# Providing Enhanced Functionality for Data Store Clients

Arun Iyengar
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
Email: aruni@us.ibm.com

***Abstract*—Data stores are typically accessed using a client-server paradigm wherein the client runs as part of an application process which is trying to access the data store. This paper presents the design and implementation of enhanced data store clients having the capability of caching data for reducing the latency for data accesses, encryption for providing confidentiality before sending data to the server, and compression for reducing the size of data sent to the server. We support multiple approaches for caching data as well as multiple different types of caches.**

**We also present a Universal Data Store Manager (UDSM) which allows an application to access multiple different data stores and provides a common interface to each data store. The UDSM provides both synchronous and asynchronous interfaces to each data store that it supports. An asynchronous interface allows an application program to access a data store and continue execution before receiving a response from the data store. The UDSM also can monitor the performance of different data stores. A workload generator allows users to easily determine and compare the performance of different data stores.**

**The paper examines the key design issues in developing both the enhanced data store clients and the UDSM. It also looks at important issues in implementing client-side caching. The enhanced data store clients and UDSM are used to determine the performance of different data stores and to quantify the performance gains that can be achieved via caching.**

## I. Introduction

A broad range of data stores are currently available including SQL (relational) databases, NoSQL databases, caches, and file systems. An increasing number of data stores are available on the cloud and through open source software. There clearly is a need for software which can easily provide access to multiple data stores as well as compare their performance. One of the goals of this work is to address this need.

A second goal of this work is to improve data store performance. Latencies for accessing data stores are often high. Poor data store performance can present a critical bottleneck for users. For cloud-based data stores where the data is stored at a location which is distant from the application, the added latency for communications between the data store server and the applications further increases data store latencies [1], [2]. Techniques for improving data store performance such as caching are thus highly desirable. A related issue is that there are benefits to keeping data sizes small; compression can be a key component for achieving this.

A third goal of this work is to provide data confidentiality as it is critically important to many users and applications. Giving users the means to encrypt data may be essential either because a data store lacks encryption features or data is not securely transmitted between the application and data store server. For certain applications, encryption at the client is a necessity as the data store provider simply cannot be trusted to be secure. There have been many serious data breaches in recent years in which confidential data from hundreds of millions of people have been stolen. Some of the widely publicized data breaches have occurred at the US government, Yahoo!, Anthem, Democratic National Committee, eBay, Home Depot, JP Morgan Chase, Sony, and Target.

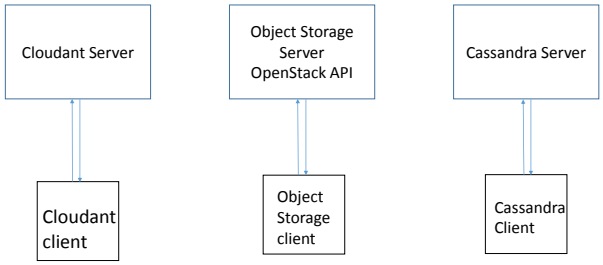


Fig. 1: Data stores are typically accessed using clients. This work focuses on enhancing the functionality of clients.

Data stores are often implemented using a client-server paradigm in which a client associated with an application program communicates with one or more servers using a protocol such as HTTP (Figure 1). Clients provide interfaces for application programs to access servers. This paper focuses on providing multiple data store options, improving performance, and providing data confidentiality by enhancing data store clients. No changes are required to servers. That way, our techniques can be used by a broad range of data stores. Requiring changes to the server would entail significantly higher implementation costs and would seriously limit the number of data stores our techniques could be applied to.

We present architectures and implementations of data store clients which provide enhanced functionality such as caching, encryption, compression, asynchronous (nonblocking) interfaces, and performance monitoring. We also present a universal data store manager (UDSM) which gives application programs access to a broad range of data store options along with the enhanced functionality for each data store. Caching, encryption, compression, and asynchronous (nonblocking) interfaces are essential; users would benefit considerably if they become standard features of data store clients. Unfortunately, that is not the case today. Research in data stores often focuses on

server features with inadequate attention being paid to client features.

The use of caching at the client for reducing latency is particularly important when data stores are remote from the applications accessing them. This is often the case when data is being stored in the cloud. The network latency for accessing data at geographically distant locations can be substantial [3]. Client caching can dramatically reduce latency in these situations. With the proliferation of cloud data stores that is now taking place, caching becomes increasingly important for improving performance.

Client encryption capabilities are valuable for a number of reasons. The server might not have the ability to encrypt data. Even if the server has the ability to encrypt data, the user might not trust the server to properly protect its data. There could be malicious parties with the ability to breach the security of the servers and steal information.

Another reason for using client-side encryption is to preserve confidentiality of data exchanged between the client and server. Ideally, the client and server should be communicating via a secure channel which encrypts all data passed between the client and server. Unfortunately, this will not always be the case, and some servers and clients will communicate over unencrypted channels.

Compression can reduce the memory consumed within a data store. Client-based compression is important since not all servers support compression. Even if servers have efficient compression capabilities, client-side compression can still improve performance by reducing the number of bytes that need to be transmitted between the client and server. In cloud environments, a data service might charge based on the size of objects sent to the server. Compressing data at the client before sending the data to the server can save clients money in this type of situation.

Our enhanced data store clients and UDSM are architected in a modular way which allows a wide variety of data stores, caches, encryption algorithms, and compression algorithms. Widely used data stores such as Cloudant (built on top of CouchDB), OpenStack Object Storage, and Cassandra have existing clients implemented in commonly used programming languages. Popular language choices for data store clients are Java, Python, and Node.js (which is actually a JavaScript runtime built on Chrome's V8 JavaScript engine). These clients handle low level details such as communication with the server using an underlying protocol such as HTTP. That way, client applications can communicate with the data store 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 [4], the Java library for OpenStack Storage (JOSS) [5], and the Java Driver for Apache Cassandra [6] (Figure 1).

The UDSM is built on top of existing data store clients. That way, we do not have to re-implement features which are already present in an existing client. The UDSM allows multiple clients for the same data store if this is desired.

It should be noted that client software for data stores is constantly evolving, and new clients frequently appear. The UDSM is designed to allow new clients for the same data store to replace older ones as the clients evolve over time.

A key feature of the UDSM is a common key-value interface which is implemented for each data store supported by the UDSM. If the UDSM is used, the application program will have access to both the common key-value interface for each data store as well as customized features of that data store that go beyond the key-value interface, such as SQL queries for a relational database. If an application uses the key-value interface, it can use any data store supported by the UDSM since all data stores implement the interface. Different data stores can be substituted for the key-value interface as needed.

