations will need to wait until at least two copies are consistent because we need at least two reads in these three copies.

When receiving a read request, local running Sedna service requests all the corresponding real nodes to get data with timestamp, then checks for **R** equality. If there are more than **R** equal data, the Sedna service will return corresponding value to clients. Similarly, when encountering write requests for given keys, Sedna service generates the requests were sent to all the relative nodes, if more than **W** nodes return the same version number then the write is considered success.

As we want to keep three copies stored in Sedna for every piece of data, any server failure will cause part of replicas broken. To maintain the data availability, Senda use read recovery to do replica recovery after nodes failing appears. Reads in Sedna system need to send requests to all the three real nodes that store replicas. According to the 'timeout', 'refuse' response from these real nodes, Sedna service will determine whether the servers have failed, and check their existence by asking the ZooKeeper service. If it does fail, reads operation will start a data duplication task asynchronously to copy the lost replica from other healthy nodes and finally change the data mapping information stored in ZooKeeper service. The possibility of lost all the three replicas in a very short time during which no reads nor writes happens can be ignored, even this happens, like the power shortage of the cluster, we can still recover the data from lost by the periodic data flushing.

D. Node Management

When a node joins the Sedna cluster, it will start the local memory storage firstly, then connect to the ZooKeeper service for status synchronization. If the ZooKeeper service has not been initialized, it will start a initial procedure instead of normal start-up procedure. After checking the existence of ZooKeeper service, newly joined node will start its Sedna Service. The service will firstly try to register itself to ZooKeeper by creating a ephemeral znodes under the *real_nodes* znode. After this, it will start number of threads (according the configuration) to ask for virtual nodes and store them locally. This action will change the mapping information between virtual node with real node, which is stored in ZooKeeper service, so the local Sedna service need write into ZooKeeper service, which means changing the value of a znode in ZooKeeper.

When a real node fails or shuts down accidentally, the heartbeat signal which is kept between this server and ZooKeeper service will lost. This will make ZooKeeper service aware of the real node's lost. Sedna does not need to do anything. Recovery work will be started when we read or write data that was stored in this real node.

In Sedna, the virtual node number is a abstracted as a configurable parameter, however, once it is set, we can not change it unless restart the Sedna cluster. The retrieving data threads mentioned above is highly related with the number of virtual nodes. Typically, we store about 100 virtual nodes in a one real node, so if a cluster contains 1,000 servers at most, the total virtual nodes number can be set to 100,000. As every real node stores 100 virtual nodes in average, the data retrieving threads number could be 16 or 8.

E. ZooKeeper Cluster

ZooKeeper cluster is the critical part to maintain virtual node distribution information in Sedna. Besides, it also provides node existence information for all nodes in Sedna cluster, therefore its performance will determine the Sedna performance. In this section, We analysis the usage of ZooKeeper in Sedna and show its performance is fit to be used in large scale storage system to provide consistent status information..

We have known that ZooKeeper is much more preferable for read than write-intensive operations, so mostly Sedna read the information from ZooKeeper service instead of writing. There are two general situations that the data stored in ZooKeeper is required to be modified: 1) when a Sedna cluster boots at the first time, it needs to create znodes in ZooKeeper, which each znode represents for a virtual node, and the overall number of virtual nodes may be millions according to the cluster size; 2) whenever a real node in cluster leaves or a new node joins, Sedna updates these information into ZooKeeper by setting znode's value. In the first situation, lots of creation operations will take a long time when the virtual nodes number is large, but it only happens once when the Sedna cluster firstly starts up. In the second situation, writes in ZooKeeper is much faster (in milliseconds) than the frequency of new nodes join, so it will not affect the Sedna performance.

ZooKeeper's read performance might be another potential bottleneck of Sedna. To avoid this bottleneck, we have three strategies: 1) use local cache. When cached data was found invalid (target node returns 'reject' or 'timeout'), Sedna reads from ZooKeeper and updates local cache. 2) each server starts a periodical thread to synchronize local cache with ZooKeeper data, which we call it the lease time. To reduce the unnecessary updates, lease time will reduce to half if there are lots of changes in ZooKeeper in last lease time, and grow to double if no change in last lease time. 3) whenever updates in ZooKeeper, it will be recorded in a separate znode directory as Sedna only refreshes modified data. In Sedna, we do not use ZooKeeper's watch mechanism because if there are many nodes watching the same znode, any change will result in an uncontrollable network storm.

