Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics

Raghu Ramakrishnan*, Baskar Sridharan*,
John R. Douceur*, Pavan Kasturi*, Balaji Krishnamachari-Sampath*, Karthick Krishnamoorthy*,
Peng Li*, Mitica Manu*, Spiro Michaylov*, Rogério Ramos*, Neil Sharman*, Zee Xu*,
Youssef Barakat*, Chris Douglas*, Richard Draves*, Shrikant S Naidu**, Shankar Shastry**,
Atul Sikaria*, Simon Sun*, Ramarathnam Venkatesan*
{raghu, baskars, johndo, pkasturi, balak, karthick, pengli, miticam, spirom, rogerr, neilsha, zeexu, youssefb, cdoug, richdr, shrikan, shanksh, asikaria, sisun, venkie}@microsoft.com

ABSTRACT

Azure Data Lake Store (ADLS) is a fully-managed, elastic, scalable, and secure file system that supports Hadoop distributed file system (HDFS) and Cosmos semantics. It is specifically designed and optimized for a broad spectrum of Big Data analytics that depend on a very high degree of parallel reads and writes, as well as collocation of compute and data for high bandwidth and low-latency access. It brings together key components and features of Microsoft's Cosmos file system—long used internally at Microsoft as the warehouse for data and analytics—and HDFS, and is a unified file storage solution for analytics on Azure. Internal and external workloads run on this unified platform. Distinguishing aspects of ADLS include its support for multiple storage tiers, exabyte scale, and comprehensive security and data sharing. We discuss ADLS architecture, design points, and performance.

Keywords

Storage; HDFS; Hadoop; map-reduce; distributed file system; tiered storage; cloud service; Azure; AWS; GCE; Big Data

1. INTRODUCTION

The Cosmos file system project at Microsoft began in 2006, after GFS [11]. The Scope language [7] is a SQL dialect similar to Hive, with support for parallelized user-code and a generalized group-by feature supporting Map-Reduce. Cosmos and Scope (often referred to jointly as "Cosmos") are operated as a service—users company-wide create files and submit jobs, and the Big Data team operates the clusters that store data and process the jobs. Virtually all groups across the company, including Ad platforms, Bing, Halo, Office, Skype, Windows and XBOX, store many exabytes of heterogeneous data in Cosmos, doing everything from exploratory analysis and stream processing to production workflows.

While Cosmos was becoming a foundational Big Data service within Microsoft, Hadoop emerged meantime as a widely used open-source Big Data system, and the underlying file system HDFS has become a de-facto standard [28]. Indeed, HDInsight is a Microsoft Azure service for creating and using Hadoop clusters.

ADLS is the successor to Cosmos, and we are in the midst of migrating Cosmos data and workloads to it. It unifies the Cosmos and Hadoop ecosystems as an HDFS compatible service that supports both Cosmos and Hadoop workloads with full fidelity.

It is important to note that the current form of the external ADL service may not reflect all the features discussed here since our goal is to discuss the underlying architecture and the requirements that informed key design decisions.

1.1 Learnings, Goals, and Highlights

Our work has been deeply influenced by learnings from operating the Cosmos service (see Section 1.2), and from engaging closely with the Hadoop community. Tiered storage in ADLS (see Section 1.3) grew out of a desire to integrate Azure storage with Cosmos [6] and related exploratory work in CISL and MSR [21], and influenced (and was influenced by) work in Hadoop (e.g., [10][15]). Based on customer feedback, an overarching objective was to design a highly secure (see Section 6) service that would *simplify management of large, widely-shared, and valuable / sensitive data collections*. Specifically, we have sought to provide:

- Tier/location transparency (see Section 1.3)
- Write size transparency (see Section 4.5)
- Isolation of noisy neighbors (see Section 4.7)

This focus on simplicity for users has had a big influence on ADLS. We have also sought to provide support for improvements in the end-to-end user experience, e.g., store support for debugging failed jobs (see Section 4.6), and to simplify operational aspects from our perspective as cloud service providers (see Section 1.2).

The key contributions of ADLS are:

- From an ecosystem and service perspective, ADLS is the successor to the Cosmos service at Microsoft, and complements Azure Data Lake Analytics (ADLA) [1], a YARN-based multi-tenanted environment for Scope and its successor U-SQL [30], as well as Hive, Spark and other Big Data analytic engines that leverage collocation of compute with data. Thus, ADLA and ADLS together unify Cosmos and Hadoop, for both internal and external customers, as Microsoft's warehouse for data and analytics. ADLS is also a very performant HDFS compatible filesystem layer for Hadoop workloads executing in Azure public compute, such

SIGMOD'17, May 14–19, 2017, Chicago, IL, USA.
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00.
DOI: http://dx.doi.org/10.1145/3035918.3056100

as Microsoft's HDInsight service and Azure offerings from vendors such as Cloudera and Hortonworks.

- Technically, ADLS makes significant advances with its modular microservices architecture, scalability, security framework, and extensible tiered storage. The RSL-HK ring infrastructure (Section 3) combines Cosmos's Paxos-based metadata services with SQL Server's columnar processing engine (Hekaton) to provide an extremely scalable and robust foundation, illustrating the value of judiciously combining key ideas from relational databases and distributed systems. The naming service that builds on it provides a flexible yet scalable hierarchical name space. Importantly, ADLS is designed from the ground up to manage data across multiple tiers, to enable users to store their data in any combination of tiers to achieve the best cost-performance trade-off for their workloads. The design is extensible in allowing new storage tiers to be easily added through a *storage provider* abstraction (Section 5).

ADLS is the first public PaaS cloud service that is designed to support full filesystem functionality at extreme scale. The approach we have taken to solve the hard scalability problem for metadata management differs from typical filesystems in its *deep integration of relational database and distributed systems* technologies. Our experience shows the potential (as in [34] and [3]) in judicious use of relational techniques in non-traditional settings such as filesystem internals. See Sections 3 and 4.4 for a discussion.

We now go a little deeper into our experience with Cosmos, and the notion of tiered storage, before presenting ADLS in later sections.

1.2 ADLS Requirements from Cosmos

The scale of Cosmos is very large. The largest Hadoop clusters that we are aware of are about 5K nodes; Cosmos clusters exceed 50K nodes each; individual files can be petabyte-scale, and individual jobs can execute over more than 10K nodes. Every day, we process several hundred petabytes of data, and deliver tens of millions of compute hours to thousands of internal users. Even short outages have significant business implications, and operating the system efficiently and reliably is a major consideration.

Cosmos tends to have very large clusters because teams value sharing data as in any data warehouse, and as new data is derived, more users consume it and in turn produce additional data. This virtuous "information production" cycle eventually leads to our exceeding a cluster's capacity, and we need to migrate one or more teams together with their data (and copies of other datasets they require) to another cluster. This migration is a challenging process that takes several weeks, requires involvement from affected users, and must be done while ensuring that all production jobs continue to execute with the desired SLA. Similarly, such large clusters also cause resource issues such as TCP port exhaustion.

Thus, a key design consideration was to improve ease of operations, including upgrades and transparent re-balancing of user data. At the scale ADLS is designed to operate, this is a big overhead, and lessons learnt from Cosmos informed our design. Specifically, the ADLS naming service provides a hierarchical file and folder namespace that is independent of physical data location, with the ability to rename and move files and folders without moving the data. Further, ADLS is designed as a collection of key microservices (for transaction and data management, block management, etc.) that are also de-coupled from the physical clusters where data is stored. Together, these capabilities significantly improve the ease of operation of ADLS.

Security and access control are paramount. ADLS has been designed from the ground up for security. Data can be secured per-user, per-folder, or per-file to ensure proper confidentiality and sharing. ADLS leverages Azure Active Directory for authentication, providing seamless integration with the Azure cloud and traditional enterprise ecosystems. Data is stored encrypted, with options for customer- and system-owned key management. The modular implementation enables us to leverage a wide range of secure compute alternatives (from enclaves to secure hardware).

Finally, we have incorporated some useful Cosmos features that are not available in other filesystems. Notably, the efficient concatenation of several files is widely used at the end of Scope/U-SQL jobs to return the result as a single file. This contrasts with Hive jobs which return the results as several (usually small) files, and thus incurring overhead for tracking artificially large numbers of files. We describe this concatenation operation in Section 2.3.5.

1.3 Tiered Storage: Motivation and Goals

As the pace at which data is gathered continues to accelerate, thanks to applications such as IoT, it is important to be able to store data inexpensively. At the same time, increased interest in real-time data processing and interactive exploration are driving adoption of faster tiers of storage such as flash and RAM. Current cloud approaches, such as storing data durably in tiers (e.g., blob storage) optimized for inexpensive storage, and requiring users to bring it into more performant tiers (e.g., local HDDs, SSDs or RAM) for computation, suffer from three weaknesses for Big Data analytics:

1. The overhead of moving all data to the compute tier on demand affects performance of analytic jobs.
2. Jobs must be able to quickly determine where data is located to collocate computation. This requires running a file manager that efficiently provides fine-grained location information in the compute tier.
3. Users must explicitly manage data placement, and consider security, access control, and compliance as data travels across distinct services or software boundaries.

