RC25599 (WAT1604-013) April 7, 2016
Computer Science

IBM Research Report

Infrastructure Components for Efficient Data Management

Arun Iyengar
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

Infrastructure Components for Efficient Data Management[1]

ARUN IYENGAR, IBM Research, aruni@us.ibm.com

This paper describes work that we have done in developing infrastructure components for efficient data management. A key component is the use of multiple storage options for storing data including file systems, SQL (relational) databases, NoSQL databases, and caches. We provide monitoring capabilities so users can determine the performance of different storage systems and pick the most appropriate one.

Another key aspect of this work is enhanced storage clients which are particularly applicable to cloud storage. Our work extends existing clients by adding integrated caching support, encryption, compression, and delta encoding. We have built enhanced clients for multiple storage systems including Cloudant and OpenStack Object Storage. We have also written a Java library for allowing our enhanced client features to be integrated with other storage clients and a wide range of applications.

1. INTRODUCTION

A broad range of storage systems are currently available including file systems, SQL (relational) databases, NoSQL databases, caches, etc. An increasing number of storage systems are cloud based. A key challenge is providing applications with easy access to a wide range of storage options. We have developed a universal storage manager (USM) which gives application programs access to a wide range of storage options. The USM also provides monitoring capabilities. That way, users can determine the performance of different storage systems and compare them to pick the best option.

A second aspects of this work is the development of enhanced storage clients. Storage systems typically consist of clients accessing data on one or more storage servers. The clients and servers communicate via a protocol such as HTTP. Widely used storage systems such as Cloudant (built on top of CouchDB), OpenStack Object Storage, and Cassandra have clients which are written for specific programming languages (e.g. Java, Python, JavaScript, etc). These clients handle low level details such as communications with the server using an underlying protocol such as HTTP. That way, client applications can communicate with the storage server via method (or other type of subroutine) calls in the language in which the client is written. Examples of such clients include the Cloudant Java client [Cloudant], the Java library for OpenStack Storage (JOSS) [Javaswift], and the Java Driver for Apache Cassandra[DataStax] (Figure 1).

[1] This paper extends Arun Iyengar, *Enhanced Storage Clients*, IBM Research Report RC25584 (WAT1512-042), December 2015.
http://domino.watson.ibm.com/library/CyberDig.nsf/papers/16214813202B330D85257F2A004A2187/$File/rc25584.pdf

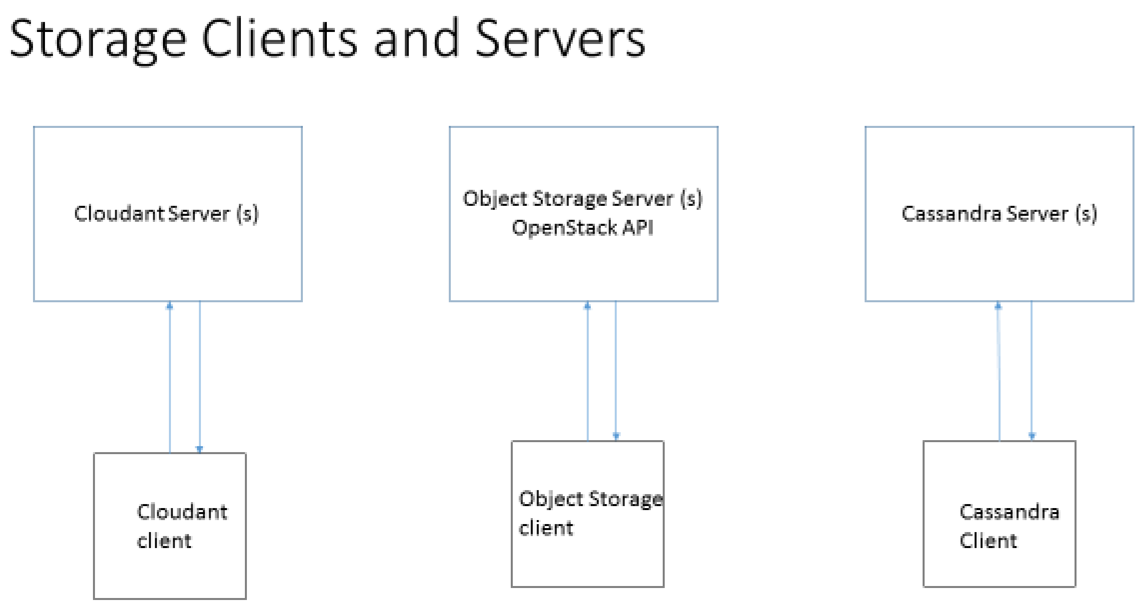

Fig. 1. Storage clients are essential for communicating with storage servers.