F. Basic APIs

1) Write: Since random writes were quite common in Internet service, the Sedna system allows writes on the same key parallel from different sources without lock mechanism or lock synchronization. Sedna provides *write_latest()* and *write_all()* for users, both of which work without lock. Data stored in Sedna are timestamped and writes with newer timestamp will successfully overwrite data with older timestamp. When the *write_latest()* request was received by a real node, local Sedna service will compare the request's timestamp with current stored data's timestamp. If current data is newer, it will reply a '$outdated$' to the request, else send a 'ok'. When the *write_all()* request is received by a real node, it will only compare the request's timestamp with the element that came from the same source server in value list. If newer, it updates the element and returns 'ok', else just returns '$outdated$'.

When application calls the *write_latest()* api, it will receive one of three possible replies: 'ok' means the write is successful, '$outdated$' means it is outdated, and '$failure$' means error happens and Sedna will start a recovery task asynchronously. *write_all()* is pretty like *write_latest()* except updates happened on one element of value list instead of all the value.

2) Read: Corresponding with the write apis, Sedna also provides *read_latest()* and *read_all()* for applications to fast access the data set. When applications call the *read_latest()*, it will receive the most fresh value no matter it was written by which node. However, the *write_all()* will return all the values corresponding that key.

3) Realtime API: In cloud focusing on the realtime processing, the interval between data arrival and proceeded is more important than batch processing system. That means upper layer applications need more sensitive way to monitor the data write operations and get useful data timely. In next section, we will introduce how Sedna support such operations and how program realtime applications based on Sedna.

IV. Triggers: Sedna for Realtime

To support realtime applications, Sedna adds trigger based APIs as fundamental functionality. Trigger based mechanisms have been widely used in database system to ensure the data integrity in the past several decades. It was a special storage procedure which will be executed once some specific events happened, like insert, update, or delete.

However, these database triggers are quite different from the trigger based APIs Senda provides. Firstly, Sedna supports trigger based programming model which programmers can use to build their applications running in a Cloud environment. For example, to implement the Indexer of a search engine, we can define a Sedna trigger monitoring on the web pages data set and perform text parsing and index establishing. Whenever any update on the web pages data set, this trigger will be scheduled to execute tasks according programmers definition. Secondly, trigger system in Sedna is much more complex than that in database. Sedna provides the filter mechanism for triggers. Filters are necessary because the high frequency of updates in Cloud applications. Programmers can filter part of them according user-defined conditions.

A. Trigger Based Programming

A typical application based on Sedna's trigger APIs usually includes several triggers to finish a complex job. The left picture in Fig. 4 shows three triggers - A, B, and C. Trigger A monitors the initial data set and executes tasks which will output its intermediate results into distributed file system, and the variation of trigger As outputs will push forward trigger C, and so on. The loop body of an iterative tasks was implemented by the interaction among these three triggers. In an iterative task, there also would be a stop condition which used to terminate the infinite loop. Programmers can use filters to stop next phase execution.

B. Flow Control

The trigger based system is readily comprehensible, however, there are many difficulties when we want to provide it as a programming model for applications. One of the most important challenges is the Ripple effect caused by multiple triggers. We use the Fig. 4 to demonstrate this problem. The right picture in Fig. 4 shows that the trigger A and D will push forward the same trigger C, and the output of trigger C will push forward A and D in turn. The circle formation of triggers is the way programmers used to support complex applications, however, in this situation, once this loop began to execute, the frequency of triggers activation will be doubled in each round and finally flood the whole cluster. This would be much more serious when different users share the same Sedna cluster as they do not know much about others applications and might form lots of such kind circles. This overflow situation was suppressed in Sedna because the filters will give every application default trigger interval. If value changes during this interval, it would be safe to discard them as the most fresh data matters most.

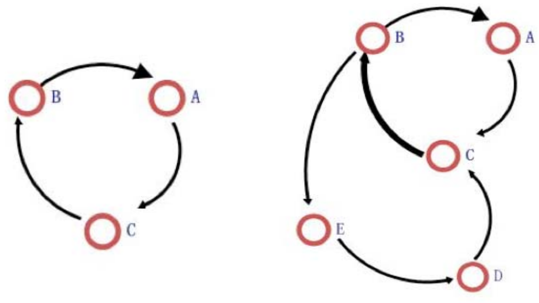


Fig. 4. Multiple Triggers Execution. A Case Study

C. Monitor & Filter

Monitor and filter are the key concepts to support realtime apis in Sedna. As we have described in section 2, Sedna uses key-value fashion to store data and it extends the key field of data to support hierarchical data space. So the least unit programs can monitor would be a key-value pair, and they also can monitor Tables which is a collector of key-value pairs, or monitor a Dataset which is a collector of tables. As Fig. 5 shows, all the storage table includes two additional columns: Dirty and Monitors. Every time data was written in this row, table, or Dataset, the Dirty filed will be written automatically. When programmers register a monitor on specific data, that program will add itself in the corresponding Monitors filed.