We simplify all this by enabling data location across storage tiers to be managed by the system, based on high-level policies that a user can specify (or use system defaults), with security and compliance managed automatically within a single integrated service, i.e., ADLS. The location of data is thus transparent to all parts of the user experience except for cost and performance, which users can balance via high-level policies.

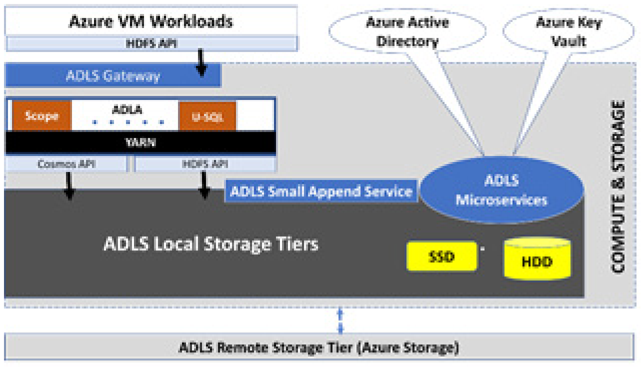


Figure 1-1: ADLS Overview

The architectural context for ADLS is illustrated in Figure 1-1. ADLS and ADLA are designed for workloads such as Apache Hadoop Hive, Spark, and Microsoft's Scope and U-SQL, that optimize for data locality. The YARN-based ADLA framework enables such distributed queries to run in a locality-aware manner on files stored in ADLS, similar to how Scope queries run on Cosmos or Hive queries run on Hadoop clusters with HDFS. The query (specifically, the query's application master or AM) calls ADLS to identify the location of its data and produces a plan that seeks to run tasks close to the data they process, and then calls YARN to get nearby compute resources for each task (i.e., on the same machines where the task's data is in local storage, if possible, or on the same racks). Local storage is durable; users can choose to always keep data there or to use (cheaper) remote storage tiers, based on their anticipated usage patterns. If relevant data is only available on remote storage, ADLS automatically fetches it on-demand into the machine where the task is scheduled for execution. Queries can also execute anywhere on Azure VMs (e.g., in IaaS Hadoop services or Microsoft's managed Hadoop service, HDInsight) and access data in ADLS through a gateway.

The rest of the paper is organized as follows. In Section 2, we present the overall architecture of ADLS. We introduce the different components, and discuss the flow of the main operations supported by ADLS. Then in Section 3 we discuss the technology behind RSL-HK rings, a foundation for many of the metadata services, before diving into the implementation of each of the services in Section 4. We examine tiered storage in more depth with a discussion of storage providers in Section 5. In Section 6, we focus on security in ADLS, including encryption and hierarchical access controls. Since composing several complex microservices in quite complex ways is so central to the architecture of ADLS, its development involved a laser-like focus on ensuring that each of the microservices achieves high availability, strong security**,** low latency, and high throughput. For this reason, we present performance data for the individual microservices throughout this paper. However, a through treatment of end-to-end performance is beyond its scope.

2. OVERVIEW OF ADLS

In this section, we introduce the basic concepts underlying file storage in ADLS, the system components that represent files, folders and permissions, and how information flows between them.

2.1 Anatomy of an ADLS File

An ADLS *file* is referred to by a URL, and is comprised of a sequence of *extents* (units of locality in support of query parallelism, with unique IDs), where an extent is a sequence of *blocks* (units of append atomicity and parallelism) up to 4MB in size. At any given time, the last extent of a file is either *sealed* or *unsealed*, but all others are sealed and will remain sealed; this last extent is the *tail* of the file. Only an unsealed extent may be appended to, and it is available to be read immediately after any write completes. A file is also either sealed or unsealed; *sealed files* are immutable i.e., new extents cannot be appended, and the file size is fixed. When a file is sealed, its length and other properties are made permanent and cannot change. While a file's URL is used to refer to it outside the system, a file has a unique ID that is used by almost all components of ADLS.

The concept of tiered storage is core to the design of ADLS—any part of a file can be in one (or more) of several storage tiers, as dictated by policy or performance goals. In general, the design supports *local tiers*, whose data is distributed across ADLS nodes for easy access during job computation, and *remote tiers*, whose data is stored outside the ADLS cluster. The set of storage tiers currently supported includes Azure Storage [5] as well as local storage in the compute cluster (including local SSD and HDD tiers), and has a modular design that abstracts tiers behind a *storage provider* interface. This interface exposes a small set of operations on ADLS file metadata and data, but not namespace changes, and is invariant across all the types of storage tiers. This abstraction allows us to add new tiers through different *provider implementations* such as Cosmos, HDFS, etc.

Local storage tiers are on the same nodes where ADLA can schedule computation, in contrast to remote tiers, and must provide enough information about extent storage to allow ADLA to optimize computations by placing computation tasks close to the data they either read or write, or both. Remote tiers do not have this responsibility, but they still need to support parallelism, as an ADLA computation tends to read (and often write) many extents simultaneously, and a job executing on thousands of back-end nodes can create a barrage of I/O requests, imposing significant requirements on remote storage tiers.

ADLS supports a concept called a *partial file*, which is essentially a sub-sequence of the file, to enable parts of a file to reside in different storage tiers, each implemented by a *storage provider* (see Section 5). This is an important internal concept, not exposed to users. A partial file is a contiguous sequence of extents, with a unique ID, and a file is represented internally in ADLS as an unordered collection of partial files, possibly overlapping, each mapped to a specific storage tier at any given time. Thus, depending on how partial files are mapped to different storage tiers, a storage tier may contain non-contiguous sub-sequences of a given file's extents. The set of partial files must between them contain all extents of a file, but some extents may be represented in more than one partial file. For each file, at most one partial file contains an unsealed extent, and that is the *tail partial extent* for the file.

Partial files can also be sealed. If for some reason the tail partial file needs to be sealed, a new partial file is created to support subsequent appends. Even after a file is sealed, the location of any of its partial files can still change, and further, the set of underlying partial files can be modified (split, merged) so long as this set continues to accurately reflect the sequence of extents of the file. In Figure 2-1, we show how two example files are represented as partial files through the various ADLS microservices and storage tiers. We will use this as an example to illustrate the ADLS architecture in the next few sections.

2.2 Components of the ADLS System

We first outline the various components of ADLS, illustrated in Figure 2-2, and return to each of the components in detail in subsequent sections. An ADLS cluster consists of three types of nodes: *back-end* nodes, *front-end* nodes and *microservice* nodes. The largest number of servers, the back-end nodes, is reserved for local tier data storage and execution of ADLA jobs. The front-end nodes function as a gateway, hosting services needed for controlling access and routing requests, and microservice nodes host the various microservices.

Central to the ADLS design is the *Secure Store Service* (SSS, see Section 4.1), which orchestrates microservices and a heterogenous set of supported storage providers. The SSS acts as the point of entry into ADLS and provides a security boundary between it and applications. It implements the API end points by orchestrating between metadata services and storage providers, applying light-weight transaction coordination between them when needed, handling failures and timeouts in components by retries and/or aborting client requests as appropriate, and maintaining a consistent internal state throughout and ensuring that a consistent state is always presented to clients. It provides a semantic translation between files exposed by the ADLS system and the underlying storage providers. It also hosts adapter code for implementing the storage provider interface for each supported storage provider.

The *RSL-HK Ring* infrastructure (see Section 3) is the foundation for how ADLS supports very large files and folders, providing efficient, scalable and highly available in-memory, persistent state; most of the metadata services described below are based on it. It implements a novel combination of Paxos and a new transactional in-memory block data management design. The scalability and availability of RSL-HK is based on its ability to dynamically add new Paxos rings, and to add machines to an existing Paxos ring. The Paxos component of RSL-HK is based on the implementation in Cosmos, which has provided very high availability across hundreds of rings in production use over many years. The transactional in-memory block management leverages technology used in SQL Hekaton [9]. The metadata services run as RSL-HK rings, with each ring typically having seven dedicated servers.

A hyper-scale distributed *Naming Service* (NS) (see Section 4.4), layered over RSL-HK rings, associates file names with IDs and provides a hierarchical namespace for files and folders across data centers, supporting renames and moves of files and folders without copying data [18]. In contrast, traditional blob storage systems lack the ability to rename or move a container, and require a recursive copy of contents to a newly created container. NS also enables the implementation of ACLs on both files and folders and, through its integration with Azure Active Directory and the ADLS *Secret Management Service* (SMS) component (see Section 4.2), provides enterprise grade secure access control and sharing. ADLS supports POSIX style ACLs [23], and can support other convenience features such as recursive ACLs. All ADLS namespace entries and operations on them are managed by NS, regardless of the tiers involved in storing the given file or folder (Section 4.4).