This paper describes work that we have done in enhancing storage clients to support integrated caching, data compression, encryption, and delta encoding. These features considerably enhance the functionality and performance of storage systems and provide key features that are needed by application programs.

The remainder of this paper is structured as follows. The first section describes our universal storage manager. The remaining sections describe our enhanced storage clients.

2. UNIVERSAL STORAGE MANAGER (USM)

The Universal Storage Manager (USM) allows application programs to access multiple storage systems including file systems, SQL (relational) databases, NoSQL databases, and caches. The current implementation has support for file systems, MySQL [MySQL], Cloudant [Cloudant], Object Storage (which implements the OpenStack Swift API) [OpenStack], Redis [Redis], and an in-process cache.

The USM provides a common key-value interface. Each storage system implements the key-value interface. That way, it is easy to switch from one storage system to another. The key-value interface exposed to application programs hides the details of how the interface is actually implemented by the underlying storage system.

In some cases, a key-value interface is not sufficient. For example, a MySQL user may need to issue SQL queries to the underlying database. The USM allows the user to access native features of the underlying storage system when needed. That way, applications can use the common key-value interface when appropriate as well as all other capabilities of the underlying storage systems when necessary.

A key feature of the USM is the ability to monitor different storage systems for performance. That way, users can measure and compare the performance of different storage systems. The USM collects both summary performance statistics such as average latency as well as detailed performance statistics such as historical latency measurements. It is often desirable to only collect latency measurements for recent requests. There is thus the capability to collect detailed data for recent requests while only retaining summary statistics for older data.

Performance data can be stored persistently using any of the storage systems supported by the USM.

Both the USM and the enhanced storage clients described in the next sections of the paper provide data encryption, compression, and serialization. The use of the key-value interface also makes it very easy to provide caching support for storage systems which are not caches. Since all of the storage systems which are caches (e.g. Redis, the in-process cache) implement the same key-value interface, similar method calls can be used to store data in the main storage system as well as in the cache.

3. ENHANCED STORAGE CLIENTS

Our enhanced storage clients are built on top of a storage client library which handles features such as caching, encryption, compression, and delta encoding. For each of these features, there is an interface and multiple possible implementations. For example, there are multiple caching implementations and multiple encryption implementations which a storage client can choose from. The storage client library (SCL) is available as a standalone entity. Storage clients for a specific storage system such as Cloudant, OpenStack Object Storage, Cassandra, etc. can be integrated with the storage client library (SCL) by adding SCL API calls at critical points in the client code. For example, when a storage client queries the server for an object, an SCL API call could be inserted to first look for the object in the cache. When a storage client updates an object at the server, an SCL API call could be inserted to update (or invalidate) the object in the cache.

If these SCL API calls are integrated with the storage client, this results in an enhanced storage client with considerable performance enhancements and added functionality over a vanilla client. An alternative approach which does not require changes to storage clients is to import the client library directly into an application. This allows applications to directly use SCL features such as caching, encryption, and compression in the most appropriate way.

4. CACHING

Caching support is critically important for improving performance [Iyengar 97]. The latency for communicating between clients and servers can be high. Caching can dramatically reduce this latency. If the cache is properly managed, it can also allow a client program to continue executing in the presence of poor or limited connectivity. Caching is most effective for data which does not change, or which is only updated rarely. Our caches have an API which controls which data are cached and allows caches to be updated. Applications can add and delete data from caches. Furthermore, cached data can have expiration times.

A key feature of our architecture is that it is modular. Our SCL includes a Cache interface which defines how an application interacts with the cache. There are multiple implementations of the Cache interface which applications can choose from (Figure 2).