The UDSM provides a synchronous (blocking) interface to data stores for which an application will block while waiting for a response to a data store request. It also provides an asynchronous (nonblocking) interface to data stores wherein an application program can make a request to a data store and not wait for the request to return a response before continuing execution. The asynchronous interface is important for applications which do not need to wait for all data store operations to complete before continuing execution and can often considerably reduce the completion time for such applications.

Most existing data store clients only provide a synchronous interface and do not offer asynchronous operations on the data store. A key advantage to our UDSM is that it provides an asynchronous interface to all data stores it supports, even if a data store does not provide a client with asynchronous operations on the data store.

The UDSM also provides monitoring capabilities as well as a workload generator which allows users to easily determine the performance of different data stores and compare them to pick the best option.

While caching can significantly improve performance, the optimal way to implement caching is not straightforward. There are multiple types of caches currently available with different performance trade-offs and features [7], [8], [9], [10], [11]. Our enhanced clients can make use of multiple caches to offer the best performance and functionality. We are not tied to a specific cache implementation. As we describe in Section III, it is important to have implementations of both an in-process cache as well as a remote process cache like Redis [7] or memcached [8] as the two approaches are applicable to different scenarios and have different performance characteristics. We provide key additional features on top of the base caches such as expiration time management and the ability to encrypt or compress data before caching it.

The way in which a cache such as Redis is integrated with a data store to properly perform caching is key to achieving an effective caching solution. We have developed three types of integrated caching solutions on top of data stores. They vary in how closely the caching API calls are integrated with the data store client code. We discuss this further in Section III.

Our paper makes the following key contributions:

- We present the design and implementation of enhanced data store clients which improve performance and security by providing integrated caching, encryption, and compression. We have written a library for

implementing enhanced data store clients and made it available as open source software [12]. This is the first paper to present caching, encryption, and compression as being essential features for data store clients and to describe in detail how to implement these features in data store clients in a general way. Our enhanced data store clients are being used by IBM customers.

- We present the design and implementation of a universal data store manager (UDSM) which allows application programs to access multiple data stores. The UDSM allows data stores to be accessed with a common key-value interface or with an interface specific to a certain data store. The UDSM provides the ability to monitor the performance of different data stores as well as a workload generator which can be used to easily compare the performance of data stores. The UDSM also has features provided by our enhanced clients so that caching, encryption, and compression can be used to enhance performance and security for all of the data stores supported by the UDSM. The UDSM provides both a synchronous (blocking) and asynchronous (nonblocking) interface to all data stores it supports, even if a data store fails to provide a client supporting asynchronous operations on the data store. Asynchronous interfaces are another commonly ignored feature which should become a standard feature of data store clients. The UDSM is available as open source software [13] and is being used by IBM customers. We are not aware of any other software which provides the full functionality of our UDSM or is designed the same way.
- We present key issues that arise with client-side caching and offer multiple ways of implementing and integrating caches with data store clients.
- We present performance results from using our enhanced clients and UDSM for accessing multiple data stores. The results show the high latency that can occur with cloud-based data stores and the considerable latency reduction that can be achieved with caching. The latency from remote process caching can be a problem and is significantly higher than latency resulting from in-process caching.

The remainder of this paper is structured as follows. Section II presents the overall design of our enhanced data store clients and UDSM. Section III presents some of the key issues with effectively implementing client-side caching. Section IV discusses how delta encoding can sometimes be used to reduce data transfer sizes when a new version of an object is stored. Section V presents a performance evaluation of several aspects of our system. Section VI presents related work. Finally, Section VII concludes the paper.

## II. Design and Implementation of Enhanced Data Store Clients

Key goals of this work are to improve performance, provide data confidentiality, reduce data sizes where appropriate, provide access to multiple data stores, and to monitor and compare performance of different data stores. Caching, encryption, and compression are core capabilities of our enhanced data store clients. These features are quite useful for a wide variety of applications; software developers should make much more of an effort to incorporate them within data store clients.

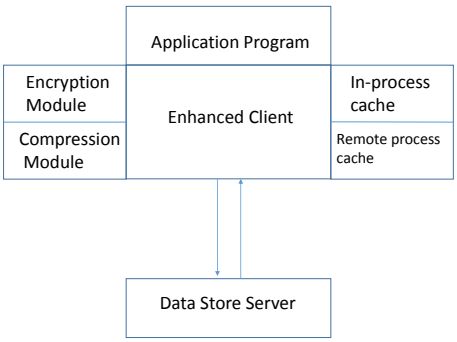


Fig. 2: Enhanced data store client.

Our enhanced data store clients are built on top of a data store client library (DSCL) which handles features such as caching, encryption, compression, and delta encoding (Figure 2). For important features, there is an interface and multiple possible implementations. For example, there are multiple caching implementations which a data store client can choose from. A Java implementation of the DSCL is available as open source software [12]. A guide for using this DSCL is also available [14].

Commonly used data stores such as Cloudant, OpenStack Object Storage, Cassandra, etc. have clients in widely used programming languages which are readily available as open source software [4], [5], [6]. These clients make it easy for application programs to access data stores since they can use convenient function/method calls of a programming language with properly typed arguments instead of dealing with low level details of a protocol for communicating with a data store.

While existing clients for data stores handle the standard operations defined on that data store, they generally do not provide enhanced features such as caching, encryption, compression, performance monitoring, and a workload generator to test the performance of a data store. Our DSCL and UDSM provide these enhanced features and are designed to work with a wide variety of existing data store clients.

Our DSCL can have various degrees of integration with an existing data store client. In a tight integration, the data store client is modified to make DSCL calls at important places in the client code. For example, DSCL calls could be inserted to first look for an object in a cache when a data store client queries a server for an object. DSCL API calls could also be inserted to update (or invalidate) an object in a cache when a data store client updates an object at a server. More complicated series of API calls to the DSCL could be made to achieve different levels of cache consistency. A similar approach can be used to enable a data store client to perform encryption, decryption, compression, and/or decompression transparently to an application.

