Preliminary version. Final version appears in *Proc of 2nd IEEE International Conference on Cloud Computing Technology and Science*. Nov 30-Dec 3, 2010 . Indianapolis, Indiana.

CSAL: A Cloud Storage Abstraction Layer to Enable Portable Cloud Applications *(work-in-progress)*

Zach Hill
Department of Computer Science
University of Virginia
zjh5f@cs.virginia.edu

Marty Humphrey
Department of Computer Science
University of Virginia
humphrey@cs.virginia.edu

Abstract

One of the large impediments for adoption of cloud computing is perceived vendor lock-in with respect to both low-level resource management and application-level storage services. Application portability is essential to both avoid lock-in as well as leverage the ever-changing landscape of cloud offerings. We present a storage abstraction layer to enable applications to both utilize the highly-available and scalable storage services provided by cloud vendors and be portable across platforms. The abstraction layer, called CSAL, provides Blob, Table, and Queue abstractions across multiple providers and presents applications with an integrated namespace thereby relieving applications of having to manage storage entity location information and access credentials. Overall, we have observed minimal overhead of CSAL on both EC2 and Windows Azure.

1. Introduction

One of the more significant issues facing cloud computing is the perception of "vendor lock-in" [1]. That is, because of the different abstractions and mechanisms of the different clouds, a developer is forced to choose one cloud provider in the early stages and then produce a solution customized for that platform. But in doing so, the developer is tied to that platform and cannot easily change, even if, for example, another cloud provider comes on-line and is cheaper or provides better service. It is very difficult to re-target a deployed cloud application to another provider or platform due to the wide variance between syntax and semantics of the APIs (both "Storage APIs" and "Compute APIs") and even the level of abstraction provided by the different infrastructures (e.g. SaaS, IaaS, PaaS).

There are a number of approaches to address this problem of potential vendor lock-in:

- *A developer can choose only those APIs that have multiple independent implementations.* For example, a developer might be less worried about being locked into Amazon EC2 [1] given the emergence and popularity of Eucalyptus [18] (or likewise Google AppEngine[13] and AppScale[3]), which is API-equivalent to Amazon's EC2 service. One disadvantage of such an approach is that counting on multiple implementations to maintain equivalence is precarious at best, and given that the shared API is generally driven by the originating provider there may periods of inconsistency between the implementations. Another challenge is finding two implementations of the same API that are equivalent in terms of scale/scalability, features, maturity, and customer support. For example, while Eucalyptus implements the EC2 API it does not include storage services equivalent to all of those offered by Amazon, and Eucalyptus itself is not a commercial cloud infrastructure, only a software stack.
- *A developer can choose a particular API that can run on multiple clouds, though not necessarily through multiple independent implementations.* MapReduce [5] and Hadoop [14] are the best examples of this approach. However, such APIs tend to be focused on a particular application model, which may not fit every developer's requirements. This approach may also require significant developer time investment for configuring, deploying, and maintaining these services as they are provided at varying levels of automation to the cloud developer. Finally, implementations are often tailored to the specific vendor offering the service, thereby limiting portability as the configuration may differ between vendors.
- *A developer can manually separate the application into an "app-logic layer" and a "cloud layer" (with code written for each cloud provider).* While this is the most general option, it requires a significant time and complexity investment by a developer to initially create the layers and further, and often more importantly, maintain them over time as the underlying APIs change.
- *A developer can wait and hope that a cloud computing set of standards is developed and program against*

those. While this is the best solution in terms of general cloud interoperability, it is far from being realized in the commercial space as each vendor has developed its own API and set of abstractions, and as vendors gain market share it will require more external pressure to get the vendors to adopt industry standards. Others also argue that standards will limit the ability of vendors to differentiate themselves on features rather than just cost. From the vendors' perspective there is very little incentive to standardize because lock-in secures longer-term revenue. All of this means that standardization could take a long time to be realized, if at all.