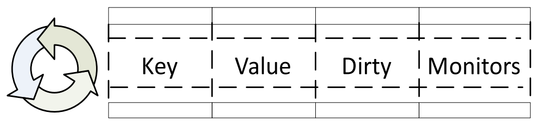


Fig. 5. Extended Key-Value Pair

Once Sedna started, it will start several threads according to the data size to scan the Dirty and Monitored fields sequentially. Whenever Dirty flag was found, that data piece will be sent to corresponding filters according to the monitor fields of that data piece. With the respect to the implementation, filters are much easier than monitors. It is an assert function to check whether specific conditions have been fulfilled. In Sedna, the assert functions in Sedna would contain four arguments, two for the new data, other two for the old data. This is useful when we need to determine when the triggers should stop.

D. Realtime APIs

Listing 1 shows an example application written in Java using Sedna's APIs. To finish specific tasks, Programmers need to write an action class (**MyAction**) extending the basic **Action** class provided by Sedna APIs. The override action function in MyAction has three input parameters: the key part of input data, a list of values share the same key, and a Result. Result provides a safe way for programmers to write processing results into distributed storage system paralleled. Besides the action class, a complete application should contain the monitors, filters, and the output setting. In our example, we build a **TriggerInput** instance from the **DataHooks** and a user-defined **MyFilter** class that bases on Sedna's **Filter** class. And the **setActionClass** method of class Job glues these things together. In this method, the **Input**, **Output**, and the **MyAction** work together to build a trigger job. Finally, in the main function, we should schedule the just created job to run. In the practical environment, Programmers should give a job a timeout measurement to avoid infinite execution.

The assert function in **Filter** class includes two noticeable features. First of all the assert function will be called on each key-value pairs where programmers set hooks on. Sedna runtime will choose to process this key-value pair or not according assert's return value, so the assert function should be as simple as possible. Secondly, the assert function provides users four parameters including the old key-value pair and new key-value pair. The reason to design it this way is, in lots of condition, the filter need to compare the difference between before and after the data updates. An obvious example is the stop condition of iterative tasks.

Listing 1. A Domino Task Example

```
public static class MyAction extends
  Action<Key, Iterator<Value>, Result>{

  protected void action(Key, Iterator<Value>,
      Result){
    //do your stuff here
  }
}
public static class MyFilter exntens Filter{

  protected boolean assert(OldKey, OldValue,
    NewKey, NewValue){
    //return true/false
  }
}

//Job Configuration
public static Job confJob(){
  h1 = new DataHooks();
  f1 = new MyFilter();
  i1 = new TriggerInput(h1,f1);
  o1 = new TriggerOutput();
  Job job = new Job();
  job.setJarByClass(this.class);
  job.setActionClass(MyAction.class,i1, o1);
  return job;
}

public static void main(String[] args){
  Job job = configureJob();
  job.schedule(Timeout);
}
```

In section 5, we will describe a real cloud realtime application based on Sedna storage layer and the trigger based programming model.

V. Sedna Usage Case

At the very beginning, the Sedna system was designed and implemented for a micro-blogging search project, so in this section we will briefly describe the usage of Sedna in micro-blogging search engine.

Micro-blogging search engine is a useful tools we designed for micro-blogging system like Twitter and Sina Weibo to find the most fresh discussion trends. In addition to the ability of usual search engines have, like indexing, parsing, and ranking according the messages' timeline, we also consider other factors that may affect the importance of the contents: 1) the social connections between the searcher and the messages' authors, 2) the importance of a message according its re-tweet or comment count, 3) the specialty of the relative messages' author. In these factors, the social connection of users, existing messages' re-tweet and comment count are changing all the times, so we need a storage system which can help our applications process these data more quickly. Except for existing messages, there are continually growing new messages in micro-blogging systems, we need a storage middle layer like Sedna to help us process newly generated messages timely.