There are two types of caches. In-process caches store data within the process corresponding to the application (Figure 3) [Iyengar 99]. That way, there is no interprocess communication required for storing the data. For our Java implementations of in-process caches, Java objects can directly be cached. Data serialization is not required. In order to reduce overhead when the object is cached, the object (or a reference to it) can be stored directly in the cache. This means that changes to the object from the application could affect changes to the cached object itself. In order to prevent the value of a cached object from being modified by changes to the object being made in the application, a copy of the object can be made before the object is cached. This results in overhead for copying the object.

Another approach is to use a remote process cache [Iyengar 97]. In this approach, the cache runs in one or more separate processes from the application. A remote process cache can run on a separate node from the application as well. There is some overhead for communication with a remote process cache. In addition, data often has to be serialized before being cached. However, remote process caches also have some advantages over in-process caches. A remote process cache can be shared by multiple clients, and this feature is often desirable. Remote process caches can often be scaled across multiple processes and nodes to handle high request rates and increase availability.

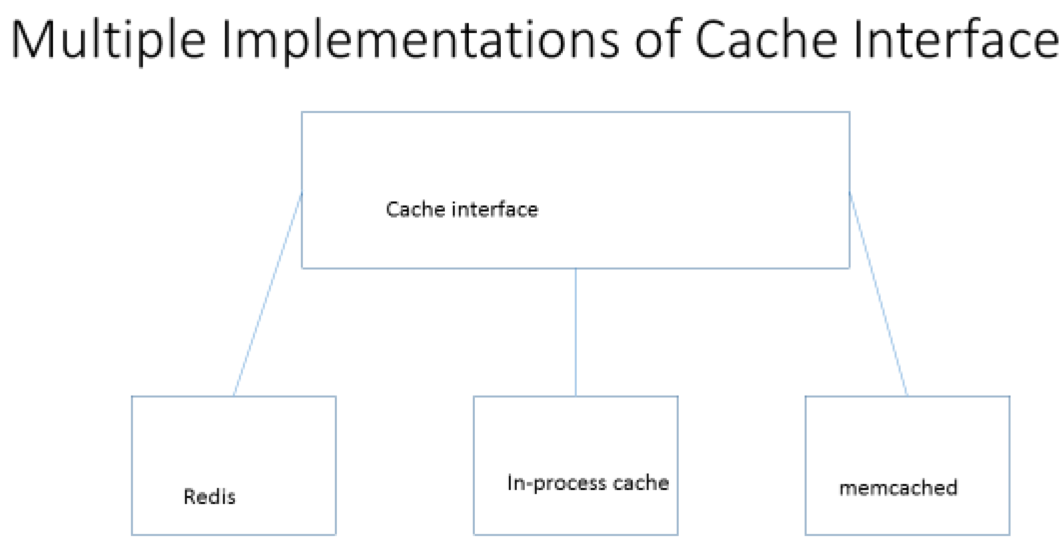

Fig. 2. Our SCL supports multiple cache implementations.

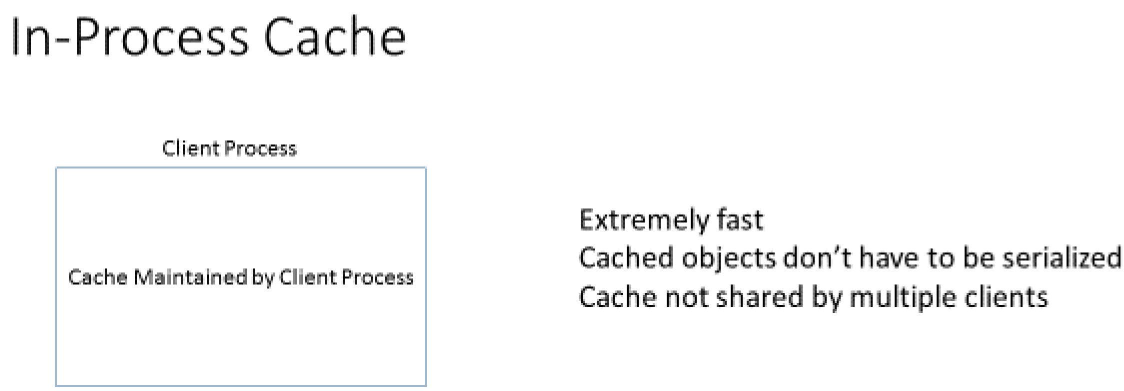

Fig. 3. In-process caches store data within the application process.