Tight integration of a data store client with the DSCL requires source code modifications to the data store client. While this is something that software developers of the data

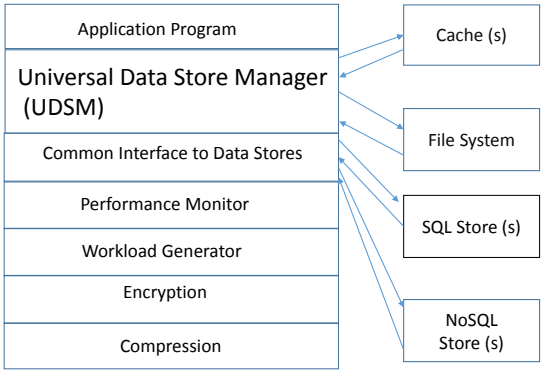


Fig. 3: Universal data store manager.

store client should be able to, it is not something that a typical user of the client can be expected to do. However, tight integration is not required to use the DSCL with a data store client. The DSCL has explicit API calls to allow caching, encryption, and compression. Users can thus use the DSCL within an application independently of any data store. The advantage to a tight integration with a data store is that the user does not have to make explicit calls to the DSCL for enhanced features such as caching, encryption, and compression; the data store client handles these operations automatically. In a loosely coupled implementation in which the data store client is not modified with calls to the DSCL, the user has to make explicit DSCL calls to make use of enhanced features.

Optimal use of the DSCL is achieved with a tight coupling of the DSCL with data store clients along with exposing the DSCL API to give the application fine grained control over enhanced features. In some cases, it may be convenient for an application to make API calls to client methods which transparently make DSCL calls for enhanced features. In other cases, the application program may need to explicitly make DSCL calls to have precise control over enhanced features.

### A. *Universal Data Store Manager*

Existing data store clients typically only work for a single data store. This limitation is a key reason for developing our Universal Data Store Manager (UDSM) which allows application programs to access multiple data stores including file systems, SQL (relational) databases, NoSQL databases, and caches (Figure 3).

The UDSM provides a common key-value interface. Each data store implements the key-value interface. That way, it is easy for an application to switch from using one data store to another. The key-value interface exposed to application programs hides the details of how the interface is actually implemented by the underlying data store.

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 UDSM allows the user to access native features of the underlying data store when needed. That way, applications can use the common key-value interface when appropriate as well as all other capabilities of the underlying data store when necessary.

A key feature of the UDSM is the ability to monitor different data stores for performance. Users can measure and compare the performance of different data stores. The UDSM collects both summary performance statistics such as average latency as well as detailed performance statistics such as past latency measurements taken over a period of time. 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 data stores supported by the UDSM.

The UDSM also provides a workload generator which can be used to generate requests to data stores in order to determine performance. The workload generator allows users to specify the data to be stored and retrieved. The workload generator automatically generates requests over a range of different request sizes specified by the user. 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. The workload generator also determines read latencies when caching is being used for different hit rates specified by the user. Additionally, the workload generator also measures the overhead of encryption and compression.

The workload generator is ideal for easily comparing the performance of different data stores across a wide range of data sizes and cache hit rates. Performance will vary depending on the client, and the workload generator can easily run on any UDSM client. The workload generator was a critical component in generating the performance data in Section V. Data from performance testing is stored in text files which can be easily imported into graph plotting tools such as gnuplot, spreadsheets such as Microsoft Excel, and data analysis tools such as MATLAB.

A Java implementation of the UDSM is available as open source software [13]. A guide for using this UDSM is also available [15]. This UDSM includes existing Java clients for data stores such as the Cloudant Java client [4] and the Java library for OpenStack Storage (JOSS) [5]. Other data store clients, such as the Java Driver for Apache Cassandra [6], could also be added to the UDSM. That way, applications have access to multiple data stores via the APIs in the existing clients. It is necessary to implement the UDSM key-value interface for each data store; this is done using the APIs provided by the existing data store clients. The UDSM allows SQL (relational) databases to be accessed via JDBC. The key-value interface for SQL databases can also be implemented using JDBC.

Most interfaces to data stores are synchronous (blocking). An application will access a data store via a method or function call and wait for the method or function to return before continuing execution. Performance can often be improved via asynchronous (nonblocking) interfaces wherein an application can access a data store (to store a data value, for example) and continue execution before the call to the data store interface returns. The UDSM offers both synchronous and asynchronous interfaces to data stores.

The asynchronous interface allows applications to continue

executing after a call to a method to a data store API method by using a separate thread for the data store API method. Since creating a new thread is expensive, the UDSM uses thread pools in which a given number of threads are started up when the UDSM is initiated and maintained throughout the lifetime of the UDSM. A data store API method invoked via an asynchronous interface is assigned to one of the existing threads in the thread pool which avoids the costly creation of new threads to handle asynchronous API method calls. Users can specify the thread pool size via a configuration parameter.

The Java UDSM implementation implements asynchronous calls to data store API methods using the ListenableFuture interface [16]. Java provides a Future interface which can be used to represent the result of an asynchronous computation. The Future interface provides methods to check if the computation corresponding to a future is complete, to wait for the computation to complete if it has not finished executing, and to retrieve the result of the computation after it has finished executing. The ListenableFuture extends the Future interface by giving users the ability to register callbacks which are code to be executed after the future completes execution. This feature is the key reason that we use ListenableFutures instead of only Futures for implementing asynchronous interfaces.

The common key-value interface serves a number of useful purposes. It hides the implementation details and allows multiple implementations of the key-value interface. This is important since different implementations may be appropriate for different scenarios. In some cases, it may be desirable to have a cloud-based key-value store. In other cases, it may be desirable to have a key-value store implemented by a local file system. In yet other cases, it may be desirable to have a cache implementation of the key-value interface such as Redis or memcached. Since the application accesses all implementations using the same key-value interface, it is easy to substitute different key-value store implementations within an application as needed without changing the source code.

Another advantage of the key-value interface is that important features such as asynchronous interfaces, performance monitoring, and workload generation can be performed on the key-value interface itself, automatically providing the feature for all data stores implementing the key-value interface. Once a data store implements the key-value interface, no additional work is required to automatically get an asynchronous interface, performance monitoring, or workload generation for the data store (unless it is necessary to implement one of these features for a type of data store access not provided by the key-value interface, such as an SQL query for a relational database). In our Java implementation of the UDSM, applying important features to all data store implementations in the same code is achieved by defining a

```
public interface KeyValue<K,V> {
```

which each data store implements. The code which implements asynchronous interfaces, performance monitoring, and workload generation takes arguments of type `KeyValue<K,V>` rather than an implementation of `KeyValue<K,V>`. That way, the same code can be applied to each implementation of `KeyValue<K,V>`.