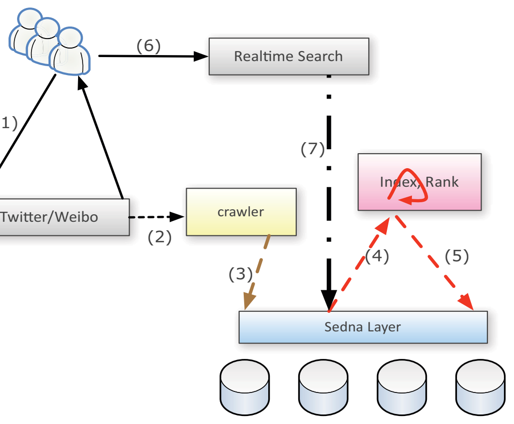


Fig. 6. Micro-Blogging Search Engine

Fig. 6 demonstrates the realtime search engine. Step 1 means users began to write their new tweets, then this tweets will be retrieved by our crawler in step 2. The interval between Step 3 and step 4 will be quite small because the help of Sedna's trigger based apis. Once newly data was processed and written back to Sedna storage layer, users can query their tweets in step 6 and 7. As a realtime search engine, the time between (1) and (7) should be less than several minutes.

Sedna was mainly used in two different aspects: 1) The storage layer, whenever the spider crawls data, not only includes the messages but also includes the social connection information, it will store this data into Sedna using *write_all* api. 2) Process layer, there are different trigger based jobs using Sedna's realtime APIs. For example, one of them is to calculate users' relationships, this job will register monitors on users' relationship data, when data changes, the job will start to run to calculate new social graphic, another job monitors on users' messages data, when new messages were scrawled, the job will start to parse these messages, index them and update the inverted index table.

This real application shows that the Senda is extremely fit for the realtime massive data processing in cloud platform.

VI. Experiments

In this section, we mainly compare the read/write performance of Sedna with Memcached [7]. Sedna uses modified Memcached as its local memory storage system, so its performance in standalone mode was limited by Memcached. However, the comparison in standalone mode and cluster mode still shows the advantages of Sedna as a high speed distributed file system other than a simple cache system like Memcached. The reason we do not compare Sedna with other memory based key value databases like MemBase [6] was most of these systems were based on memcached, and designed for different usages, which make it unfair to be compared together. We believe our results and other's comparing with memcached can give a clear clue on their advantages and disadvantages.

A. Sedna Write/Read Performance

We set up a cluster of 9 real servers for experiments, each server has a Xeon dual-core 2.53 GHz CPU and 6GB memory. All machines are connected with a single gigabit Ethernet link, all of them are in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond. Servers are configured to use 4GB memory as local memory storage if they are not ZooKeeper nodes, otherwise use 3GB memory. All clients run the Sedna load test programs, we use the same number of clients as servers to ensure the clients are never a bottleneck. Besides, a Memcached cluster is deployed on the same cluster to be compared with Sedna.

1) One Client Performance: In this test, there are only one client that issues read and write requests running in Sedna cluster and Memcached cluster. The size of key value pair is very small because the data size is not the most important factor which will affect Sedna efficiency. And, if the key or value was very large, some more careful strategies need to be implemented than what we have done in current Sedna implementation. To make all the test comparable, all the Key-Value pair has a 20 bytes key which was generated randomly like 'test-00000000000000', and has a 20 bytes value which was a constant value.

To show the efficiency of Sedna system, we compare Sedna with MemCached instead of other distributed file system. Some MemCached clients support a distributed way to write data, we use this features in MemCached test programs. Sedna test programs works like MemCached test programs excepts it uses Sedna strategy to manage all the data.

There are a significant difference between Sedna and Memcached when we write and read data: Sedna writes every key value pair three times into different real nodes parallel, and reads every key value pair three times from different real nodes. To make the writes in these two systems comparable, we set the Memcached test program write and read every data three times and compare the result with Sedna system as Fig. 7(a) has shown.

Though Fig. 7(a) shows Sedna has better W/R performance than Memcached, It is not fair for Memcached because three times read and write in Sedna were issued and processed parallel, but in Memcached these reads and writes requests were issued sequentially. It is still necessary to compare Sedna with Memcached when writes each data only one time. Fig. 7(b) shows the W/R performance difference when comparing Sedna and Memcached when writes each data only once. From Fig. 7, we can conclude that Sedna performance is quite stable, and slightly slower than original write-once Memcached performance, however, it is better than writing data into three different servers using Memcached clients.