The *Partial File Management Service* (PFM) (see Section 4.3) maintains the list of partial files that comprise a file, along with the provider (i.e., storage tier) for each partial file. The implementation of each partial file, including its metadata, is the responsibility of the storage provider to which that partial file is mapped. Accordingly, data and file metadata operations against an ADLS file are partly handled by delegation to the corresponding storage providers. For example, when a file is written to, the SSS uses the file's ID and the PFM to locate the tail partial file that contains the extent, and appends to it.

Depending on usage patterns, policies and the age of data, partial files need to be created on a specific tier, moved between store tiers or deleted altogether. When a file is created, a single partial file is immediately created to represent it. Any append to the file is always to a single partial file on a single tier.

We move partial files between tiers through decoupled copying and deleting. To change the tier of a partial file, a new partial file is created in the target tier, by copying data from the source partial file. For a time, two separate partial files (in two different tiers) are represented in the PFM, containing identical data. Only then is the source partial file deleted. When a partial file is no longer needed, it is deleted, while ensuring that all extents in it also exist in some other partial file, unless the file itself is being deleted.

The *Extent Management Service* (EMS) tracks the location of every extent of every file in a remote storage provider (see Section 5.3), similar to the HDFS NameNode. Scalability of the NameNode has long been a challenge in HDFS, and in contrast the EMS, using the RSL-HK ring infrastructure, achieves very high scalability, performance, and availability. Notably, ADLS differs from HDFS in separating naming (and other file metadata), stored in the NS, from extent metadata, stored in the EMS. This is to handle the disparate scale characteristics and access patterns for these two kinds of metadata.

The *Transient Data Store Service* (TDSS) (see Section 4.6) temporarily stores output from ADLA computation tasks that is yet to be used as input to other such tasks, and makes many optimizations to achieve significant performance improvements.

ADLS is designed to support low-latency append scenarios. Typically, low-latency scenarios involve appends that are small (a few bytes to a few hundred KB). These scenarios are sensitive to the number of operations and latency. The *Small Append Service* (SAS) (see Section 4.5) is designed to support such scenarios without requiring an ADLS client to use different APIs for different append sizes. It enables a single ADLS file to be used for both low-latency, small appends as well as traditional batch system appends of larger sizes that are sensitive to bandwidth. This is made possible by detecting the write sizes in real-time, storing the appends in a temporary (but durable) buffer, and later coalescing them to larger chunks. The small appends are acknowledged immediately and thus the client can immediately read the tail of the file.

2.3 Flow of Major Operations

This section outlines how ADLS microservices interact with storage providers to implement the core operations, including examples drawn from Figure 2-1.

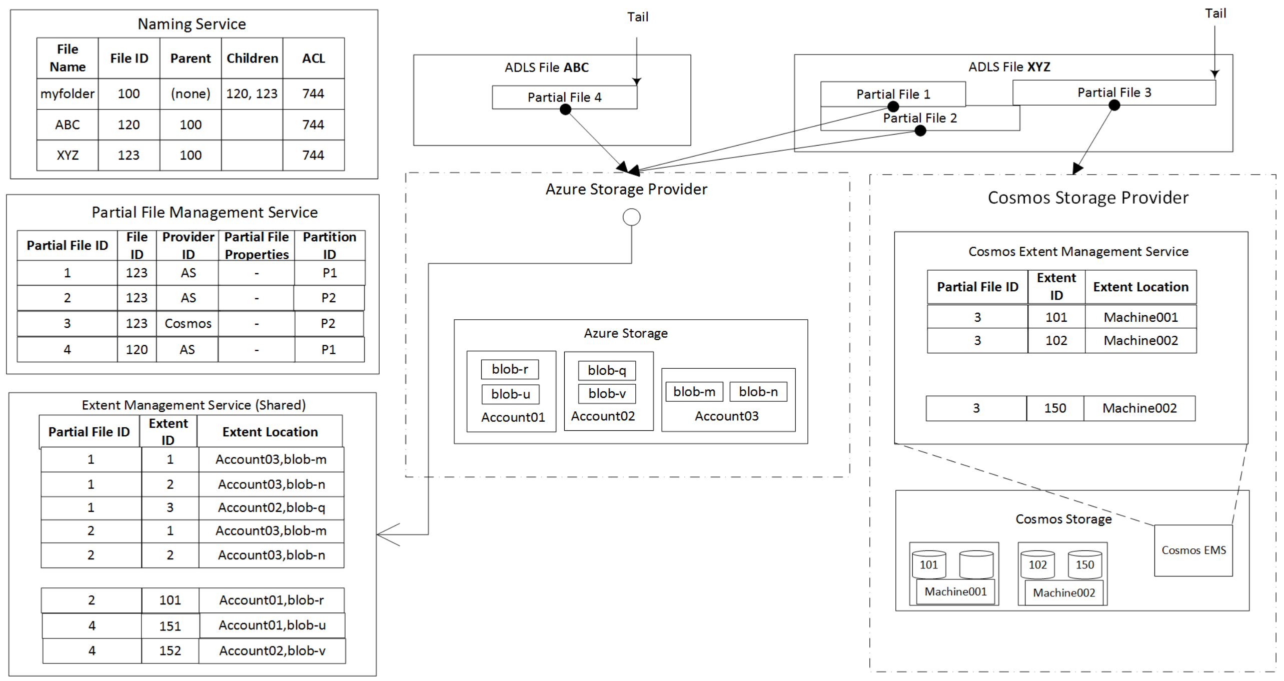
2.3.1 Opening a File

To create a new file, the SSS creates an entry in the NS, and associates a new file ID with the name. The SSS then chooses the storage provider for the tail partial file, and associates the provider and a newly generated partial file ID (as tail partial file) with the file ID in the PFM for use by subsequent operations. Finally, the SSS requests the chosen provider to create a file indexed by the chosen partial file ID. To open an existing file, the flow is similar, but the file ID is looked up in the NS, the provider ID and tail partial file ID are looked up in the PFM. In both cases, access controls are enforced in the NS when the name is resolved.

Using the example from Figure 2-1, consider opening /myfolder/ABC. The entry for "myfolder" is found in the NS, child links are followed to the entry for "ABC", and file ID 120 is obtained. This is looked up in the PFM, to get the provider ID Azure Storage and partial file ID 4. These are associated with the returned file handle and stored in the SSS for later use.

2.3.2 Appending to a File

ADLS supports two different append semantics: fixed-offset append, and free offset append. In the case of fixed-offset append, the caller specifies the starting offset for the data and if that offset already has data, the append is rejected. In the case of free offset append, ADLS determines the starting offset for the data and hence the append operation is guaranteed not to fail due to collisions at the offset. One crucial append scenario is upload of large files, typically performed in parallel. When the order of data within the file is not critical and duplicates (due to timeout or job failure) can be tolerated, multiple threads or clients can use concurrent, free-offset appends to a single file. (A similar approach to parallel appends was used in Sailfish [29], with the added generality of writing in parallel to extents other than the tail.) Otherwise, all clients can use fixed-offset uploads to their own intermediate files and then use the fast, atomic concatenation operation described in Section 2.3.5 to concatenate them in order into a single file.


Naming Service

File Name	File ID	Parent	Children	ACL
myfolder	100	(none)	120, 123	744
ABC	120	100		744
XYZ	123	100		744

Partial File Management Service

Partial File ID	File ID	Provider ID	Partial File Properties	Partition ID
1	123	AS	-	P1
2	123	AS	-	P2
3	123	Cosmos	-	P2
4	120	AS	-	P1

Extent Management Service (Shared)

Partial File ID	Extent ID	Extent Location
1	1	Account03,blob-m
1	2	Account03,blob-n
1	3	Account02,blob-q
2	1	Account03,blob-m
2	2	Account03,blob-n
2	101	Account01,blob-r
4	151	Account01,blob-u
4	152	Account02,blob-v

Cosmos Extent Management Service

Partial File ID	Extent ID	Extent Location
3	101	Machine001
3	102	Machine002
3	150	Machine002