The UDSM provides data encryption and compression in a similar fashion as the DSCL. The DSCL can be used to provide integrated caching for any of the data stores supported by the UDSM. In addition, the key-value interface allows UDSM users to manually implement caching without using the DSCL. The key point is that via the key-value interface, any data store can serve as a cache or secondary repository for one of the other data stores functioning as the main data store. The user would make appropriate method calls via the key-value interface to maintain the contents of a data store functioning as a cache or secondary repository. The next section explores caching in more detail.

## III. Caching

Caching support is critically important for improving performance [17]. 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 an application to continue executing in the presence of poor or limited connectivity with a data store server.

Our enhanced data store clients use three types of caching approaches. In the first approach, caching is closely integrated with a particular data store. Method calls to retrieve data from the data store can first check if the data are cached, and if so, return the data from the cache instead of the data store. Methods to store data in the data store can also update the cache. Techniques for keeping caches updated and consistent with the data store can additionally be implemented. This first caching approach is achieved by modifying the source code of a data store client to read, write, and maintain the cache as appropriate by making appropriate method calls to the DSCL.

We have used this first approach for implementing caching for Cloudant. The source code for this implementation is available from [18]. We have also used this approach for implementing caching for OpenStack Object Storage by enhancing the Java library for OpenStack Storage (JOSS) [5]. While this approach makes things easier for users by adding caching capabilities directly to data store API methods, it has the drawback of requiring changes to the data store client source code. In some cases, the client source code may be proprietary and not accessible. Even if the source code is available (e.g. via open source), properly modifying it to incorporate caching functionality can be time consuming and difficult.

The second approach for implementing caching is to provide the DSCL to users and allow them to implement their own customized caching solutions using the DSCL API. The DSCL provides convenient methods allowing applications to both query and modify caches. The DSCL also allows cached objects to have expiration times associated with them; later in this section, we will describe how expiration times are managed. It should be noted that if the first approach of having a fully integrated cache with a data store is used, it is still often necessary to allow applications to directly access and modify caches via the DSCL in order to offer higher levels of performance and data consistency. Hence, using a combination of the first and second caching approaches is often desirable.

The third approach for achieving caching is provided by the UDSM. The UDSM key-value interface is implemented for both main data stores as well as caches. If an application is using the key-value interface to access a data store, it is

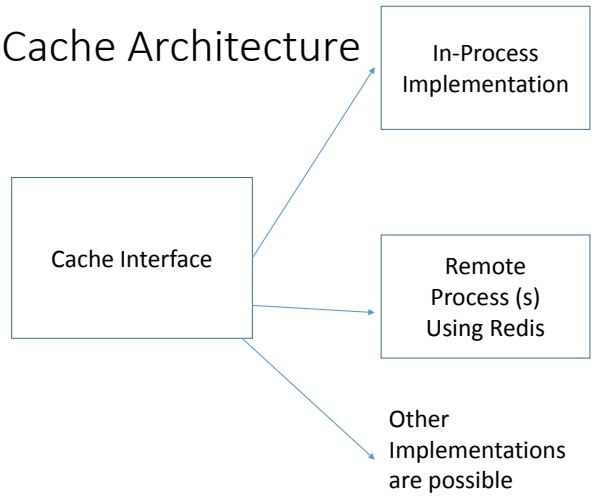


Fig. 4: Our modular architecture supports multiple cache implementations.

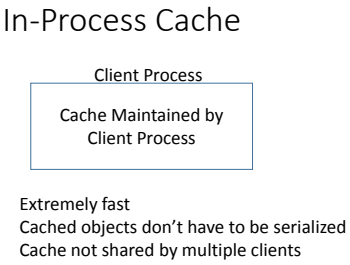


Fig. 5: In-process caches can be used to improve performance.

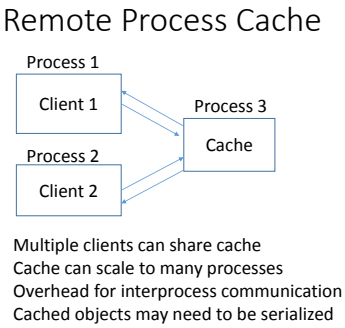


Fig. 6: Remote process caches can be used to improve performance.

very easy for the application writer to store some of the key-value pairs in a cache provided by the UDSM. Both the main data store and cache share the same key-value interface. This approach, like the second approach, requires the application writer to explicitly manage the contents of caches. Furthermore, the UDSM lacks some of the caching features provided by the DSCL such as expiration time management. An advantage to the third approach is that any data store supported by the UDSM can function as a cache or secondary repository for another data store supported by the UDSM; this offers a wide variety of choices for caching or replicating data.

The DSCL also supports multiple different types of caches via a Cache interface which defines how an application interacts with caches. There are multiple implementations of the Cache interface which applications can choose from (Figure 4).

There are two types of caches. In-process caches store data within the process corresponding to the application (Figure 5) [19]. 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. One consequence of this approach is that changes to the object from the application will change 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 (Figure 6) [17]. In this approach, the cache runs in one or more processes separate 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. Therefore, remote process caches are generally slower than in-process caches (as shown in Section V). 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.

There are several caches that are available as open source software which can be used in conjunction with our DSCL. Redis [7] and memcached [8] 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 in Java, C, C++, Python, Javascript, PHP, and several other languages.

Examples of caches targeted at Java environments include the Guava cache [9], Ehcache [10], and OSCache [11]. 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 DSCL allows any of these caches to be plugged into its modular architecture. In order to use one of these caches, an implementation of the DSCL Cache interface needs to be implemented for the cache. We have implemented DSCL Cache interfaces for a number of caches including redis and the Guava cache.

The DSCL 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

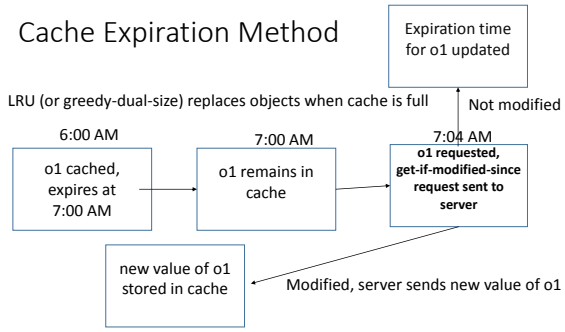


Fig. 7: Handling cache expiration times.