In this paper, instead, we describe our vendor-independent cloud abstraction layer that presents a set of cloud storage abstractions ("Cloud Storage Abstraction Layer", or "CSAL"). CSAL features Blobs, Tables, and Queues, and is implemented with a plug-in architecture for extensibility. We purposefully address only storage services for CSAL rather than computing services (i.e. interacting with virtual machines) for two reasons: a) there are several projects working on standardizing virtual machine (VM) formats to enable portability [9][15][19][20]; and, b) applications themselves rarely interact directly with the compute resource APIs but are heavily dependent on vendor-provided storage services for available and scalable storage. Portable virtual machines are a necessary but not sufficient component of application portability because most applications require a scalable storage system that is persistent and VM storage is not always persistent in many cloud platforms.

CSAL allows an application to be portable between cloud providers and to access storage across cloud boundaries without modifying its code. This is a significant benefit over the user-created layering as described as the third option above because it allows a developer to focus on application logic rather than developing and maintaining storage abstractions.

There are several excellent projects working to create a single interface to multiple storage services using varying levels of abstraction, but they do not provide a unified namespace across clouds and they require the application to know where a specific data object is located and which service to use to access it. Cloudloop[5] provides a filesystem-like interface to blob storage, but does not support other abstractions such as tables and queues. The jclouds[16] project provides a framework for interacting with both blob storage, queue storage, and compute resources for a variety of clouds but requires service and location information to access a data item. Dasein Cloud API[7] includes blob storage, compute, and network abstractions, but does not include table or queue-based storage, and because it aims to interface with all aspects of cloud infrastructures it is quite complex. SimpleCloud[24] is a PHP API for interacting with blob, table, and queue storage services of several providers. It is a good PHP implementation, but PHP is typically limited to web applications.

The critical difference between the existing multi-cloud API projects and CSAL is the idea of a unified namespace across providers and the abstraction to the application that a single cloud exists rather than a set of clouds which must be managed independently. CSAL maintains state about the namespace it manages and thus is more than a simple API pass-through to translate calls to web services.

The SAGA API [22], has similar goals as CSAL, but in the Grid Computing space. SAGA presents a complete application API for the entire execution environment and thus can be quite complex. It attempts to abstract the grid application away from the details of its execution environment in a way that is largely achieved by virtualization in typical clouds and thus CSAL focuses exclusively on storage resources in order to maintain simplicity.

The contributions of this paper are:

- We show the design of a set of single set of cloud storage abstractions that can be mapped to existing (and near-future) cloud offerings.
- We show its efficient implementation on two of the major cloud providers (EC2 and Azure [17]) and describe our plans for more cloud implementations.
- We argue that the technique described in this paper is a valuable foundation for a *multi-cloud application,* whereby an application spans multiple clouds at once, essentially leveraging the capabilities of multiple cloud infrastructures.

Overall, we believe that the value of CSAL is *not* the particular storage abstractions supported (e.g., Blobs, Tables, and Queues) but rather the ability of our CSAL implementation to provide good performance, a single unified view of cloud storage across platforms, and to manage the metadata necessary for utilizing storage services across multiple clouds.

The rest of this paper is organized as follows: Section 2 describes the overall design of CSAL and introduces the main components. Section 3 provides an overview of the implementation and issues encountered. Section 4 presents a performance evaluation of the CSAL and its overhead versus programming directly against a given cloud API. Sections 5 and 6 present a discussion of the future of CSAL and our conclusions respectively.

2. Design of CSAL

CSAL provides the application developer with three storage abstractions: blobs, tables, and queues. These three abstractions should be familiar to those who have used Amazon's or Microsoft's cloud infrastructures. We

have intentionally kept our basic storage abstractions close to those already provided by multiple vendors because users are familiar with the basic concepts and could thus easily transition to using CSAL without having to learn new abstractions or figure out how to map their existing storage objects to a new structure. We do not directly address whether these storage abstractions are the *best* abstractions for scalable storage in cloud computing, as others do [1][21]; however, we hope to gain some insight on the issue through the course of building and using CSAL on multiple platforms.