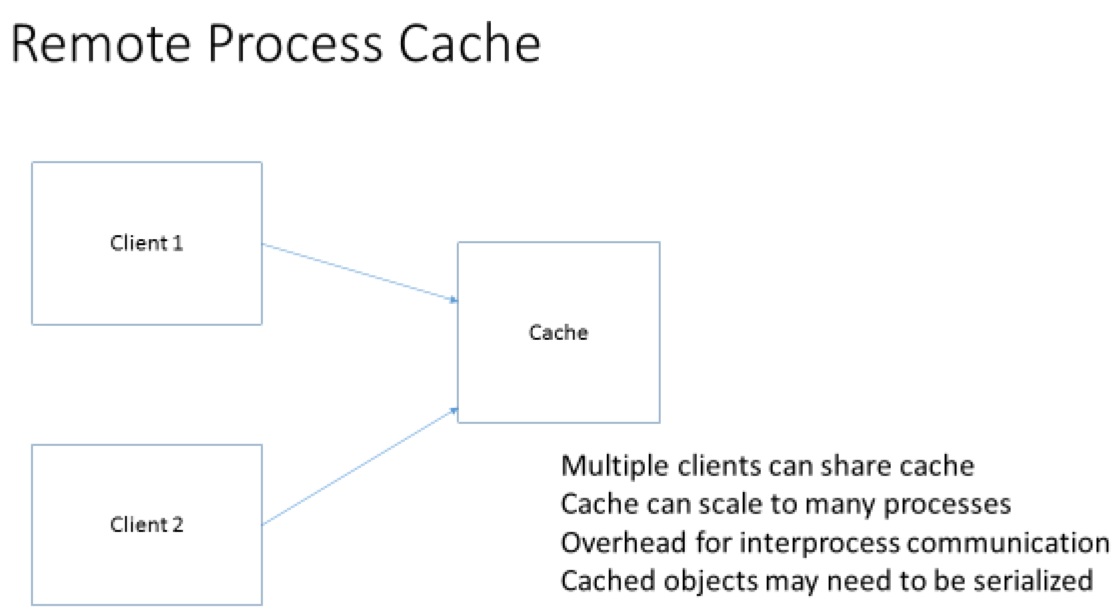

Fig. 4. Remote process caches store data outside of the application process.

There are several caches available that are available as open source solutions. Redis [Redis] and memcached [Memcached] are widely used remote process caches. They can be used for storing serialized data across a wide range of languages. Clients for redis and memcached are available across a wide range of languages (e.g. Java, C, C++, Python, Javascript, PHP, several others).

Examples of caches targeted at Java environments include the Guava cache [Decker], Ehcache [Ehcache], and OSCache [OSCache]. A common approach is to use a data structure such as a HashMap or a ConcurrentHashMap with features for thread safety and cache replacement. Since there are several good open source alternatives available, it is probably better to use an existing cache implementation instead of writing another cache implementation unless the user has specialized requirements not handled by existing caches.

Our SCL allows any of these caches to be plugged into its modular architecture. In order to use one of these caches, an implementation of the SCL Cache interface needs to be written for the cache. We have already implemented SCL Cache interfaces for redis and the Guava cache and are working on interfaces for other caches such as memcached.

The SCL allows applications to assign (optional) expiration times to cached objects. After the expiration time for an object has elapsed, the cached object is obsolete and should not be returned to an application until the server has been contacted to either provide an updated version or verify that the expired object is still valid. Cache expiration times are managed by the SCL and not by the underlying cache. There are a couple of reasons for this. Not all caches support expiration times. A cache which does not handle expiration times can still implement the SCL Cache interface. In addition, for caches which support expiration times, objects whose expiration times have elapsed might be purged from the cache. We do not always want this to happen. After the expiration time for a cached object has elapsed, it does not necessarily mean that the object is obsolete. Therefore, the SCL has the ability to keep around a cached object o1 whose expiration time has elapsed. If o1 is requested after its expiration time has passed, then the client might have the ability to revalidate o1 in a manner similar to an HTTP GET request with an If-Modified-Since header. The basic idea is that the client sends a request to fetch o1 only if its version of o1 has changed. In order to determine if its version of o1 is obsolete, the client could send a timestamp, entity tag, or other information identifying the version of o1 stored at the client. If the server determines that the client has an obsolete version of o1, then the server will send a new version of o1 to the client. If the server determines that the client has a current version of o1, then the server will indicate that the version of o1 is current (Figure 5).