Figure 2-1: Anatomy of a File

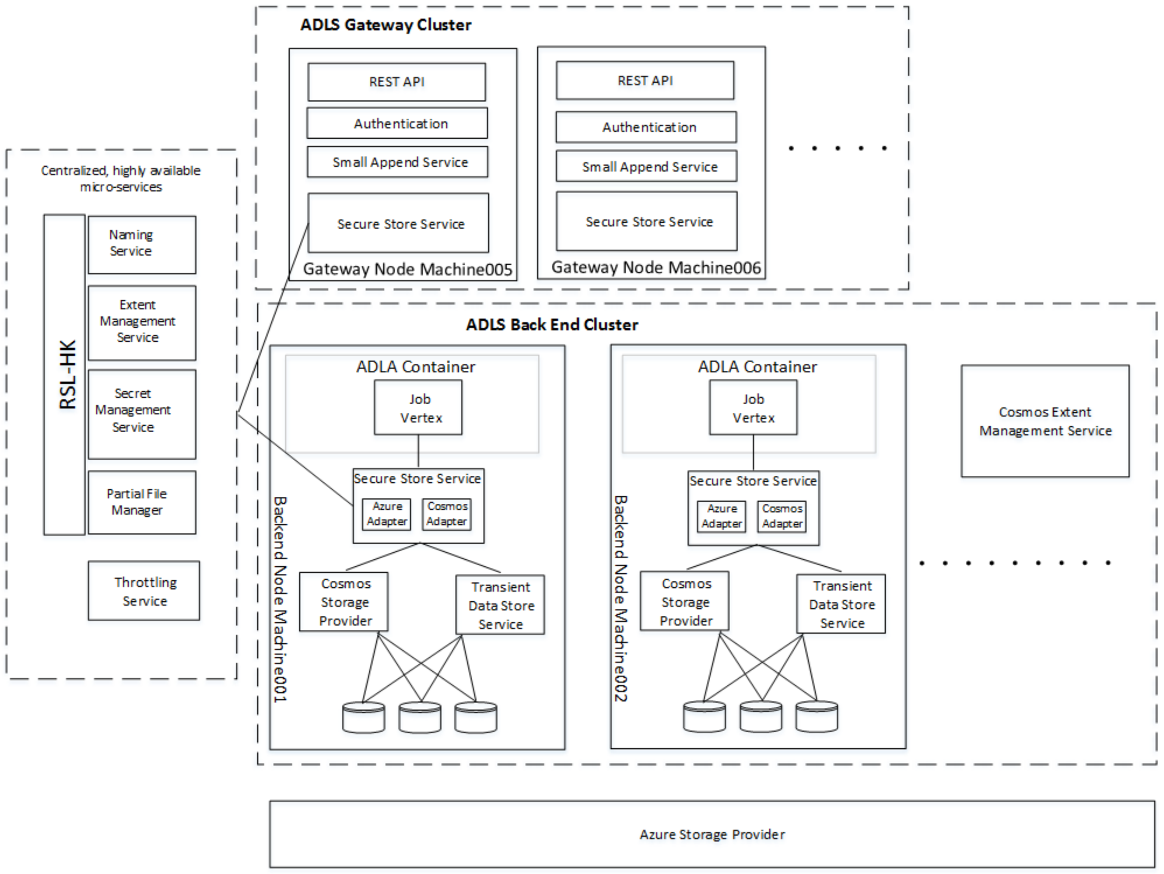


Figure 2-2: ADLS Architecture

To append to an open file, SSS looks up the tail partial file from the PFM. Note that the tail partial file is guaranteed to be unique for any particular file. If there is no tail partial file in PFM, this means the file is either empty or was sealed for some reason.

In that case, SSS registers a new partial file in PFM, and creates it in the appropriate storage provider, and opens a handle to this new tail partial file. *Once SSS has a valid tail partial file handle, all subsequent appends go directly to the storage provider.* Neither the NS nor the PFM is involved. SSS only needs to interact with PFM again when the current tail partial file is sealed.

In Figure 2-1, consider appending to /myfolder/XYZ. The file ID in the NS is 123, and according to the PFM the tail partial file is 3, mapped to the Cosmos provider. The consequence of not involving PFM in every append is that PFM does not have the up-to-date information for the tail partial file. Before the tail partial file has been sealed, requests for the length are delegated to the provider. When a partial file (and hence also its tail extent) is sealed, the length is updated in the PFM. If an append operation on a tail partial file exceeds the storage provider's file length limit, the operation fails (locally), causing the SSS to initiate sealing the partial file, and retry the append. Notably, sealing a partial file is idempotent (a requirement on the provider implementation), and hence if concurrent appends experience this local failure, they are all allowed to succeed, even though only the one that won the race to seal actually made a change.

2.3.3 Read

There are two methods to read data from ADLS. The first method is by specifying a byte range to read. Data in the byte range can be available in multiple storage providers, e.g. in both Cosmos provider and Azure Storage provider. SSS attaches a numerical value to each storage provider to indicate its performance relative to the other stores. If the data is available in multiple storage providers, SSS selects the "fastest" storage provider to read from. Once SSS determines which partial file to read from, it sends the read request to the partial file without PFM being involved.

In Figure 2-1, consider reading bytes 0 to 9 from the file /myfolder/XYZ, and notice that partial files 1 and 2 both start at the beginning of the file. Assume both are long enough to contain these 10 bytes. The SSS has a choice of which storage provider to read from, based on policies. If it chooses file 1, the provider looks up this partial file in the EMS to find that extent 1 is in blob-m of storage account Account03. The internal APIs of Azure Storage are then used to read the required bytes from that blob.

The second method is by accessing extents and their replicas directly from the local storage providers when compute and data are collocated. First, the GetFileInformation entry point (described below) is used to obtain the location of all extents of a file, including their replicas, across all storage providers. Then the job chooses the extents it needs and the instances it prefers because of cost, performance or availability. Computation tasks are collocated with the extents that are read, and they access them via the store provider interface through the SSS.

2.3.4 Obtaining File Metadata

The GetFileInformation operation returns file meta data and a sequence of extent meta data, primarily for use by ADLA in choosing back-end nodes for placing a job's computation tasks. File and extent meta data are split between PFM and storage providers. This is implemented by looking up the file in the PFM, and then traversing all the partial file references, following them to the storage providers to obtain extent metadata including replicas, and then combining it all in the result. The extent metadata returned is organized in terms of the user-visible extents: there is one object returned for each of these extents, which contains the metadata about extent instances in the storage providers, including their replicas. (Replica details are only of interest for local tiers—for remote tiers they are merely a detail of provider implementation.) The metadata returned includes immutable information such as extent ID, length, last modified time and CRC.

In Figure 2-1, consider the file /myfolder/XYZ. For a call to GetFileInformation(), the SSS first contacts the PFM to obtain the partial files: 1, 2 and 3. Since the first two are mapped to the Azure Storage provider, the SSS contacts the EMS to obtain metadata for extents 1 to 101, stored there and it contacts the Cosmos provider to obtain metadata for extents 101 to 150. The SSS then merges this information, returns the metadata for all 151 instances of the 150 logical extents, representing the fact that extent 101 has two instances, one in each of the providers. Thus, a total of 150 metadata records will be returned, one for each extent. The record for extent 101 will be more complex because it contains both information about all replicas of the extent in the Cosmos tier, and also the account and blob mapping for the Azure Storage tier.

Because extent metadata is exposed outside of ADLS, it must remain the same regardless of the storage provider where the extent resides at a given point in time, and in practice this means that a particular extent must always have the same ID, even if it resides in more than one provider.

2.3.5 Concatenation

This concatenates a set of source files to produce a single target file. After the concatenation, the source files are effectively deleted. All source files and partial files in each source file must have been sealed before concatenation, and this operation will seal them if necessary. This property allows the size of all partial files to be known, and makes it safe to combine, and if necessary re-sequence, the partial file metadata into a single file.

This is implemented purely as a metadata operation in the NS and PFM, except that sealing a partial file, if necessary, requires the involvement of the storage provider.

Returning to the example of Figure 2-1, consider concatenating /myfolder/ABC and /myfolder/XYZ to produce /myfolder/NEWFILE. Only the contents of the NS and the PFM will change. First, a new file ID is assigned—say 130. Then, the PFM is consulted to determine the set of partial file IDs that will comprise the new file—1, 2, 3, 4. All four corresponding entries in the PFM atomically have their file ID updated to 130 (from variously 120 and 123.) Finally, the file entries ABC and XYZ in NS are atomically replaced with the entry NEWFILE, and the entry for myfolder is updated to have only NEWFILE as a child.

2.3.6 Enumeration

File enumeration (within a given folder) is mostly a metadata operation in the NS except that it also returns the file length and the last modified time. PFM contains the up-to-date length of all sealed partial files. If the tail partial file is unsealed, SSS queries the owning storage provider for the tail partial file. File enumeration requires the appropriate permissions to access the entire path and the entries to be returned, and these are enforced by the NS.

3. RSL-HK RINGS

ADLS metadata services need to be scalable, highly available, have low-latency and high throughput, and be strongly consistent. The set of metadata services in Cosmos, while less extensive, have similar requirements, and we have long used the Replicated State Machine approach [2], based on a proprietary Replicated State

Library (RSL) that implements the complex underlying replication, consensus, checkpointing and recovery mechanisms. RSL implements Viewstamped Replication [22][31], with consensus based on Paxos [19][20] and improvements described in [16].

RSL-based services are deployed in quorum-based rings, usually consisting of seven servers, appropriately distributed across failure domains. There is always a single primary node, which services all updates, and a new primary is elected if the primary node becomes unavailable. While RSL is a useful building block, any internal state that needs to be persistent requires custom (and complex) code to efficiently leverage the state management features of RSL, and this has been a source of many issues over the years in Cosmos.