The consistency semantics provided by CSAL are the same as those of many cloud storage services: eventual consistency. Eventual consistency [25] guarantees that data will be consistent eventually, but there may be a window of time in which strong consistency is not guaranteed. CSAL does not implement any techniques for strengthening the consistency guarantees of the underlying services it utilizes.

Increasingly, vendors are providing the option of operations with stronger consistency semantics but with a performance penalty as well as faster eventually-consistent versions. CSAL currently does not utilize these stronger consistency operations, but they could be easily incorporated as a preferred mode of operation by the plug-ins, which interact directly with the storage services. Providing transparent strong consistency under the assumption of eventual consistency is preferable to the reverse because the semantics of an application will not be changed if it is assuming eventual consistency.

CSAL is designed around providing three storage abstractions that we describe next. We next present each abstraction and the operations it supports. We will then describe the design for namespaces and metadata management.

2.1. Blob

The blob abstraction is a common cloud computing storage abstraction in which data is organized into an unstructured sequence of bits that have simple get/put semantics and are intended to store potentially large amounts of data in a single object. In CSAL, blob names are only valid within a container—CSAL represents these containers only as a name. A blob's name is only required to be unique within its container, but a single container may house many blobs.

The interface to perform operations on blob containers and to get/put blobs is the *BlobStore*. Table 1 is a summary of the operations available on the *BlobStore* interface and the semantics of each.

The blob semantics of CSAL more closely follow those found in Amazon's S3 service in that blobs are immutable objects which may be replaced wholly, but not modified in place and which are operated only as an atomic unit. We chose this abstraction because it is simpler and easier to understand for application developers.

Table 1. BlobStore API

Operation	*Description*
createContainer	Create an empty blob container
deleteContainer	Deletes a container
listContainers	Lists all blob containers
listBlobs	Lists the BlobObjects stored in the named container
getBlob	Get the BlobObject
putBlob	Create or overwrite the blob specified in the named BlobContainer.

2.2. Table

CSAL's table storage provides an abstraction for storing semi-structured data composed of sets of named attributes. A table is a named set of rows, each of which has a unique identifier—a *RowKey*. A *TableRow* is a set of *RowAttributes*, and *TableRows* within a given table need not have the same sets of *RowAttributes*. A *RowAttribute* is a name-value pair associated with a type (string, binary, date, etc). Tables are semi-structured because no schema is enforced on a table by the storage system. *TableRows* can be accessed via get/put based on *RowKey* or via a query interface that allows querying on any attribute or a set of attribute.

The *TableStore* in CSAL is the primary interface to interact with tables. Table 2 below summarizes the operations supported by the *TableStore*.

Table 2. TableStore API

Operation	*Description*
createTable	Create new table, location may be specified
deleteTable	Delete a table by name
listTables	List tables
query	Query table either by RowKey or by a set of attributes with comparison and logical operators
insert	Insert a new row into a table
update	Update a row
delete	Delete a row based on RowKey
consistentRead	A strongly consistent read, but slower than a normal query

The query operation allows queries against a table to be based on the *RowKey* or a set of other attributes and standard comparison and logical operators are available. Of particular note in the *TableQuery* interface is the lack of a *join* operation or the concept of foreign keys as found in nearly every relational database implementation. The *join* operation is not supported in this abstraction, as is the

case in all of the major commercial cloud storage services.

2.3. Queue

Queues abstract a FIFO storage structure with the semantics typically expected of a queue data-structure. Queues are relatively simple abstractions that provide get, put, and peek operations to retrieve messages, add messages, and view the end of the queue without altering its state respectively. Table 3 shows the API calls. Additionally, CSAL adopts the message visibility-timeout feature found in many queue implementations. Associated with each message is a timeout value that initializes a timer when the message is retrieved from the end of the queue.

Between when the message is retrieved and the timer has not expired, the message is not visible to any subsequent 'get' calls by any client. If the timer expires before the message is explicitly deleted by the client that retrieved it then the message will automatically become visible, or retrievable, again to any following operations to view the end of the queue. This feature slightly violates strict FIFO rules, but is very useful in highly-scalable systems where nodes may fail while processing a message and the message contains information that needs to be reprocessed by another node.