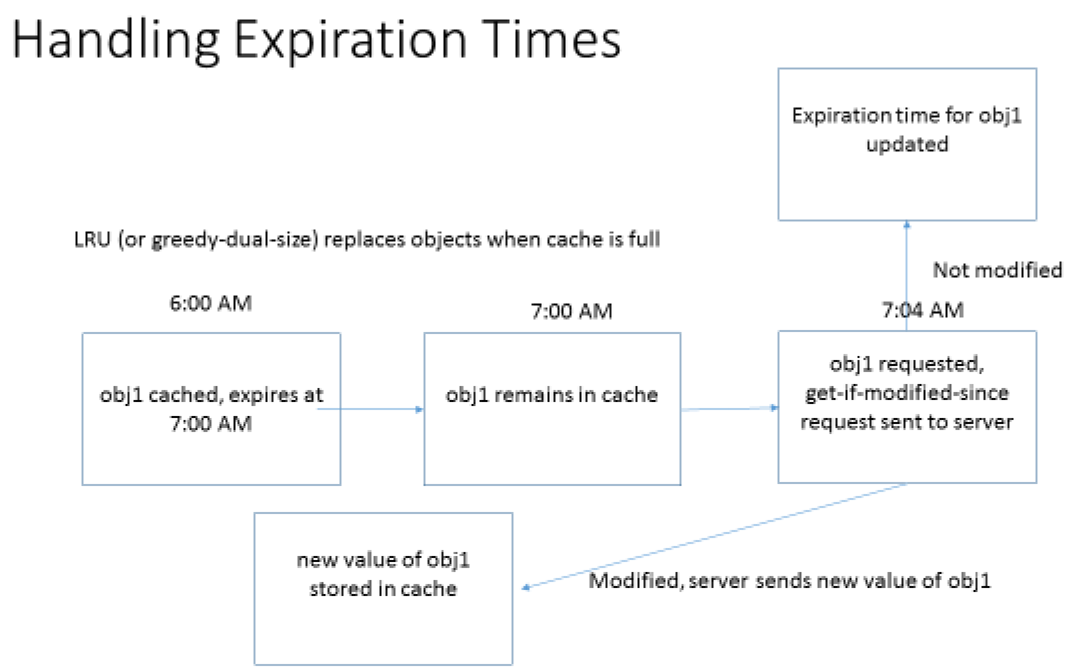

Fig. 5. Revalidation with the server after expiration times have expired can avoid fetching cached objects which have not changed, even if their expiration times have passed.

Using this approach, the client does not have to receive identical copies of objects whose expiration times have elapsed even though they are still current. This can save considerable bandwidth and improve performance. There is still latency for revalidating o1 with the server, however.

If caches become full, a cache replacement algorithm such as least recently used (LRU) or greedy-dual-size [Cao] can be used to determine which objects to retain in the cache.

Caching can be used for situations in which there is poor connectivity between the client and server. In some cases, an application can be written so that it can execute for considerable periods of time using locally cached data with sporadic batch updates with the server.

Some caches such as redis have the ability to back up data in persistent storage (e.g. to a hard disk or solid-state disk). This allows data to be preserved in the event that a cache fails. It is also often desirable to store some data from a cache persistently before shutting down a cache process. That way, when the cache is restarted, it can quickly be brought to a warm state by reading in the data previously stored persistently.

Data can be replicated across multiple caches as another alternative to preserving cached data in the event of a cache failure. This approach can slow be used to scale caches to handle high request rates. Multiple data copies introduce the problem of data inconsistencies. Caching generally works best for data which does not change very frequently.