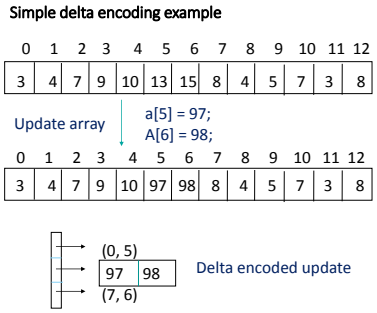


Fig. 8: Delta encoding.

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 DSCL 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 DSCL 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 DSCL 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 the server's version of o1 is different than the client's version. In order to determine if the client's 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 7).

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 [20] can be used to determine which objects to retain in the cache.

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.

The encryption capabilities of the DSCL can also be used in conjunction with caching. People often fail to recognize the security risks that can be exposed by caching. A cache may be storing confidential data for extended periods of time. That data can become a target for hackers in an insecure environment. Most caches do not encrypt the data they are storing, even though this is sometimes quite important.

Remote process caches also present security risks when an application is communicating with a cache over an unencrypted channel. A malicious party can steal the data being sent between the application and the cache. Too often, caches are designed with the assumption that they will be deployed in a trusted environment. This will not always be the case.

For these reasons, data should often be encrypted before it is cached. The DSCL provides the capability for doing so. There is some CPU overhead for encryption, so privacy needs need to be balanced against the need for fast execution.

The DSCL compression capabilities can also be used to reduce the size of cached objects, allowing more objects to be stored using the same amount of cache space. Once again, since compression entails CPU overhead, the space saved by compression needs to be balanced against the increase in CPU cycles resulting from compression and decompression.

## IV. Delta Encoding

Data transfer sizes between the client and server can sometimes 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 [21].

A simple example of delta encoding is shown in Figure 8. Only two elements of the array in the figure change. Instead of sending a copy of the entire updated array, only the delta shown below the array 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.

Our delta encoding algorithm uses key ideas from the Rabin-Karp string matching algorithm [22]. 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 8. 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, an 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 $b[i]$.

Delta encoding works best if the server has support for delta encoding. If the server does not have support for delta encoding, the client can handle all of the delta encoding operations using the following approach. The client communicates an update to the server by storing a delta at the server with an appropriate name. After some number of deltas have been sent to the server, the client will send a complete object to the server after which the previous deltas can be deleted. If a delta encoded object needs to be read from the server, the base object and all deltas will have to be retrieved by the client so that it can decode the object.

Delta encoding without support from the server will often not be of much benefit because of the additional reads and writes the client has to make to manage deltas. The other features of our enhanced data store clients do not require special support from the server. We consider delta encoding to be a performance optimization which can sometimes be of benefit but is not as important for our enhanced clients as caching, encryption, or compression.

## V. Performance Evaluation

Our enhanced data store clients and UDSM offer multiple benefits including improved performance via caching, encryption, compression, access to a wide variety of data stores via synchronous and asynchronous interfaces, performance monitoring, and a workload generator for easily comparing performance across different data stores. This section uses the UDSM to determine and compare read and write latencies that a typical client would see using several different types of data stores. We show the performance gains that our enhanced data store clients can achieve with caching. We also quantify the overheads resulting from encryption, decryption, compression, and decompression.

We test the following data stores by using the UDSM to send requests from its workload generator using the key-value interface:

- A file system on the client node accessed via standard Java method calls.
- A MySQL database [23] running on the client node accessed via JDBC.
- A commercial cloud data store provided by a major cloud computing company (referred to as Cloud Store 1).
- A second commercial cloud data store provided by a major cloud computing company (referred to as Cloud Store 2).
- A Redis instance running on the client node accessed via the Jedis client [24].

The Redis instance also acts as a remote process cache for the other data stores. A Guava cache [9] acts as an in-process cache for each data store. The performance numbers were all obtained using the common key-value interface which the UDSM implements for each data store.

A 2.70GHz Intel i7-3740QM processor with 16 GB of RAM running a 64-bit version of Windows 7 Professional was used for running the UDSM and enhanced data store clients. The performance graphs use log-log plots because both the x-axis (data size in bytes) and y-axis (time in milliseconds) values span a wide magnitude of numbers. Experiments were run multiple times. Each data point is averaged over 4 runs of the same experiment. Data from files in a file system were stored and retrieved from different data stores to generate the performance numbers. We did not find significant correlations between the types of data read and written and data store read and write latencies. There was often considerable variability in the read and write latency for the experimental runs using the same parameters.

Figure 9 shows the average time to read data as a function of data size. Cloud Store 1 and 2 show the highest latencies because they are cloud data stores geographically distant from the client. By contrast, the other data stores run on the same machine as the client. Another factor that could adversely affect performance for the cloud data stores, particularly in the case of Cloud Store 1, is that the requests coming from our UDSM might be competing for server resources with computing tasks from other cloud users. This may be one of the reasons why Cloud Store 1 exhibited more variability in read latencies than any of the other data stores. Significant variability in cloud storage performance has been observed by others as well [2]. The performance numbers we observed for Cloud Store 1 and 2 are not atypical for cloud data stores, which is a key reason why techniques like client-side caching are essential for improving performance.

Redis offers lower read latencies than the file system for small objects. For objects 50 Kbytes and larger, however, the file system achieves lower latencies. Redis incurs overhead for interprocess communication between the client and server. There is also some overhead for serializing and deserializing data stored in Redis. The file system client might benefit from caching performed by the underlying file system.

Redis offers considerably lower read latencies than MySQL for objects up to 50 Kbytes. For larger objects, Redis offers only slightly better read performance, and the read latencies converge with increasing object size.

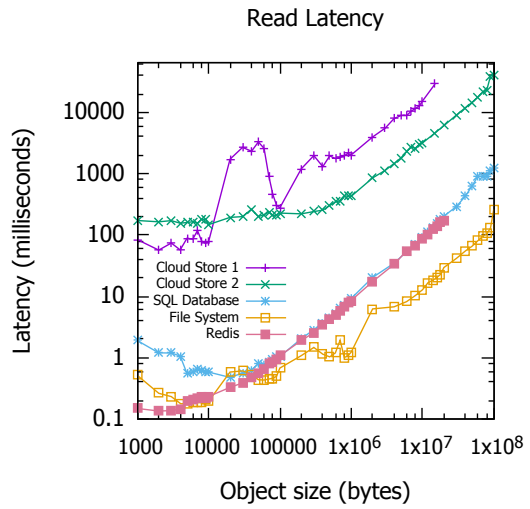


Fig. 9: Read latencies for data stores.