Table 3. QueueStore API

Operation	*Description*
createQueue	Create new queue by name
deleteQueue	Delete a queue by name
listQueues	List all queues
getMessage	Returns the message at the tail of the queue and sets its visibility to false for the duration of a timer
putMessage	Insert a message into the head of the queue
viewMessage	Similar to getMessages, but does not change the visibility of the message or move the tail of the queue.
deleteMessage	Deletes the message with the given message receipt
getMessageCount	Returns the number of messages in the queue
clearMessages	Deletes all messages in the queue without deleting the queue itself

2.4. Namespaces

A blob container in CSAL is wholly located within a single storage service such as Amazon's S3 and does not span services. Additionally, CSAL is designed to provide a unified namespace for blob containers so a container can simply be referenced by its name and container names must be unique across a user's accounts in all the services accessible in a given CSAL deployment. The advantage to such a design is that an application does not have to keep track of where containers are located unless it explicitly wants to in order to optimize performance or cost, for example. CSAL will eventually be able to optimize the location of storage for the application based on various metrics, which we discuss further in Section 5.

2.5. Metadata

Because CSAL presents a single namespace across all storage services utilized it must maintain metadata about each storage container entity—blob containers, tables, and queues. The metadata management component utilizes a table-base storage service to store service, credential, and naming information about each storage entity. We do not utilize explicit synchronization of metadata between clients but rather rely on the backing store as the synchronization point.

CSAL can accept relaxed consistency models for metadata storage because in the event that data is stale the client will fail when attempting to access the storage entity and the metadata manager simply catches the failure, re-fetches the metadata, and retries the operation. Because we only maintain metadata for container entities rather than the data-objects themselves, we expect that metadata updates will be rare and this will not significantly impact performance. We provide more details on how metadata is managed in the implementation section.

3. Implementation of CSAL

The implementation of the CSAL is architected in a plug-in style and thus the principle implementation challenge is mapping the storage abstractions' requirements into actual calls to a storage service. Our implementation currently supports Microsoft's Windows Azure Platform and Amazon Web Services storage service (S3, SimpleDB, and SQS).

CSAL currently leverages readily available Java APIs for the cloud services supported in order to build the connector objects interact with the vendors' services directly. Our implementation leverages the Windows4j Java API[26] for Windows Azure built by Soyatec as well as the Amazon Web Services provided Java API implementations for S3, SimpleDB, and SQS. CSAL provides a namespace management and metadata layer on top of the existing API implementations, although a completely custom plug-in which speaks REST to the underlying services will be implemented in the future.

The Windows4j package handles REST communications with Windows Azure's Blob Service, Table Service, and Queue Service. The AWS

implementations use the S3 HTTPS SOAP interface and the REST interfaces for SimpleDB and SQS. A long-term goal is to remove these intermediate layers and implement direct calls to the services ourselves in order to improve performance and enable code-reuse.

In implementing the various connector classes we found that while the basic abstractions are often similar between vendors the specific API details often require significant workarounds to achieve the common CSAL semantics and naming conventions.

One example is the difference between Windows Azure blobs and AWS S3's blob storage services. Windows Azure Blob Service requires that all blobs larger than 64MB use a block-type interface in which a blob is composed of a set of smaller blocks and a listing of the ordering of the blocks. Thus, for large blobs the connector for Azure must break the data into smaller pieces and create a manifest and then upload them and the ordering to the Azure Blob Service in two distinct operations whereas Amazon supports only a single-piece blob using a single get/put operation.

The naming scope of each service presented barriers as well. Windows Azure defines names within a namespace prefixed automatically by the user's access key (username) whereas AWS requires that names be globally unique. In practice this means that AWS names are usually prefixed by the user's access key, a non-protected key so a layer of name translation between the AWS connectors and CSAL was required in order to shield the application developer from having to know the specific AWS account access key and naming containers to conform to AWS requirements.