In developing ADLS, we sought to improve RSL by creating a more declarative foundation for developers to do state management, based on in-memory tables in the SQL Server Hekaton Engine. The new service, called *RSL-HK Rings*, maintains the underlying Paxos ring (including electing a new primary as needed, keeping secondaries up-to-date, and hydrating newly deployed secondaries). It makes it significantly easier to develop metadata services by providing the following functionality:

- *Management of the metadata service's persistent state:* State is maintained as replicated in-memory Hekaton tables and indexes, and RSL-HK executes updates using Hekaton.
- *RPC handling and dispatch to callback functions*: Metadata operations are written as transactions with ACID semantics realized via optimistic, lock-free techniques in Hekaton.
- *Fault-tolerance of state changes*, *including checkpointing, logging and recovery*: This includes writing the transaction log for the primary, and replicating it to secondaries using RSL; full or incremental backups of log and checkpoint files, which serves as the basis for cross-datacenter replication to additional RSL-HK rings; and recovery from local checkpoint files and logs when a service instance restarts.

The service developer is responsible for *defining the structure of the service's state,* by defining the schema of the Hekaton tables, and specifying desired indexes on them for performance. The service developer also *defines the RPC callback functions for service operations* as transactions on the Hekaton tables. This includes ensuring that the semantics of each operation is appropriately handled (e.g., updates are only allowed on the primary, and reads are appropriate on secondaries only if the metadata service can tolerate slightly stale data, given the asynchronous replication of RSL-HK). Further, the developer must exercise some care to obtain the best performance, e.g., making operations non-blocking, and avoiding excessive contention for individual table rows.

In summary, RSL-HK leverages Hekaton to give service developers a more declarative way to manage state, and transparently provides replication and recovery in an application-agnostic way, thereby freeing service developers to concentrate solely on the service's external interface and its logic.

Figure 3-1 shows the architecture of a RSL-HK based service in a reduced three-node configuration, indicating the flow of transaction log replication from the primary node to the secondary nodes, and the fact that each secondary produces its own checkpoints based on the replicated transaction flow.

The NS performance chart shown in Section 4.4.3 is a good indication of RSL-HK's performance. While NS implements non-trivial service logic over RSL-HK, its performance is dominated by that of RSL-HK. The results show that (a) RSL-HK achieves high throughput, and (b) latencies are less than 1ms for read transactions and less than 10ms for write transactions, and they do not increase with throughput until CPU saturation.

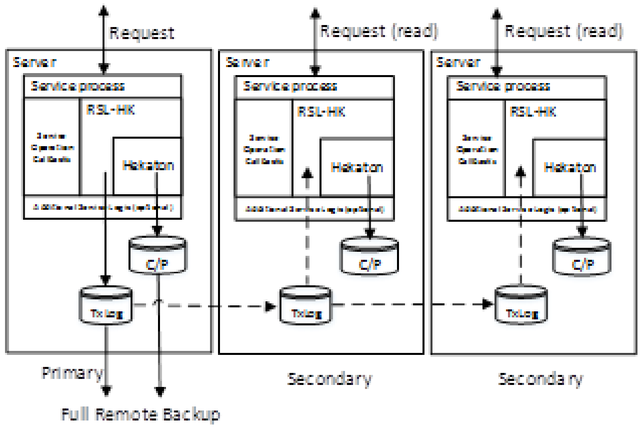


Figure 3-1: Architecture of an RSL-HK based service

4. ADLS COMPONENTS

Next, we will describe the architecture and behavior of the individual services that comprise ADLS.

4.1 Secure Store Service (SSS)

The SSS provides the API entry points into ADLS, creates the security boundary for ADLS, and orchestrates between the microservices and storage providers.

The SSS communicates with the NS for names of user-visible objects such as ADLS accounts, folders, files and the access control lists for these objects. It implements authorization in conjunction with the NS.

The SSS supports file operations on persistent data and transient data. Data is compressed and encrypted during append operations and decrypted and decompressed during read operations, using the Secret Management Service. As part of its orchestration and transaction coordination roles, the SSS handles failures and timeouts in components by retries and/or aborting client requests as appropriate, maintaining a consistent internal state.

The adapter component of each storage provider is hosted in the SSS, which uses the storage provider ID associated with each partial file to route operations to the appropriate provider via the adapter.

SSS provides a secure endpoint with limited functionality to untrusted code running in a sandbox, such as an ADLA query, and trusted endpoints for services internal to Azure Data Lake (ADL), providing additional functionality. Each request comes with a security token that has enough information to identify the caller and do authorization checks as needed to complete the request.

Operations at the SSS are metered and used for billing, diagnosis, service health monitoring etc. Logs are generated for auditing purposes and exposed to end users. SSS enforces bandwidth and throughput limits at ADLS account granularity and at each ADLS node. A garbage collection service cleans up deleted files and folders and expired files.

4.2 Secret Management Service (SMS)

In ADLS, a large number of *secrets* need to be handled with high throughput and low-latency. This includes internal secrets needed for the functioning of the other microservices described here as well as customer managed secrets (such as keys to access Azure Storage for the accounts they will associate with their ADLS accounts).

The Secret Management Service (SMS) has two related roles: (a) a secret repository, in which secrets can be stored indexed by a tuple <scope, name>, and (b), a broker for interaction with external secret repositories, such as Azure Key Vault (AKV), to enable the storage of secrets for which access is dependent on the SMS' access to a master key present in the external secret repository.

In the first model, the client (internal service or the external user) needs to have the tuple values to use them as a primary key, and the identity of the client needs to satisfy the authorization policies associated with the tuples.

The second model extends the first model by allowing the secret identified by the tuple <scope, name> to be encrypted using a key identified by a tuple <KeyName, VaultName>. To access the secret, the client needs to have the tuple values, and the identity of the client needs to satisfy both the authorization policies associated with the tuples in the SMS as well as in the AKV identified by the second tuple (<KeyName, VaultName>). In both cases, only the security principals satisfying these requirements can access the secrets—for example, administrators by default do not have the rights to access secrets.

In addition to secret management, SMS also handles compression, decompression, encryption, and decryption. The Trusted Software Module (TSM) component in SMS handles these data transformations. TSM is a library that is often hosted in a free-standing process or microservice. Such processes sit behind a dedicated proxy process that handles both communication with its paired TSM instance and the proxy processes paired with other TSM instances.

The security functionality of ADLS is available on every back-end, front-end and microservice node.

The SMS includes a Secret Store back-end, which stores the two kinds of secret tuples described above, in its own RSL-HK rings. Its front-end orchestrates requests from various TSM instances (for store operations) and the ADLS gateway (for user administration of accounts and secrets) and depends on the other SMS microservices. There is a local TSM instance for key management and its proxy. Finally, there is a cache of secret tuples from the Secret Store back-end.

Figure 4-1 shows the components in SMS.

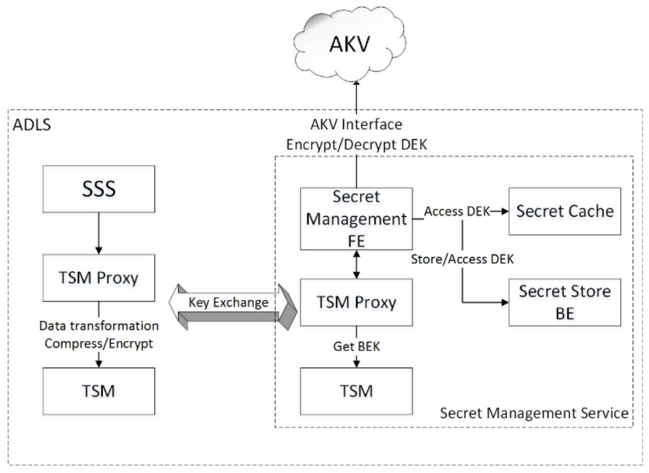


Figure 4-1: Architecture of Secret Management Service

This architecture has three main advantages: (a) it can incorporate external secret repositories for interoperability with other applications while limiting the amount of traffic sent to any such store, to avoid burdening them with the key retrieval scalability needs of ADLS, (b) for the very large files that ADLS supports, it provides scalable encryption and decryption at a block level, by allowing it to take place on the back-end nodes where computation takes place, and (c) the encryption and decryption code can be separated out into a secure process, to gain additional protection.

4.3 Partial File Manager (PFM)

The Partial File Manager (PFM) maps each file ID to the set of one or more partial files (and the IDs of their providers) that represent the file. It is the source of truth for all files in ADLS irrespective of which provider(s) are used to store them. However, the PFM does not understand or represent the internal structure of the partial files, including the extents. This functionality is delegated to the specific storage provider.

The PFM is the source of truth for following metadata:

1. File to partial file mapping
2. Start and end offset of each partial file, except the tail
3. The storage tier in which the partial file resides
4. Partial file creation and modification date/time
5. File length
6. File and partial file seal state

The current length of an unsealed partial file may not be known to PFM. It is only when a file is sealed, allowing all partial files to be sealed, including the one representing the tail, that the PFM can provide the up to date length of the file. To provide interim length as a file is being appended to, it is necessary to periodically ensure that all the other (non-tail) partial files are sealed, and thus able to provide lengths.

The PFM enforces the requirement that all extents of a file are represented by at least one partial file.