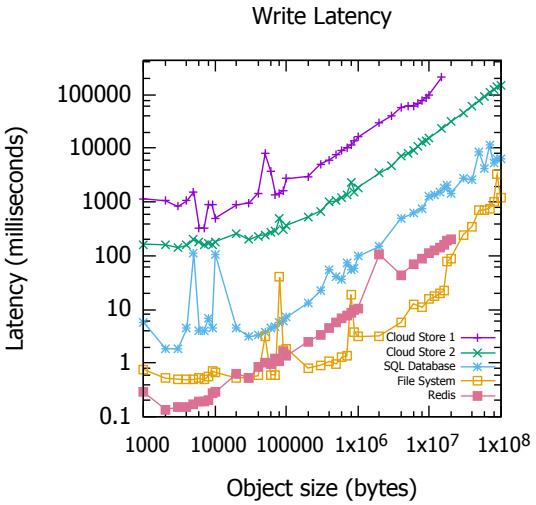


Fig. 10: Write latencies for data stores.

Figure 10 shows the average time to write data as a function of data size. Cloud Store 1 has the highest latency followed by Cloud Store 2; once again, this is because Cloud Store 1 and 2 are cloud-based data stores geographically distant from the client. MySQL has the highest write latencies for the local data stores. Redis has lower write latencies than the file system for objects of 10 Kbytes or smaller. Write latencies are similar for objects of 20-100 Kbytes, while the file system has lower write latencies than Redis for objects larger than 100 Kbytes.

Write latencies are higher than read latencies across the data stores; this is particularly apparent for MySQL for which writes involve costly commit operations. It is also very pronounced for larger objects with the cloud data stores and the file system. Write latencies for the file system and MySQL exhibit considerably more variation than read latencies.

Figures 11 - 19 show read latencies which can be achieved with the Guava in-process cache and Redis as a remote process cache. Multiple runs were made to determine read latencies for each data store both without caching and with caching when the hit rate is 100%. From these numbers, the workload generator can extrapolate performance for different hit rates. Each graph contains 5 curves corresponding to no caching and caching with hit rates of 25%, 50%, 75%, and 100%.

The in-process cache is considerably faster than any of the data stores. Furthermore, read latencies do not increase with increasing object size because cache reads do not involve any copying or serialization of data. By contrast, Redis is considerably slower, as shown in Figure 19. Furthermore, read latencies increase with object size as cached data objects have to be transferred from a Redis instance to the client process and deserialized. An in-process cache is thus highly preferable from a performance standpoint. Of course, application needs may necessitate using a remote process cache like Redis instead of an in-process cache.

Figure 18 shows that for the file system, remote process caching via Redis is only advantageous for smaller objects; for larger objects, performance is better without using Redis. This is because the file system is faster than Redis for reading larger objects. Figure 16 shows a similar trend. While Redis will not result in worse performance than MySQL for larger objects, the performance gain may be too small to justify the added complexity.

Figure 20 shows the times that an enhanced data store client requires for encrypting and decrypting data using the Advanced Encryption Standard (AES) [25] and 128-bit keys. Since AES is a symmetric encryption algorithm, encryption and decryption times are similar.

Figure 21 shows the times that an enhanced data store client requires for compressing and decompressiong data using gzip [26]. Decompression times are roughly comparable with encryption and decryption times. However, compression overheads are several times higher.

## VI. Related Work

Clients exist for a broad range of data stores. A small sample of clients for commonly used data stores includes the Java library for OpenStack Storage (JOSS) [5], the Jedis client for Redis [24] and the Java Driver for Apache Cassandra [6]. These clients do not have the capabilities that our enhanced clients provide. Amazon offers access to data stores such as DynamoDB via a software development kit (SDK) [27] which provides compression, encryption, and both synchronous and asynchronous APIs. Amazon's SDK does not provide integrated caching, performance monitoring, or workload generation; we are not aware of other clients besides our own which offer these features. Furthermore, we offer coordinated access to multiple types of data stores, a key feature which other systems lack.

Remote process caches which provide an API to allow applications to explicitly read and write data as well as to maintain data consistency were first introduced in [17], [28]. A key aspect of this work is that the caches were an essential component in serving dynamic Web data efficiently at several highly accessed Web sites. Memcached was developed several years later [8]. Design and performance aspects for in-process caches were first presented in [19].

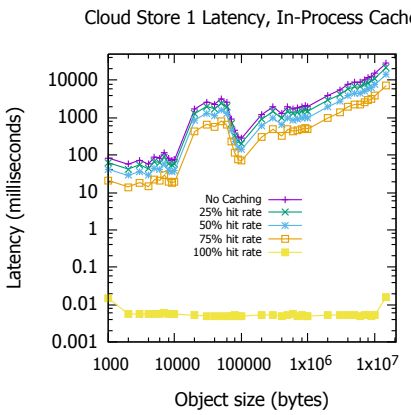


Fig. 11: Cloud Store 1 read latencies with in-process caching.

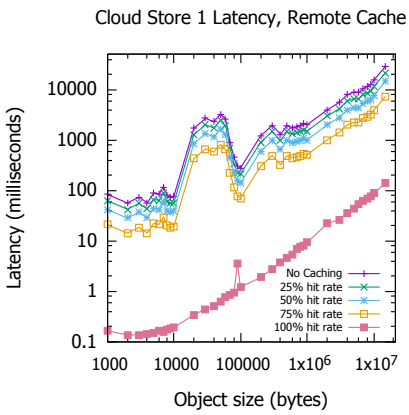


Fig. 12: Cloud Store 1 read latencies with remote process caching.

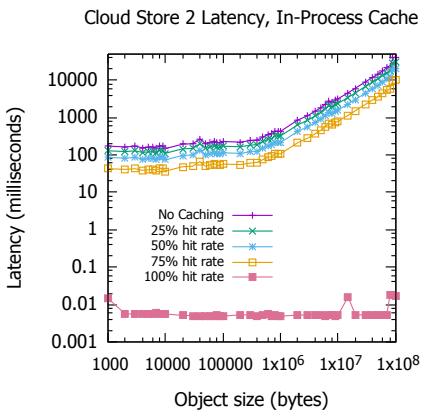


Fig. 13: Cloud Store 2 read latencies with in-process caching.