One difficult hurdle with regard to any cross-platform abstraction layer is the performance variation across platforms. Depending on which service is being used and how the CSAL connector is implemented, the same operation, for instance a *putBlob* call, could have dramatically different performance. The block-type blob interface in Windows Azure mentioned previously can lead to either increased performance due to parallel block transfers or slower performance if used sequentially due to the 2-operation commit process required--upload the blocks, then send the ordering information.

3.1. Managing Metadata

As mentioned in Section 2, CSAL manages metadata for container-level entities—blob containers, tables, and queues—as well as for the cloud storage services themselves. Storage container entities are uniquely named across all services used and thus a lookup of a name should at most resolve to a single service and entity. This allows the application to simply refer to these containers, tables, or queues by a simple name rather than having to keep track of location information and credential information and also allows applications to move entities to new locations without having to explicitly inform all other components that the entity has been moved to a new service.

The implementation of CSAL uses a table-based back-end storage service to keep the entity metadata as well as service metadata such as account credentials and service endpoints. In our experiments we utilized SimpleDB and the Azure Table service for metadata storage. The service information for the metadata backing-store is included in the configuration file used by the application client to configure CSAL. Because accessing metadata requires network activity and can quickly degrade performance, caching of metadata is used whenever possible. This includes both service metadata as well as entity metadata.

All metadata is organized by a single namespace identifier. The metadata manager utilizes a set of tables, one for each storage type and namespace pair. Thus, the performance of applications concurrently accessing metadata for different namespaces is not impacted.

4. Evaluation

To evaluate the performance overhead of CSAL we measured the difference in the time to execute calls to storage resources located within the same cloud infrastructure using CSAL compared to the native, vendor-provided cloud-specific API. We ran a series of micro-benchmarks testing the total latency of each storage operation using both the vendor-recommended or provided Java API and the CSAL call. Each operation was run 100 times and the latency recorded using the Java timer *System.nanotime()* to record the time immediately before and after the API call being tested. We also instrumented the CSAL code to get a breakdown of the total operation time into both core operation time, the time to perform the actual intended operation, as well as the time for metadata operations. In all graphs we show the median value because it is less affected by outliers, and the error bars on each graph show one standard deviation of the data series.

The payload data used for the benchmarks is a 155KB text file for the blob operations, address information (first name, last name, city, state, ID#) in string fields for the table tests, and a simple short string, approximately 100 characters, for the queue tests. We chose small payloads to ensure that the payload data transfer time would be minimal allowing us to see the affects of the API implementation on the operation performance.

The configuration of the CSAL for the tests was set so that the metadata storage used to maintain the namespace was stored in the same cloud infrastructure as the test in order to more directly compare the two APIs by not introducing unnecessarily large overheads to the metadata service requests. In a real deployment scenario it is possible that an application component using CSAL and the metadata service back-end could be in entirely

different clouds potentially very far geographically from each other. We omit performance measurements from such as scenario as the network overhead would be so significant that the difference between the native API and CSAL would merely be the difference between LAN and WAN performance. While such an evaluation would be useful for a specific application deployment with specific requirements, for general benchmark purposes we think it offers much less insight.

For each of the performance benchmarks, we compare CSAL to the Java SDK API provided by each cloud vendor—in our case Amazon and Microsoft. Because the actual storage services provide a REST interface the APIs provided by the vendors merely create the requests and parse responses from the services. Therefore, we make no claims that the APIs provided by the vendors are the best performing or that third-party implementations of the APIs for the same services would not be able to be more efficient. We compare CSAL to the vendor provided APIs because they are the de-facto choices for application developers.

4.1. Amazon Web Services

We evaluated CSAL's performance overhead on Amazon's EC2 platform against the S3, SimpleDB, and SQS services using the Amazon-provided Java APIs for each service as a baseline. Each test was run from within EC2's 'us-east' region on a Fedora Core 8 32-bit instance.