2) Multi-Clients Performance: Throughput is critical for a distributed file system, we test the read and write performance of Sedna when all servers in cluster are running client applications. In this test, nine clients begin to issue the read/write requests nearly at the same time. The network bandwidth is one the most important factors affecting the read/write performance, in our test, all the servers are connected by 1Gb Ethernet.

Fig. 8 shows the performance of Sedna in nine clients and one client. We can figure that the I/O performance indeed reduce when there are more concurrent read/write clients. However, the reduction is easily understood, in multi-clients test, there are more clients issuing write operations at the same time, and each write operation will be sent to three real nodes for processing. The limitation of each server's CPU speed, network I/O bandwidth will make the individual client's speed slower, however, the overall throughput is larger than one client, as there are 9 clients reading and writing.

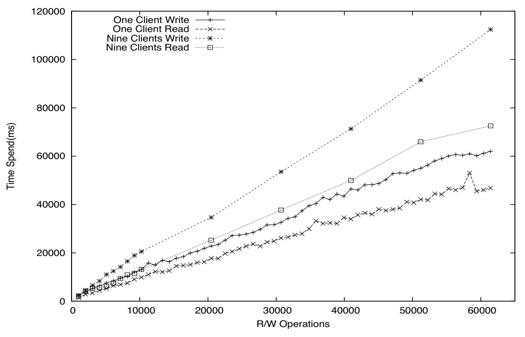


Fig. 8. R/W Speed, Nine vs One Client

As above experience has shown, writes and reads in Sedna is slightly slower than Memcached in the quiet mode. It is reasonable as there are at least 3 writes in Sedna instead of one write in Memcached. The most important result is a ZooKeeper like service will not obstruct Sedna's read and write efficiency.

VII. Related Work

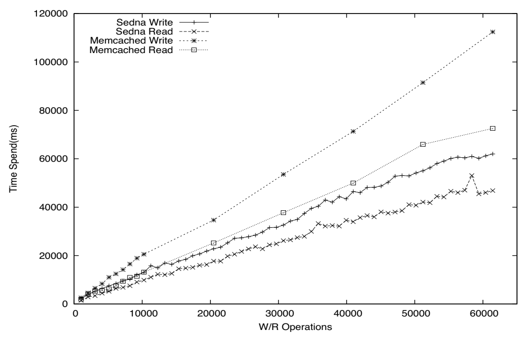
The Google File System (GFS) [15] is a distributed file system built for hosting the state of Google's internal applications, it uses a simple design with a single master server serving the entire cluster for meta data read/write, the data is split into chunks and stored in chunk servers, the GFS master is made of fault tolerant using the Chubby [13] abstraction.

Dynamo [16] is a storage system that is used by Amazon to store and retrieve user shopping carts. Dynamo's Gossip based membership algorithm helps every node maintain information about other nodes, by which Redis [9] uses to maintain cluster information too. Dynamo can be defined as a structured overlay with at most one-hop request routing. Dynamo detects updated conflicts using a vector clock scheme, but prefers a client side conflict resolution mechanism. The write operation in Dynamo also requires a read to be performed for managing the vector timestamps, this would limit the performance when systems need to handle a very high write throughput.

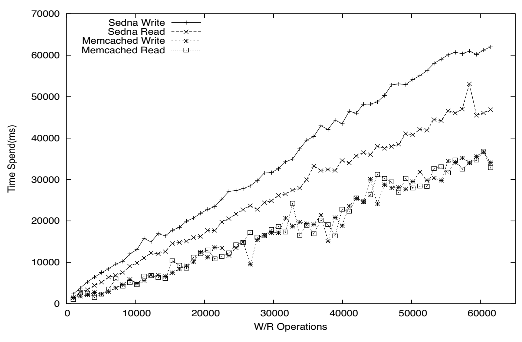
Cassandra [1] is a distributed storage system that combines Dynamo [16] and HBase [5] together, it provides data model as HBase provides, and constructs cluster in a P2P fashion as Dynamo to avoid the single point failure. Cassandra partitions the data over a set of nodes using consistent hashing as Dynamo does, however, Cassandra using different ways to ensure uniform load distribution, which was used in Chord [18] system. Cluster membership in Cassandra is based on Scuttlebutt [19], an anti-entropy Gossip based mechanism. To place replicas of one data, Cassandra system elects a leader amongst its nodes using a system called ZooKeeper [11]. All nodes contact the leader who tells them for what ranges they are replicas for. The metadata about the ranges a node is responsible is cached locally at each node and in a fault-tolerant manner inside Zookeeper.