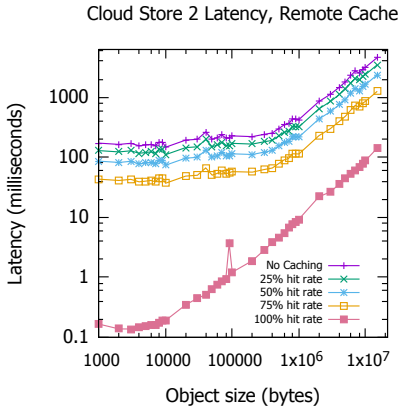


Fig. 14: Cloud Store 2 read latencies with remote process caching.

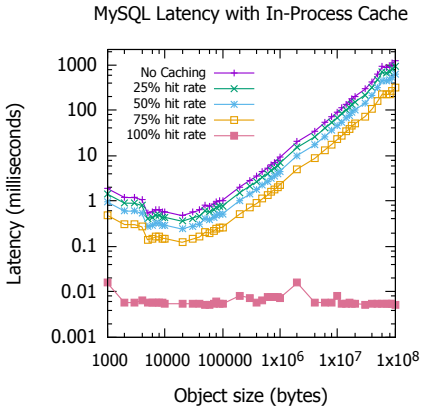


Fig. 15: MySQL read latencies with in-process caching.

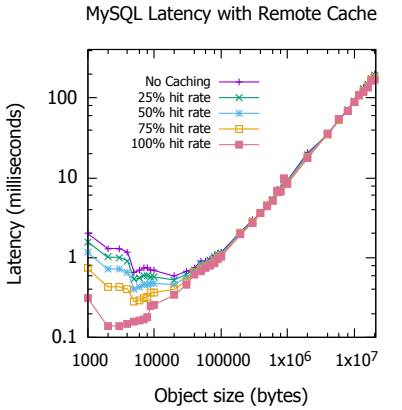


Fig. 16: MySQL read latencies with remote process caching.

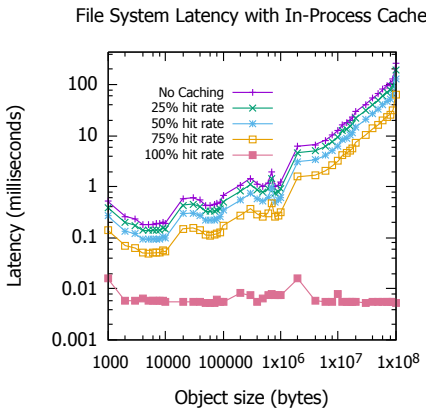


Fig. 17: File system read latencies with in-process caching.

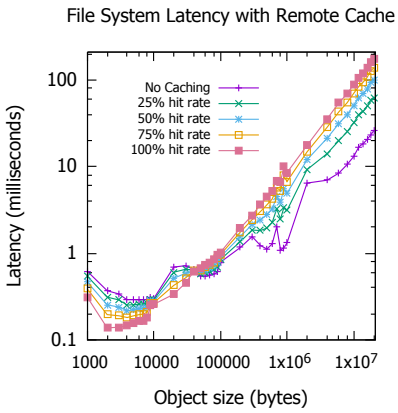


Fig. 18: File system read latencies with remote process caching.

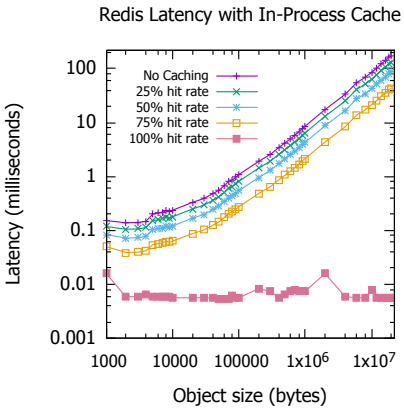


Fig. 19: Redis read latencies with in-process caching.

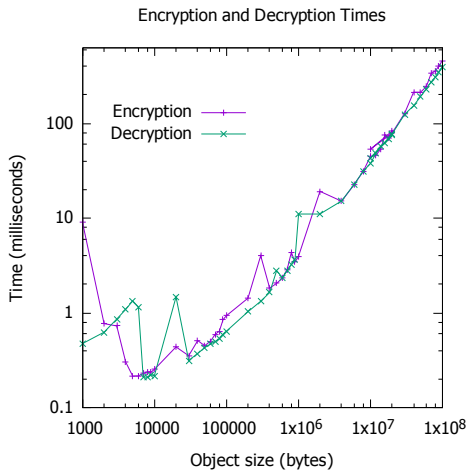


Fig. 20: Encryption and decryption latency.

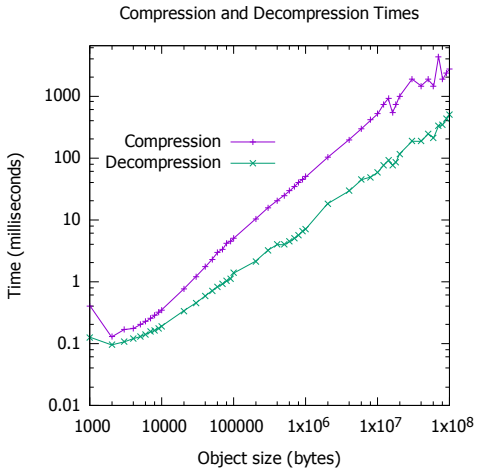


Fig. 21: Compression and decompression latency.

In recent years, there have been a number of papers which have looked at how to improve caches such as memcached. Facebook's use of memcached and some of the performance optimizations they made are described in [29]. A more detailed description of Facebook's memcached workload is contained in [30]. Cliffhanger is a system which optimizes memory use among multiple cache servers by analyzing hit rate gradient curves [31]. A memcached implementation of Cliffhanger described in the paper considerably reduces the amount of memory required to achieve a certain hit rate. Optimizations to memcached are presented in [32] including an optimistic cuckoo hashing scheme, a CLOCK-based eviction algorithm requiring only one extra bit per cache entry, and an optimistic locking algorithm. These optimizations both reduce space overhead and increase concurrency, allowing higher request rates. Expanding the memory available to memcached via SSDs is explored in [33]. A cache management system that performs adaptive hashing to balance load among multiple memcached servers in response to changing workloads is presented in [34]. Memory partitioning techniques for memcached are explored in [35]. MIMIR is a monitoring system which can dynamically estimate hit rate curves for live cache servers which are performing cache replacement using LRU [36].