Figure 1 shows the performance of container-level operations-creating blob containers, tables, and queues-benchmarks in Amazon's EC2 using both the API library provided by Amazon as well as CSAL. The slowdown seen is expected because CSAL requires additional metadata be stored in the backing-store (SimpleDB in this case) for all container-level operations. The metadata operations required are a single table-row lookup and a single table-row 'put' operation, the combined time for both the lookup and get is shown as the 'Metadata Op Time' column in Figure 1. The 'CSAL Total Time' shows the total time to perform the intended operation and is composed of the 'Core Op Time' and the 'Metadata Op Time' as well as all client-local overheads. The overhead for table creation is low because the native operation itself dwarfs the cost of the additional metadata operations required by CSAL.

Figure 2 shows the performance of CSAL when compared to the Java API for AWS for data-object operations is very similar and incurs little overhead. Note that the variance between operations was very high in several tests. In this specific case the metadata for container used was cached locally as a result of the client using CSAL for the container creation operation. If the container had not been previously used or created on that node, then we would expect to see an overhead increase for the metadata lookup of the container being used, but only on the first operation. All subsequent operations would use the cached data rather than having to perform more lookups.

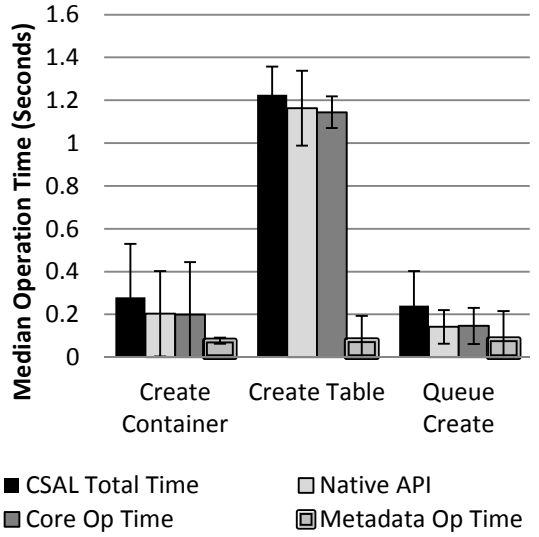

Figure 1 Time Breakdown of Container-Level Operations of CSAL in AWS. Error bars indicate one standard deviation.

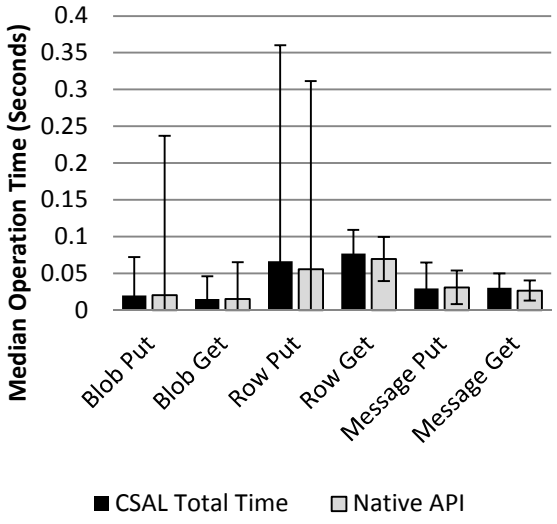

Figure 2 Performance of CSAL vs AWS Native API in Data-Object Operations. Error bars indicate one standard deviation.

In some cases the CSAL data-object performance is actually slightly better than the native API. There were a relatively small number of operations that experienced very long delays in the AWS native API test runs which affect the median calculation even though they are not indicative of the typical performance, and since we are showing the median value the data points reflect a single operation execution. The error bars displaying the standard deviation of these tests also show how widely the

performance varied between iterations of the same operation and exemplify one property of all cloud storage services, which is that performance may vary significantly from one invocation to the next based on factors beyond a developer's control.

4.2. Windows Azure Platform

We performed the same set of benchmarks on the Windows Azure Platform service. The code used for both AWS and Azure is exactly the same. The only difference is a few lines in the configuration files which specify to use Azure for metadata storage and to use Azure as the default service location for all storage operations.