5. ENCRYPTION AND DATA CONFIDENTIALITY

Data confidentiality is critically important. Many storage systems encrypt data within the server. However, this may not be sufficient. Users of a storage system might prefer to encrypt data before it reaches the server since the server might not be completely trustworthy. It may also be undesirable to store confidential information in a cache as the contents of the cache could be read by a malicious third party. Both the data in a cached object as well as the key used to index the object in a cache could contain confidential information.

Our SCL can encrypt data both before sending data to the server as well as before caching the data. The architecture supports different encryption algorithms. We currently are using an implementation of the Advanced Encryption Standard (AES) [NIST].

There are situations in which data anonymization is needed but not strong encryption. In this case, a data anonymization implementation of the encryption module can be used. The data anonymizer might preserve anonymity while still allowing some types of computations over the anonymized data. Since decryption would not be needed for analyzing the data, performance might be improved. In addition, privacy would be better preserved since the actual data would not be fully exposed, even during data analysis.

6. COMPRESSION AND DELTA ENCODING

Compression can considerably reduce the space consumed by data objects. Even if the storage server provides compression, compressing data at the client may be desirable to reduce data transfer sizes between the client and server. Furthermore, it may be desirable to compress objects before caching them in order to conserve cache space. The SCL has the capability to compress data using gzip [Gzip]. Other compression algorithms can also be used.

Data transfer sizes between the client and server can further be reduced by delta encoding. The key idea is that when the client updates an object o1, it may not have to send the entire updated copy of o1 to the server. Instead, it sends a delta between o1 and the previous version of o1 stored at the server. This delta might only be a fraction of the size of o1 [Douglis].

A simple example of delta encoding is shown in Figure 6. Only two elements of the array in the figure change. Instead of sending a copy of the entire updated array, only the delta shown at the bottom is sent. The first element of the delta indicates that the 5 array elements beginning with index 0 are unchanged. The next element of the delta contains updated values for the next two consecutive elements of the array. The last element of the delta indicates that the 6 array elements beginning with index 7 are unchanged.

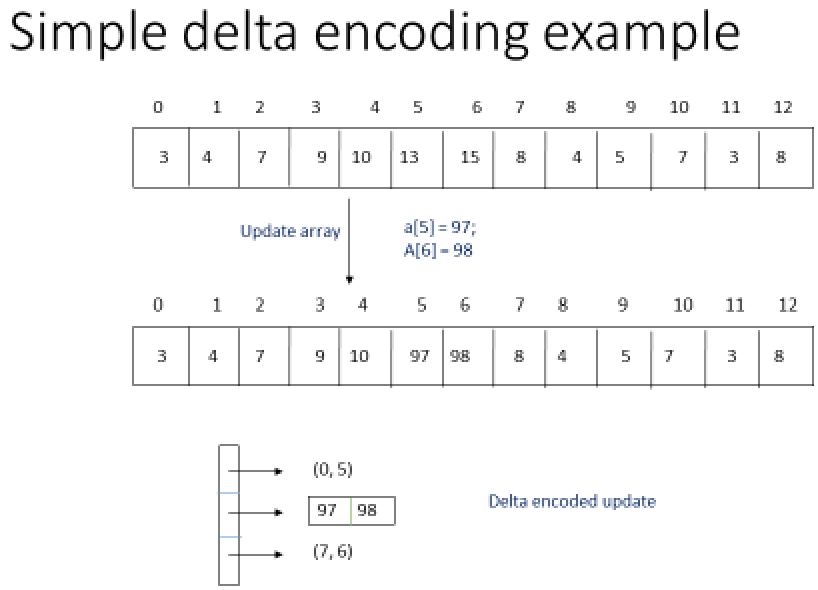

Fig. 6. Example of delta encoding.

Our delta encoding algorithm uses key ideas from the Rabin-Karp string matching algorithm [Karp]. Data objects are serialized to byte arrays. Byte arrays can be compressed by finding substrings previously encountered. If the server has a previous substring, the client can send bytes corresponding to the substring by sending an index corresponding to the position of the substring and the length of the substring as illustrated in Figure 6. Matching substrings should have a minimum length, WINDOW_SIZE (e.g. 5). If the algorithm tries to encode differences by locating very short substrings (e.g. of length 2), the space overhead for encoding the delta may be higher than simply sending the bytes unencoded. When a matching substring of length at least WINDOW_SIZE is found, it is expanded to the maximum possible size before being encoded as a delta.