There have also been a number of papers which have studied the performance of cloud storage systems. Dropbox, Microsoft SkyDrive (now OneDrive), Google Drive, Wuala (which has been discontinued), and Amazon Cloud drive are compared using a series of benchmarks in [1]. No storage service emerged from the study as being clearly the best one. An earlier paper by some of the same authors analyzes Dropbox [37]. A comparison of Box, Dropbox, and SugarSync is made in [2]. The study noted significant variability in service times which we have observed as well. Failure rates were less than 1%. Data synchronization traffic between users and cloud providers is studied in [38]. The authors find that much of the data synchronization traffic is not needed and could be eliminated by better data synchronization mechanisms. Another paper by some of the same authors proposes reducing data synchronization traffic by batched updates [39].

## VII. Conclusions

This paper has presented enhanced data store clients which support client-side caching, data compression, and encryption. These features considerably enhance the functionality and performance of data stores and are often needed by applications. We have also presented a Universal Data Store Manager (UDSM) which allows applications to access multiple data stores. The UDSM provides common synchronous and asynchronous interfaces to each data store, performance monitoring, and a workload generator which can easily compare performance of different data stores from the perspective of the client. A library for implementing enhanced data store clients as well as a Java implementation of the UDSM are available as open source software. Enhanced data store clients are being used by IBM customers.

Most existing data store clients only have basic functionality and lack the range of features that we provide. Users would significantly benefit if caching, encryption, compression, and asynchronous (nonblocking) interfaces become commonly supported in data store clients.

Future work includes providing more coordinated features across multiple data stores such as atomic updates and two-phase commits. We are also working on new techniques for providing data consistency between different data stores. The most compelling use case is providing stronger cache consistency. However, the techniques we are developing are also applicable to providing data consistency between any data stores supported by the UDSM.

## VIII. Acknowledgments

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

## References

[1] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking Personal Cloud Storage," in *Proceedings of IMC '13*, 2013, pp. 205–212.

[2] R. Gracia-Tinedo, M. Artigas, A. Moreno-Martinez, C. Cotes, and P. Garcia-Lopez, "Actively Measuring Personal Cloud Storage," in *Proceedings of the IEEE 6th International Conference on Cloud Computing*, 2013, pp. 301–308.

[3] J. Dean and P. Norvig, "Latency numbers every programmer should know," https://gist.github.com/jboner/2841832.

[4] Cloudant, "Cloudant java client," https://github.com/cloudant/java-cloudant.

[5] Javaswift, "Joss: Java library for openstack storage, aka swift," http://joss.javaswift.org/.

[6] DataStax, "Datastax java driver for apache cassandra," https://github.com/datastax/java-driver.

[7] Redis, "Redis home page," http://redis.io/.

[8] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, no. 124, p. 5, 2004.

[9] C. Decker, "Caches explained," https://github.com/google/guava/wiki/CachesExplained.

[10] Ehcache, "Ehcache: Java's most widely-used cache," http://www.ehcache.org.

[11] OSCache, "Oscache," https://java.net/projects/oscache.

[12] IBM, "Data store client library," https://developer.ibm.com/open/data-store-client-library/.

[13] ——, "Universal data store manager," https://developer.ibm.com/open/universal-data-store-manager/.

[14] A. Iyengar, "Enhanced Storage Clients," IBM Research Division, Yorktown Heights, NY, Tech. Rep. RC 25584 (WAT1512-042), December 2015.

[15] ——, "Universal Data Store Manager," IBM Research Division, Yorktown Heights, NY, Tech. Rep. RC 25607 (WAT1605-030), May 2016.

[16] Google, "Listenablefutureexplained," https://github.com/google/guava/wiki/ListenableFutureExplained.

[17] A. Iyengar and J. Challenger, "Improving Web Server Performance by Caching Dynamic Data," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997, pp. 49–60.

[18] Cloudant, "Java cloudant cache," https://github.com/cloudant-labs/java-cloudant-cache.

[19] A. Iyengar, "Design and Performance of a General-Purpose Software Cache," in *Proceedings of the 18th IEEE International Performance, Computing, and Communications Conference*, 1999, pp. 329–336.

[20] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997, pp. 193–206.

[21] F. Douglis and A. Iyengar, "Application-specific Delta-encoding via Resemblance Detection," in *Proceedings of the USENIX 2003 Annual Technical Conference*, 2003, pp. 113–126.

[22] R. Karp and M. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[23] MySQL, "Mysql home page," https://www.mysql.com/.

[24] Jedis, "A blazingly small and sane redis java client," https://github.com/xetorthio/jedis.

[25] NIST, "Announcing the Advanced Encryption Standard (AES)," National Institute of Standards and Technology, Tech. Rep. Federal Information Standards Publication 197, November 2001.

[26] Gzip, "The gzip home page," http://www.gzip.org/.

[27] Amazon, "Aws sdk for java," https://aws.amazon.com/sdk-for-java/.

[28] J. Challenger and A. Iyengar, "Distributed Cache Manager and API," IBM Research Division, Yorktown Heights, NY, Tech. Rep. RC 21004 (94070), 1997.

[29] R. Nishtala *et al.*, "Scaling Memcache at Facebook," in *Proceedings of NSDI '13*, 2013, pp. 385–398.

[30] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of SIGMETRICS '12*, 2012, pp. 53–64.

[31] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling Performance Cliffs in Web Memory Caches," in *Proceedings of NSDI '16*, 2016, pp. 379–392.

[32] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *Proceedings of NSDI '13*, 2013, pp. 371–384.

[33] X. Ouyang *et al.*, "SSD-assisted Hybrid Memory to Accelerate Memcached Over High Performance Networks," in *Proceedings of ICPP 2012*, 2012, pp. 470–479.

[34] J. Hwang and T. Wood, "Adaptive Performance-Aware Distributed Memory Caching," in *Proceedings of ICAC '13*, 2013, pp. 33–43.

[35] D. Carra and P. Michiardi, "Memory Partitioning in Memcached: An Experimental Performance Analysis," in *Proceedings of ICC 2014*, 2014, pp. 1154–1159.

[36] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic Performance Profiling of Cloud Caches," in *Proceedings of SoCC '14*, 2014, pp. 1–14.

[37] I. Drago, M. Mellia, M. Munaf, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding Personal Cloud Storage Services," in *Proceedings of IMC '12*, 2012, pp. 481–494.

[38] Z. Li *et al.*, "Towards Network-level Efficiency for Cloud Storage Services," in *Proceedings of IMC '14*, 2014, pp. 115–128.

[39] ——, "Efficient Batched Synchronization in Dropbox-like Cloud Storage Services," in *Proceedings of Middleware 2013*, 2013, pp. 307–327.