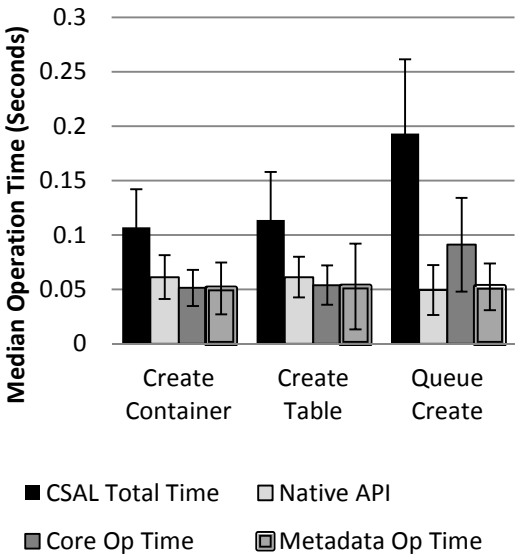

Figure 3 Time Breakdown of Container-Level Operations of CSAL in Windows Azure. Error bars indicate one standard deviation.

In contrast to the performance of CSAL on AWS, we see that in the Azure platform, CSAL incurs a higher performance penalty for container-level operations, but part of the reason is that the Azure container-level operations themselves are significantly faster in AWS, so any additional processing and network time shows a more significant impact. Figures 3 and 4 are the performance of CSAL in Windows Azure. Note how the vertical axis scales differ from those of Figures 1 and 2 in their magnitude. We saw consistently lower latencies in Azure for the container-level operations. This may be a product of differing replication strategies between Azure and AWS or how storage resources are allocated with respect to compute resources. However, because the primary goal of this paper is not to compare the performance of AWS to Azure, we will forgo a more lengthy analysis of what may cause the difference in performance.

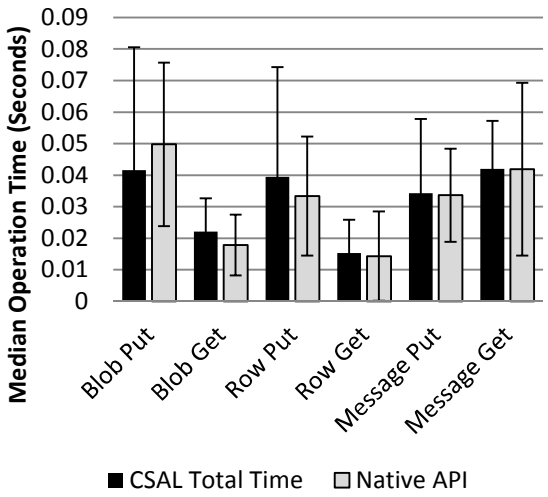

Figure 4 Performance of CSAL vs Azure Native API in Data-Object Operations. Error bars indicate one standard deviation.

In Azure, CSAL exhibits slowdown within 2X for all container-level operations except the queue creation operation, which exhibited a slowdown in the core operation as seen in Figure 3. The core operation time itself was significantly slower for the queue creation run. The core operation time is essentially the time to do the exact same operation as the 'Native API' operation, so either the network or the storage service exhibited some performance degradation which impacted the results of the core operation and the overall slowdown is attributed to that factor rather than the impact of the metadata operations themselves.

5. Discussion

Abstraction layers have the inherently difficult position of trying to remove detail from the model presented to developers while still maintaining performance and flexibility. There is always a tradeoff between exposing lots of detail to programmers, who can customize for performance as they see fit, and providing simpler interfaces that are easier to use but prevent tuning for performance. CSAL does make flexibility tradeoffs for ease-of-use, but it does allow a degree of flexibility to remain through the use of user-created plug-ins. If required, optimizations can be incorporated into a custom plug-in for use against a specific service.

Our work on CSAL to date has focused on abstracting the storage resources provided by many cloud infrastructures; however, the long-term goal of CSAL is to provide a cross-cloud platform such that applications can be deployed easily across clouds to take advantage of

the quickly changing landscape of cloud infrastructure offerings.