As we mentioned, a cached object o can be serialized to a byte array, b. We find matching substrings of o by hashing all subarrays of b of length WINDOW_SIZE in a hash table. In order to hash substrings of o efficiently, we use a rolling hash function which can be efficiently computed using a sliding window moving along b. That way, the hash value for the substring beginning at b[i + 1] is efficiently calculated from the hash value for the substring beginning at b[i].

7. IMPLEMENTATION

We have implemented enhanced storage clients for multiple storage systems including Cloudant and OpenStack Object Storage. A library (SCL) implementing several of the features discussed previously is available as open source software on github [Storage-client-library]. The github repository supports caching, encryption, and data compression. We have not yet added delta encoding to the public repository but may do so in the future. This library (SCL) can be used for enhancing storage clients as well as for a wide variety of other applications when caching, encryption, and/or compression are needed.

We now give some examples of how our SCL can be used. One can create an in-process cache of integers indexed by strings using the Guava cache with a maximum cache size of "numObjects" and a default lifetime of "defaultLifetime" in milliseconds via:

```
InProcessCache<String, Integer> cache = new InProcessCache<String, Integer>(
        numObjects, defaultLifetime);
```

One can connect to a redis cache of byte arrays indexed by strings running on the same node from a Java program via:

```
RedisCache<String, byte[]> cache = new RedisCache<String, byte[]>
    ("localhost", 6379, 60, defaultExpiration);
```

Here, 6379 is the port number, and 60 indicates that idle connections should be closed after 60 seconds.

The following method call adds 42 to the cache indexed by "key1". "lifetime" is the lifetime of the cached value in milliseconds:

```
cache.put(key1, 42, lifetime);
```

The following method call adds all key-value pairs corresponding to "map" to the cache and assigns each key-value pair a lifetime of "lifetime" milliseconds:

```
cache.putAll(map, lifetime);
```

The following method call returns the value corresponding to "key2". It returns null if "key2" is not found in the cache, or if the value is expired.

```
val = cache.get(key2);
```

In the following method call, "list" is a list of keys. getAll looks up all key-value pairs corresponding to keys in "list" and returns a map containing them. In this example, cache keys are strings, and values are integers.

```
Map<String, Integer> map = cache.getAll(list);
```

The following method call deletes the key-value pair corresponding to "key2" from the cache if present:

```
cache.delete(key2);
```

In the following method call, "list" is a list of keys. deleteAll deletes all key-value pairs corresponding to a key in "list":

```
cache.deleteAll(list);
```

The following method call deletes all objects in the cache:
```
cache.clear();
```

The following method call outputs information about a cache entry:
```
cache.printCacheEntry(key1);
```

The size method returns the number of cached objects:
```
System.out.println("Cache size: " + cache.size());
```

The following displays the contents of the entire cache. It should not be invoked if the cache contains a large amount of data as the data outputted would be prohibitively large:
```
System.out.println(cache.toString());
```

The following displays cache statistics, such as hit rates:
```
System.out.println(cache.getStatistics().getStats());
```

The following method call generates an encryption key for encrypting data:
```
Encryption.Key secretKey = Encryption.generateKey();
```

The following method call encrypts “hm” using encryption key “secretKey”:
```
SealedObject so = Encryption.encrypt(hm, secretKey);
```

The following method call decrypts “so” using encryption key “secretKey”:
```
HashMap<String, Integer> hm2 = Encryption.decrypt(so, secretKey);
```

The following method call compresses “hm”:
```
byte[] compressed = Util.compress(hm);
```

The following method call decompresses “compressed”:
```
HashMap<String, Integer> hm2 = Util.decompress(compressed);
```

Enhanced storage clients are written by adding SCL API calls at critical points in the source code. That way, users of the enhanced storage clients can get the benefits of caching without having to explicitly add and delete objects from the cache. Our enhanced storage clients handle caching transparently to users.

In some cases, users may want explicit control over the cache contents. This may be important in situations in which cached objects are changing and there is no way for the storage client to make the correct caching decisions to achieve both optimal performance and cache consistency. In these situations, the user can directly use SCL API calls to explicitly control the contents of caches.