Redis [9] and Memcached [7] are both memory based storage system, the latter should be named as a cache system instead of storage system as its simple architecture. However, Redis can be treated as a complete distributed storage system based on memory. Redis constructs cluster based on Gossip, replicates data to N copies based on master-slaves architecture, which means all writes will firstly be sent to master, and slaves will keep synchronize with their master. Redis provides data availability using write-ahead log, all the data modification will first be written on local append-only file before processing.

Sedna differs from the aforementioned storage systems in terms of the realtime processing needs. Firstly, Sedna is targeted at high random reads and random writes speed, which is also the reason we choose memory instead of disks. Secondly, Sedna is built for an infrastructure with hundreds or thousands of servers. Considering the possibility of failure in such a large scale servers, we get rid of the centralize architecture to avoid the single point failure. Thirdly, to meet the stringent speed requirements of realtime applications, we avoid routing requests through multiple nodes like Chord use, and avoid



(a) Memcached(3) vs. Sedna



(b) Memcached(1) vs. Sedna

Fig. 7. W/R Speed Comparing

Gossip mechanism to maintain a consistent cluster status like Cassandra and Redis does. Sedna uses a zero-hop DHT that each node caches enough routing information locally to route a request to the appropriate node directly, and a ZooKeeper min-cluster which keeps the newest information. Most important, Sedna is not designed and implemented as a substitution of disk-base storage systems. With the high w/r speed memory based storage ability and trigger based APIs, it provides a new way to program realtime applications on cloud platform.

VIII. Conclusion

In this paper, we introduce our work - Sedna, a key-value memory based storage system in cloud environment for realtime applications. Sedna is designed especially for huge size data centers and realtime processing. It provides many advantages comparing with other storage systems: a high speed random I/O access, a scalable structure, and new trigger based data access apis to support realtime programming. The usage of Sedna in our micro-blogging search also shows its potential in real time processing in cloud platform.

Acknowledgement

In the development of Sedna, we receive a lot of help and recommendation from our colleagues and teachers, thanks them a lot. We also want to give thanks to Liming Cui for his remarkable work on Memcached, the great discussion about Sedna persistency strategy with Feng Yang, and also YuChao Huang for his great ideas, and lots of other people.

References

[1] Apache Cassandra. Available at http://cassandra.apache.org/.
[2] Distributed hash table. http://en.wikipedia.org/wiki/Distributed-hash-table.
[3] Facebook message. www.facebook.com/about/messages.
[4] Apache hadoop. Available at http://hadoop.apache.org.
[5] Apache hbase. Available at http://hbase.apache.org.
[6] MemBase. http://www.membase.org
[7] Memcached. http://en.wikipedia.org/wiki/Memcached.
[8] Facebook realtime-analytics system. Available at http://highscalability.com/blog/2011/3/22/facebooks-new-realtime-analytics-system-hbase-to-process-20.html.
[9] Redis. http://redis.io.
[10] Scribe. https://github.com/facebook/scribe/wiki.
[11] Apache zookeeper. Available at http://zookeeper.apache.org.
[12] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.
[13] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.
[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.
[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.
[16] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's a highly available key-value store. In *In Proc. SOSP*, pages 205–220, 2007.
[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. *Data Mining Workshops, International Conference on*, 0:170–177, 2010.
[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.
[19] R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 6:1–6:7, New York, NY, USA, 2008. ACM.
[20] Leonardo Neumeyer, Bruce Robbins, Anish Nair, Anand Kesari S4: Distributed Stream Computing Platform In *IEEE International Conference on Data Mining - ICDM*, pp. 170-177, 2010.
[21] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, Russell Sears MapReduce Online In *Networked Systems Design and Implementation - NSDI*, pp. 313-328, 2010.
[22] Jeffrey Dean, Sanjay Ghemawat MapReduce: Simplied Data Processing on Large Clusters In *Operating Systems Design and Implementation - OSDI*, pp. 137-150, 2004
[23] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazires, Subhasish Mitra, Aravind Narayanan, Guru M. Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann The case for RAMClouds: scalable high-performance storage entirely in DRAM In *Operating Systems Review - SIGOPS*, vol43, no 4, pp. 92-105, 2009
[24] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica Spark: Cluster Computing with Working Set In *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* pp. 10, 2010