One criticism of any API as a research tool is that it is difficult to argue its 'correctness'. Rather than arguing that our API is *the* correct API for cross-cloud applications we merely posit that CSAL represents *an* API and that a common API is one way to address issues of cloud vendor lock-in as well as application portability and mobility. We intend CSAL to be a tool with which we can study how developers and scientists might approach applications that have extremely high availability requirements, such that a single provider is a single point of failure, as well as those that seek to exploit the ever-changing nature and costs of cloud computing as the marketplace develops.

The CSAL API provides developers with a simple storage API which works on multiple platforms to access the vendor provided storage services, but what if a user wants to leverage CSAL on a cloud that does not provide the same abstractions or even any storage services at all? Examples of such clouds are Eucalyptus[11], GoGrid[12], and Rackspace[23], which provide either no storage services or provide only a blob-type service. There are, however many third-party storage systems available that can be installed and used within such infrastructures. Third party services can be leveraged by CSAL using the connector interface just as SimpleDB or Azure Queues are. Ultimately, out goal with CSAL is to leverage application portability to explore multi-cloud application deployments and management as well as dynamic resource allocation optimization for cost and performance metrics.

5. Conclusion

Vendor lock-in is a difficult problem that needs to be addressed immediately because the cloud landscape is just taking shape. Standards take too long to develop, so we have presented CSAL, a vendor-agnostic abstraction layer that sits above cloud-specific APIs to provide common storage abstractions for multiple cloud platforms and to support highly portable applications even if the data itself is not as portable due to size and/or cost. Our performance results show that this layer imposes small overhead for common data-object operations—and reasonable overhead—typically within 2X—for metadata-intensive container-level operations.

CSAL provides a set of generic storage abstractions common to many cloud platforms. By combining a unified namespace across all supported platforms as well as high-level abstractions, CSAL allows an application to be moved with nearly no code changes and provides a foundation for multi-cloud applications that, because of mobility, can exploit the dynamic nature of the cloud landscape optimize costs and/or performance.

References

[1] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari, "Cloud Analytics: Do We *Really* Need to Reinvent the Storage Stack?" in *Proc. of Usenix HotCloud'09,* 2009.

[2] Amazon. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/

[3] AppScale. http://code.google.com/p/appscale/

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M.i Zaharia. "Above the Clouds: A Berkeley View of Cloud Computing." Univ. of California, Berkely, Technical Report EECS-2009-28, 2009.

[5] Cloudloop. http://www.cloudloop.com

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. of Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[7] Dasein Cloud API: http://dasein-cloud.sourceforge.net/

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters." in *Proc. of Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[9] DeltaCloud. http://deltacloud.org/

[10] DMTF Open Virtualization Format. http://www.dmtf.org/standards/published_documents/DSP0243_1.0.0.pdf

[11] Eucalyptus. http://www.eucalyptus.com

[12] GoGrid. http://www.gogrid.com

[13] Google AppEngine. http://code.google.com/appengine/

[14] Hadoop. http://hadoop.apache.org/

[15] Libcloud: http://incubator.apache.org/libcloud/

[16] jclouds. http://www.jclouds.org/

[17] Microsoft Windows Azure Platform. http://www.microsoft.com/windowsazure/windowsazure/

[18] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The Eucalyptus Open-Source Cloud-Computing System," *Proc. of IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2009.

[19] Open Cloud Computing Interface Working Group. http://www.occi-wg.org/doku.php

[20] OpenNebula: http://opennebula.org/start

[21] S. Patil, G. A. Gibson, G. R. Ganger, J. Lopez, M. Polte, W. Tantisiroj, and L. Xiao , "In Search of an API for Scalable File Systems: Under the Table or Above It?", *Proc. of Usenix HotCloud'09* , 2009.

[22] SAGA: Simple API for Grid Applications, http://saga.cct.lsu.edu/

[23] Rackspace Cloud. http://www.rackspacecloud.com/

[24] SimpleCloud API. http://www.simplecloud.org/

[25] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System." in *Proc. of the Symposium on Operating Systems and Principles*, pages 172-182. ACM Press, 1995.

[26] WindowsAzure4J. http://www.windowsazure4j.org