The PFM is partitioned for scalability, and it adopts the convention that all partial files for a given file have their metadata represented in the same partition. Partitions can be added dynamically.

4.4 Naming Service (NS)

At the core of ADLS file management is a scalable, strongly consistent, highly-available, high-performance, hierarchical naming service that maps mutable hierarchical paths to fixed references to objects in other ADLS metadata services. It supports full hierarchical naming semantics, including renaming elements of paths, moving subtrees within the namespace, and paged enumeration of folders. It also supports POSIX-style access control, with both classic permissions (owner, group, other) and extended ACLs (access and default).

The service consists of a bank of soft-state front-end proxies and a set of servers. The proxies route commands to servers, coordinates execution of multi-server commands (e.g. move), and responds to changes in namespace partitioning. The server layer persistently stores namespace state, executes commands, partitions the namespace among the set of servers, repartitions the namespace as needed, and informs proxies of changes in namespace partitioning.

4.4.1 Naming Service Server Architecture

The NS server executes on RSL-HK's low-level (non-relational) database interface to application code. It uses tables in this database to store namespace data. Some information, such as ACLs, tends to be widely duplicated among namespace entries, so it is stored in normalized form, i.e., in auxiliary tables with a level of indirection that allows multiple entries to refer to the same set of auxiliary rows, allowing deduplication. This is highly effective in practice, and a good illustration of the flexibility offered by relational abstractions in optimizing internal system designs: ACL tables are orders of magnitude smaller than the main entry tables.

Not all information, however, is stored in its most straightforward normalized representation. RSL-HK uses Hekaton's multi-version concurrency control (MVCC), whose lightning-fast performance is compromised by transactional conflicts that can arise when data is fully normalized. For example, POSIX-compliance requires recording the time at which a folder's contents were last modified. Two concurrent changes to the folder's contents will both try to modify the last-modified time and will transactionally conflict if this requires modifying the same row in a table. So, last-modified time is stored de-normalized in a table that is appended by each update, and a run-behind thread aggregates this information in the background. This table is indexed by row-insertion time, so an efficient lookup can determine the most up-to-date value.

The NS provides strong consistency in its external semantics, both because this is what users and applications expect from a file system and also because strong consistency is often needed to maintain namespace integrity. (For instance, if two concurrent Move operations use snapshot isolation for their reads, the result can be an orphaned loop in the namespace.) Therefore, our default policy is that all write operations use serializable isolation and all read operations use snapshot isolation. However, because serializable isolation can lead to a high rate of transactional conflicts, we carefully reduce the isolation level (at the cost of some software complexity) when this can be done without compromising external semantics or namespace integrity.

As an example, because ACLs are de-duplicated, one ACL may be referred to by multiple namespace entries. When an entry is deleted, its associated ACL rows should be removed only if no other entries refer to the ACL. Using reference counting for this would require a serializable update on every create and delete, which could cause a high rate of transactional conflicts for popular ACLs. So, instead, we have an index on the ACL ID in the entry table, and at the time of deletion, we check whether at least two entries refer to this ACL ID. This mostly works well; however, we found that under certain workload patterns, this check leads to a high rate of transactional conflicts. To minimize such conflicts, this check is performed optimistically using snapshot isolation and then confirmed using serializable isolation. It is very common for the first check to succeed, allowing us to avoid the serializable check. This is semantically safe because we remove the ACL only if we serializably confirm that it is not needed. However, when the snapshot check fails, we might fail to remove an ACL that is no longer referred to, which is a resource leak. A background process cleans these up.

4.4.2 Partitioning and Relocation

For scalability, the namespace is partitioned among multiple RSL-HK rings, with each ring having "custody" over one or more regions of the namespace. A custody region is specified as a path terminating in a half-open interval of names, avoiding a hard limit on folder size. For example, the region /hello/world/[bar,foo) includes the paths /hello/world/bar, /hello/world/cat, and /hello/world/doggie/dig/bone, but not /hello/world/foo. This is a strict generalization of Sprite's prefix tables [33]. Each server records its own custody ranges and those it has relocated to other server rings.

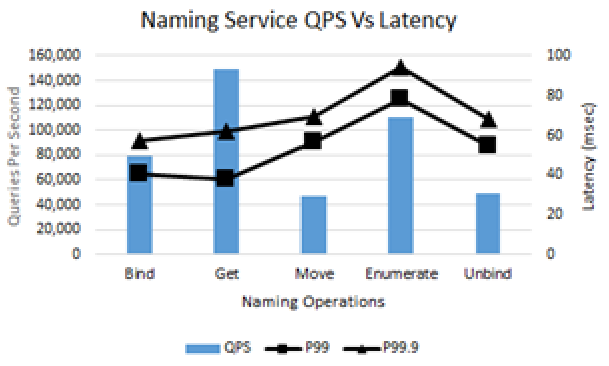


Figure 4-2: Performance of key Naming Service Operations

Each proxy maintains a non-authoritative cache of server custody ranges. Because this cache can be incomplete or stale, a proxy might forward a command to an inappropriate server, in which case the server will respond with an internal error code and any relevant custody information. The proxy then updates its cache and retries the request.

4.4.3 Performance

We evaluate the performance of the NS one operation at a time [17][27], driving that operation towards 100% CPU usage on a single ring. Figure 4-2 shows that high throughputs, over 140,000 QPS, are attained for read operations like Get (name resolution) and Enumerate (paging through entries in a folder). The most complex operation, moving a folder to a point higher in the hierarchy, achieves well over 40,000 QPS. All are achieved with 99.9% latencies below 100ms. In production conditions, the average load on the rings are managed such that the read latencies around 1ms and the write latencies are around 10ms.

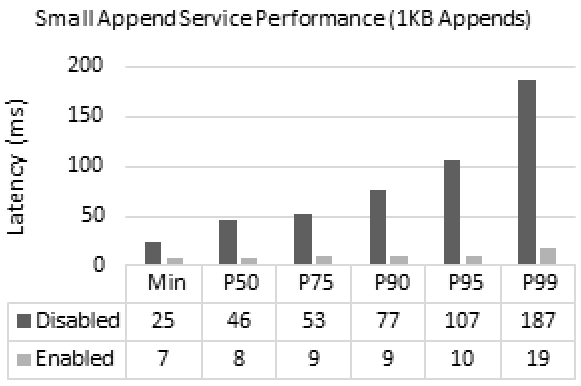
The computers used for the performance tests mentioned in this section and others are commodity servers with 2 x 12 core Intel® Haswell CPUs running at 2.40 GHz, 40 Gb NIC, 4 x 2 TB HDDs and 4 x 500 GB SSDs.

4.5 Small Append Service (SAS)

ADLS provides both high throughput for batch workloads and low-latency for small appends. ADLS supports low-latency small appends, a weakness of HDFS, via an ADLS microservice called the *Small Append Service* (SAS). It does so transparently to the application based on file write patterns—the application uses a single API, and requests are routed appropriately behind it. The Figure 4-3 shows that SAS not only decreases latencies across the board, but it decreases the worst-case latencies by a wide margin. These results are based on YCSB simulations [8].

Low-latency, small appends are critical for transactional systems such as HBase [4][13] on top of ADLS. HBase Write-Ahead Logging append latency directly impacts HBase transaction rate. Figure 4 3 shows the append latencies with and without SAS enabled.

SAS tracks the history of append patterns at a per-file level. If it determines that a file's append pattern is dominated by small appends, it automatically switches those appends to the *small append path,* which uses low-latency storage to store the payload. Appends on the small append path are durable once acknowledged, like all other appends. For a file whose append pattern is dominated by large appends, SAS passes each append to the downstream storage providers directly. This happens without user intervention.



	Min	P50	P75	P90	P95	P99
■ Disabled	25	46	53	77	107	187
■ Enabled	7	8	9	9	10	19

Figure 4-3: SAS Performance

To minimize metadata overheads and improve read performance, appends on the small append path are coalesced asynchronously into larger append blocks before being moved to one of the storage tiers. When a file's appends are being processed by the small append path, only a short section of the tail remains in low-latency storage—the remainder is periodically flushed to a tier. After a period of no activity on a file, the remainder is moved to a storage tier, and the file is forgotten by the SAS. If it is subsequently appended again, its evaluation for the small append path starts again from scratch.

4.6 Transient Data Store Service (TDSS)

During the execution of ADLA jobs, the amount of temporary data generated by operations such as shuffle (and used later in the job) can be large. The *Transient Data Store Service* (TDSS) is responsible for storing such intermediate job data, and runs on every back-end storage/computation node. The temporary data generated by a computation is stored on the machine where it is generated. In general, it is subsequently read from a different machine, but the execution is optimized to collocate reads to the extent possible (e.g., schedule the consumption to be on the same machine or rack).

Unlike most data persisted in ADLS, such intermediate job data can always be regenerated by partially re-running a job, and it is only needed for a short period of time since it is only consumed within the same job. This argues for a design that optimizes for higher performance and lower cost at the cost of occasional re-generation:

1. Only one copy of the data is required. If that copy is lost, the computation that generated it can be re-run.
2. Data can be simply persisted to the filesystem cache.
3. Jobs don't interact with each other, so the security of transient data can be job-centric rather than user or account-centric.