The decision to compress or encrypt data often needs to be made at the application level. Therefore, users will often directly use SCL API calls to encrypt or compress specific objects without relying on storage clients to do so. It is also possible for storage clients to define encryption and compression policies wherein certain objects by default will be encrypted and/or compressed before they are sent to the client. Similar policies can be defined to automatically encrypt and/or compress certain objects by default before caching them.

The performance improvements we have achieved with caching can be significant. For data stored remotely in the cloud, we have seen latency improvements of multiple orders of magnitude when caching is used.

8. PERFORMANCE TESTING

Since storing and retrieving data can have significant overhead, it is important to understand the performance of data storage systems. The USM allows storage operations to be monitored so that users can understand the overhead of their data storage systems as well as compare the overhead of different storage systems. We also provide automatic performance testing features which allow users to quickly and easily test the performance of different data stores under different workloads.

In order to test and compare performance of different data stores, we provide a workload generator which generates storage requests and determines the time for satisfying the requests. The workload generator can synthetically generate data objects to be stored. Alternatively, users can provide their own data objects for performance tests either by placing the data in input files or writing a user-defined method to provide the data.

Our automated performance tester determines performance both with and without caching using either an in-process or remote process cache. Performance is determined for a variety of different hit rates, where the granularity of hit rates is specified by users.

Performance for encryption, decryption, compression, and decompression is also provided.

Data from performance testing is stored in text files which can be easily imported into graph plotting tools such as gnuplot, spreadsheests such as Microsoft Excel, and data analysis tools such as MATLAB.

A key aspect of our performance tester is the ease of generating performance data. Any client using our libraries can easily generate performance data across all data stores supported by our libraries.

ACKNOWLEDGMENTS

The author would like to thank Rich Ellis and Mike Rhodes for their assistance and support in developing caching support for Cloudant.

REFERENCES

Dulcardo Arteaga, Ming Zhao, "Client-side Flash Caching for Cloud Systems", Client-side Flash Caching for Cloud Systems", Proceedings of ACM SYSTOR 2014.

Pei Cao, Sandy Irani, "Cost-Aware WWW Proxy Caching Algorithms", Proceedings of USITS '97.

Feng Chen, Michael Mesnier, Scott Hahn, "Client-aware Cloud Storage". Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST 2014).

Cloudant, Cloudant Java Client, https://github.com/cloudant/java-cloudant

DataStax. Java Driver for Apache Cassandra, http://docs.datastax.com/en/developer/java-driver/2.0/java-driver/whatsNew2.html

Colin Decker, "CachesExplained", https://github.com/google/guava/wiki/CachesExplained

Fred Douglis, Arun Iyengar, “Application-specific Delta-encoding via Resemblance Detection”, Proceedings of the USENIX 2003 Annual Technical Conference.

Ehcache, http://www.ehcache.org/

Gzip, “The gzip home page”, http://www.gzip.org/"

David A. Holland, Elaine Angelino, Gideon Wald, Margo I. Seltzer, “Flash Caching on the Storage Client”, Proceedings of the 2013 USENIX Annual Technical Conference.

Arun Iyengar and Jim Challenger, “Improving Web Server Performance by Caching Dynamic Data", In Proceedings of the USENIX 1997 Symposium on Internet Technologies and Systems (USITS '97), Monterey, CA, December 1997.

Arun Iyengar, "Design and Performance of a General-Purpose Software Cache". In Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference (IPCCC'99), Phoenix/Scottsdale, Arizona, February 1999.

Javaswift, JOSS: Java library for OpenStack Storage, aka Swift, http://joss.javaswift.org/

Richard M. Karp, Michael O. Rabin, “Efficient randomized pattern-matching algorithms”, IBM Journal of Research and Development, Vol. 31 no. 2, March 1987, pp. 249-260.

Memcached, http://memcached.org/

MySQL, https://www.mysql.com/

NIST, “Announcing the Advanced Encryption Standard (AES)”, Federal Information Processing Standards Publication 197, November 26, 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

OpenStack, “Welcome to Swift’s documentation!”, http://docs.openstack.org/developer/swift/

OSCache, https://java.net/projects/oscache

Redis, http://redis.io/

Storage-client-library, https://github.com/aruniyengar/storage-client-library