As a practical matter, we note that to support the U-SQL debugging features it is necessary to keep transient data around after the job completes. For jobs that succeed, the transient data is only retained for a few extra hours. For jobs that fail, the transient data is retained for several days. From our Cosmos experience, we have found such support for debugging to be a valuable feature.

These optimizations mean that all appends are to the filesystem cache, but reads (many from remote machines) don't always come from the disk cache. The net effect is that appends are significantly faster than reads, the reverse of a typical store.

Figure 4-4 shows the performance for reading and writing of 4MB blocks on local HDD and SSD on a single machine, for the optimal throughput/latency tradeoff. Four drives of each type were striped for this test and the TDSS client was configured to use 10 threads each

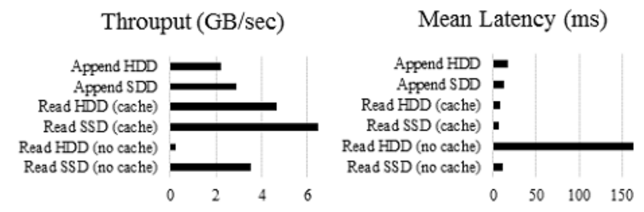


Figure 4-4: TDSS Performance

doing synchronous requests. It shows the importance of caching by comparing results with and without the cached reads.

TDSS maintains an expiration time for each job's temporary data to help facilitate debugging. As with persistent data, all temporary data is encrypted at rest, and all communication is across secure channels.

4.7 Throttling Service (TS)

Each account has an individual quota for bandwidth and number of operations (both read and write) associated with it. The TS is responsible for collating information from all instances of SSS and identifying which accounts have exceeded their quota. Every 50 ms, each SSS sends to the TS the latest per-account request counts and byte counts observed during that period. The TS is shared for scalability and aggregates the data from all SSS instances, and identifies a list of those accounts that should be throttled and returns this list to the SSS. The SSS uses this list to block requests for the indicated accounts for the next 50ms period. The global knowledge provided by the TS allows the entire cluster to react to changing per account activity with a lag of no more than 50ms. The TS and SSS together prevent accounts from impacting each other, thereby alleviating the noisy neighbor problem.

5. STORAGE PROVIDERS

A *storage provider* is essentially a flat-namespace, append-only object store, where each object is a partial file. Storage providers are not required to implement any hierarchical namespace or map any part of ADLS namespace.

Storage providers don't communicate directly with each other or with the microservices. As mentioned previously, the SSS coordinates between microservices and storage providers. This design pattern leverages existing and future ADLS features to accrue to the storage providers. Authentication, authorization, encryption, hierarchical namespace management, etc. are examples of such features.

Furthermore, the addition of storage providers cannot weaken ADLS' data durability and availability. That does not mean data in individual storage providers must be highly durable and available. It means the combination of all copies in all storage providers needs to be durable and available. Each storage provider may have different levels (or probability) of data durability and availability. For example, typically three copies of an extent are maintained in the Cosmos Storage Provider to provide durability.

A storage provider may be built specifically to implement an ADLS tier, or it may have been designed and implemented independently [5][32]. Either way, its implementation consists of some distributed storage system, together with a *storage adapter* that implements the simple, low-level *storage provider interface.* The storage adapter runs as part of the SSS. It contains all logic needed to translate between the storage provider interface and the underlying storage system. (This can vary considerably in complexity, depending on the size of the semantic gap.) It is also responsible for reliable, efficient communication with the underlying storage system.

5.1 Local Storage Providers

Local storage providers store their extents distributed across the ADLS nodes so that they can be accessed quickly by computation tasks executing on these nodes. Of course, the distribution of extents must also meet the needs of scalable and durable distributed storage, with replicas distributed across failure domains. There can be more than one local provider, typically utilizing different classes of storage hardware on the computation nodes.

The Cosmos Storage Provider is a local storage provider based on Cosmos which is designed to support high performance, massive scale analytics by storing data on computation nodes. Its implementation is beyond the scope of this paper, but at a high level it consists of a single, partitioned RSL-based service that fills both the roles of naming and extent management, and an extent storage service that resides on every computation node and is responsible for the extents that are stored on that node. Ideally, computation tasks rely mostly on extents stored on the node on which they execute, but the extent storage service also supports remote read and append requests.

5.2 Remote Storage Providers

The Azure Storage Provider is remote in the sense that the data is stored in one or more separate Azure Storage clusters. The specific location of ADLS extents in Azure Storage is of no interest to ADLS, but high bandwidth, low-latency access to them is of great interest. Because of bandwidth constraints imposed on Azure Storage accounts, an ADLS cluster uses multiple accounts, and each ADLS partial file is separately striped across these accounts, with each ADLS extent represented by a single Azure Storage blob. Extent IDs are hashed to determine the Azure Storage account in which they are stored. This means that each Azure Storage account contains extents from many different ADLS partial files, from many ADLS files, belonging to many ADLS accounts. This is illustrated in Figure 2-1.

5.3 Extent Management Service (EMS)

To incorporate a remote storage provider into ADLS, in many cases it is necessary to introduce extent management—a way to map from a partial file to the extents that comprise it, and how those extents are represented in the underlying storage provider. For this, we have developed a generic EMS. Specifically, the EMS is used to provide extent management for data stored in the Azure Storage provider. Given a partial file and offset, the EMS resolves them to an offset within an extent. It is the source of truth for the following extent metadata:

1. Logical size of the extent (while physical size is managed by Azure Storage)
2. Index of an extent for a given partial file
3. Offset range that an extent covers for a given partial file
4. Location of the extent in the storage tier

Consequently, the EMS is critical for executing read and append operations, translating logical offsets in partial files to physical extent locations in the data node.

The EMS also enforces the pre-defined extent size limit, by sealing the current extent and starting a new one when necessary. It also ensures that a block does not cross an extent boundary. Furthermore, it deals with rejecting fixed offset appends when the specified offset does not match the end of the file.

The EMS is built on top of the RSL-HK infrastructure, but the scale of ADLS requires several rings to represent all extents. To achieve this the EMS supports extent range partitioning. It groups sequences of consecutive extents of a partial file into an *extent range*, which is the unit of partitioning for this service, and is not exposed outside this service. The number of extents in an extent range is limited by the available memory in a partition. Every partial file has a root EMS partition which stores the sequence of partition IDs and extent ranges. To support dynamic rebalancing of partitions (to deal with request hotspots), an extent range can be moved from one partition to another without downtime. When a partial file is deleted, the EMS retains its extent metadata for a configured time to support recovery. A garbage collector within the EMS handles the eventual deletes, together with compaction to reduce memory usage.

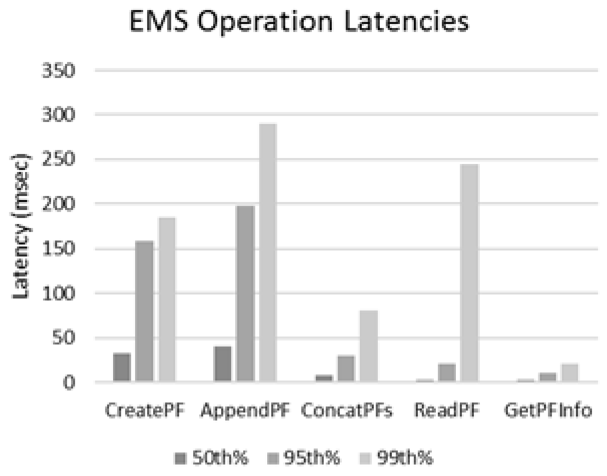


Figure 5-1: Performance of key EMS operations

We have evaluated the performance of EMS by stressing five rings to 90% CPU utilization for thousands of hours from over 100 clients. Each ring experienced 150,000 QPS as reads and 50,000 QPS as writes. The median, 99% and 99.9% latencies are shown in Figure 5-1. In production conditions, the average load on the rings are managed to keep the read latencies under 1ms and the write latencies under 10ms.

6. SECURITY

ADLS is architected for security, encryption, and regulatory compliance to be enforced at scale. Key security requirements such as user authentication, including multi-factor, and Role-Based Access Control (RBAC) are performed through integration with Azure Active Directory (AAD) as the identity provider. ACLs are enforced using POSIX-compliant permissions [12][24] on files and directories using the NS. As the ADLS APIs are HDFS-compliant, they allow ACL data in a manner respecting the POSIX standard [14]. Data flowing into and through ADLS is encrypted in transit and at rest. In ADLS, overall security responsibility rests with the SMS, described earlier. Next, we describe in detail the enforcement of security and encryption, in terms of the SSS and other components.

6.1 Authentication and Authorization

Every ADLS API requires a valid OAuth token from a supported identity provider. AAD allows ADLS to support users and service principals from managed tenants (customers' hosted Active Directory instance), federated tenants (on premise Active Directory instances) and Microsoft accounts. When an ADLS API call is made and an OAuth token is not provided, one is acquired by redirecting the call to AAD. Next, the token is augmented with the user's security groups, and this information is passed through all the ADLS microservices to represent user identity.

Authorization for ADLS operations is a two-step process. First, Azure RBAC is used to check the user's broad permissions: an account owner has rights that override the individual storage entities ACLs. Second, additional checks are made against the ACLs stored in the NS during name resolution, etc. Each entry in the NS contains a set of Access Control Entries (ACEs) that describe the scope, qualifiers, and which of read, write, and/or execute permissions are applicable. The SSS orchestrates with the NS to execute the second step. Authorization is invariant with respect to the entry point for the operation (portal, application built in top of one of our SDKs, REST APIs, etc.) A security audit trail is provided at multiple granularities for all operations.

6.2 Encryption

ADLS provides encryption at rest for all data. *Each append block is encrypted separately, using a unique key, ensuring that the amount of cypher text produced using any given key is small.* The emphasis that ADLS places on extremely large files makes this especially important. The header for every block contains metadata to allow block-level integrity checks and algorithm identification.

Both service-managed and user-managed keys are supported by integrating with AKV for enterprise-grade key management. Integration with AKV and isolation of encryption and decryption are performed by the SMS. Various encryption and decryption services, software and hardware, can be plugged in.

Three types of keys are used to encrypt data: Master Encryption Key (MEK), Data Encryption Key (DEK), and Block Encryption Key (BEK). For every ADLS account, a MEK is generated (by the user or the service) and stored in AKV. All other keys used in the account are encrypted using the account's MEK. The user can generate the MEK, store it securely in AKV, and provide access to ADLS to encrypt and decrypt data using the MEK but without access to the MEK itself. For service-managed keys, the account MEK is generated by the SMS and stored securely in AKV. The lifecycle of a user managed MEK is managed by the user directly in AKV. The use of MEK to encrypt other keys in the account enables the ability to "forget" all data by deleting the MEK. ADLS also supports periodic MEK and DEK rotations.

An account's DEK is generated by SMS, and forms the root key for all the subsequent file encryption keys. The DEK is encrypted using MEK and is stored in the ADLS cluster. The BEK is generated for each data block using the account's DEK and the block's ID in the TSM, and is used to encrypt the block. It is this key that minimizes the surface area as described above. Encryption, decryption, compression. decompression, and key derivation are performed in a separate security boundary [26] in the TSM.

Encryption and decryption might be expected to introduce significant overheads. However, in executing a broad mix of U-SQL queries on ADLA, producing a 60/40 read/write mix of data traffic, encryption and decryption only added 0.22% in total execution time (on a baseline of over 3000 hours).

7. CONCLUSION

ADLS has made significant improvements over Cosmos, notably in its support for open-source analytic engines from the Hadoop ecosystem, its security features, scale-out microservices and tiered storage. It is replacing Cosmos for Microsoft's internal big data workloads and is an externally-available Azure service. Feedback from users has been very positive and has also helped us identify feature areas to address, such as avoiding migrations while supporting ever-larger datasets, and querying data that is geographically distributed.

8. ACKNOWLEDGMENTS

We are grateful for the work of the ADLS team and ADLA teams in implementing the design and demonstrating its practicality at scale. We would also like to thank the Azure Storage, Azure Networking, MCIO, CISL and MSR teams for working closely with us. Individually, we wish to acknowledge the help of the following people: Nachiket Acharya, Nischay Anikar, Sai Prasanna Annamalai, Omkar Aradhya, Manjunath Ballur, Avigdor Boyko, Dave Campbell, Kiran Chandra, Pushkar V. Chitnis, Manuel Costa, Sudheer Dhulipalla, Vishwajeet Dusane, Joe Gagne, Jeffrey Gruen, Kushagr Gupta, Solom Heddaya, Herodotus Herodotou, Chris Hescott, Jon Howell, Khaled Ibrahim, Virajith Jalaparti, Sandeep Paul Karedi, Lucky Katahanas, Sai Charan Koduru, Lanyue Lu, Saugat Majumdar, Scott Marks, Fayssal Martani, Sumant Mehta, Jilin Meng, Vinit Ogale, Kavan Patil, Yashwant Patil, Puneeth Poduri, Mihail Popescu, Sriram Rajamani, Sriram Rao, Joseph Raisanen, Vivek Ratan, Padmashree Ravindra, Shradha Revankar, Jitendra Sangwan, Sukanya Sreshta, Shravan Srinivas, Chakrapani Bhat Talapady, Pradeep Tamma, Aditya Telidevara, Sneha Vijayarajan, Gaoxiang Xu, and Fan Zhang. Finally, thanks to the many people who reviewed the paper and provided feedback.

9. AUTHORS' ADDRESSES

*Microsoft, One Microsoft Way, Redmond, WA 98052 USA

+1 425 882 8080

**#9, Vigyan, Lavelle Road, Floors Gr, 2 &3, Bangalore, KA, 560001, India

+91 (80) 66586000

10. REFERENCES

[1] https://docs.microsoft.com/en-us/azure/data-lake-analytics/data-lake-analytics-overview

[2] J.I. Aizikowitz. Designing Distributed Services Using Refinement Mappings, Cornell University TR89-1040.

[3] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J.M. Hellerstein, R. Sears. Boom Analytics: Exploring Data-Centric, Declarative Programming. In Eurosys 2012.

[4] J. Baker, C. Bond, J.C. Corbett, J.J. Furman, A. Khorlin, J. Larson, J.M. Léon, Y. Li, A. Lloyd, V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In CIDR, 2011.

[5] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F.ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: a highly available cloud storage service with strong consistency. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP), pages 143-157, 2011.

[6] D. Campbell and R. Ramakrishnan. Tiered Storage. Architectural Note, Microsoft, Nov 2012.

[7] R. Chaiken, B. Jenkins, P-A Lar.son, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (August 2008), 1265-1276.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears. Benchmarking cloud serving systems with YCSB. In ACM SoCC, 2010.

[9] C. Diaconu, C. Freedman, E. Ismert, P-A. Larson, P. Mittal, R. Stonecipher, N. Verma, M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pages 1243-1254.

[10] C. Douglas and V. Jalaparti, HDFS Tiered Storage, 2016 Hadoop Summit, June 28-30, San Jose, California.

[11] S. Ghemawat, H. Gobioff, and S-T. Leung. The Google file system. In SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 29–43, New York, NY, USA, 2003. ACM Press.

[12] A. Grünbacher. Access Control Lists on Linux. SuSE Lab.

[13] http://hbase.apache.org/.

[14] HDFS Permission Guide http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsPermissionsGuide.html

[15] HDFS-9806: Chris Douglas. "Allow HDFS block replicas to be provided by an external storage system" https://issues.apache.org/jira/browse/HDFS-9806

[16] J. Howell, J. R. Lorch, and J. Douceur. Correctness of Paxos with replica-set-specific views. Technical report MSR-TR-2004-45, Microsoft Research, 2004.

[17] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. ACM Transactions on Computer Systems (TOCS), 6(1):51-81,1988.

[18] H.T. Kung and John T. Robinson, "On Optimistic Methods for Concurrency Control," ACM Transactions on Database Systems, vol. 6, no. 2, pp. 213-226, June 1981.

[19] L. Lamport. The part-time parliament. ACM Transactions on Computer Systems, 16(2):133–169, May 1998.

[20] L. Lamport. Paxos made simple. SIGACT News, 2001.

[21] L. Lu, H. Herodotou, R. Ramakrishnan, S. Rao, G. Xu. Tiered Storage Based Hadoop on Azure, Microsoft, 2013.

[22] B. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly available distributed systems. ACM PODC 1988.

[23] POSIX FAQ http://www.opengroup.org/austin/papers/posix_faq.html

[24] Draft Standard for Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Interface Amendment #: Protection, Audit and Control Interfaces [C Language], IEEE Computer Society, Work Item Number: 22.42. Draft P1003.1e #17, 1997.

[25] J. Rao, E. J. Shekita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. Proceedings VLDB., 4(4):24-254, 2011.

[26] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, M. Russinovich, VC3: Trustworthy Data Analytics in the Cloud using SGX, in IEEE Symposium on Security and Privacy, 2015

[27] P. Schwan. Lustre: Building a file system for 1000-node clusters. In Linux Symposium, 2003.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pages 1-10, 2010.

[29] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovslannikov, D. Reeves. Sailfish: a framework for large scale data processing. In ACM SoCC, 2012.

[30] http://usql.io/.

[31] R. Van Renesse and F. Schneider. Chain replication for supporting high throughput and availability. In OSDI, 2004.

[32] S. Weil, A. Leung, S. Brandt, and C. Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In Workshop on Petascale Data Storage, 2007.

[33] B. Welch, J. Ousterhout. Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System, 6th ICDCS, 1986.

[34] WinFS, en.wikipedia.org/wiki/WinFS