SDC
STORAGE DEVELOPER CONFERENCE
SNIA SANTA CLARA, 2016

Azure-consistent Object Storage in Microsoft Azure Stack

Ali Turkoglu
Principal Software Engineering Manager
Microsoft

Mallikarjun Chadalapaka
Principal Program Manager
Microsoft

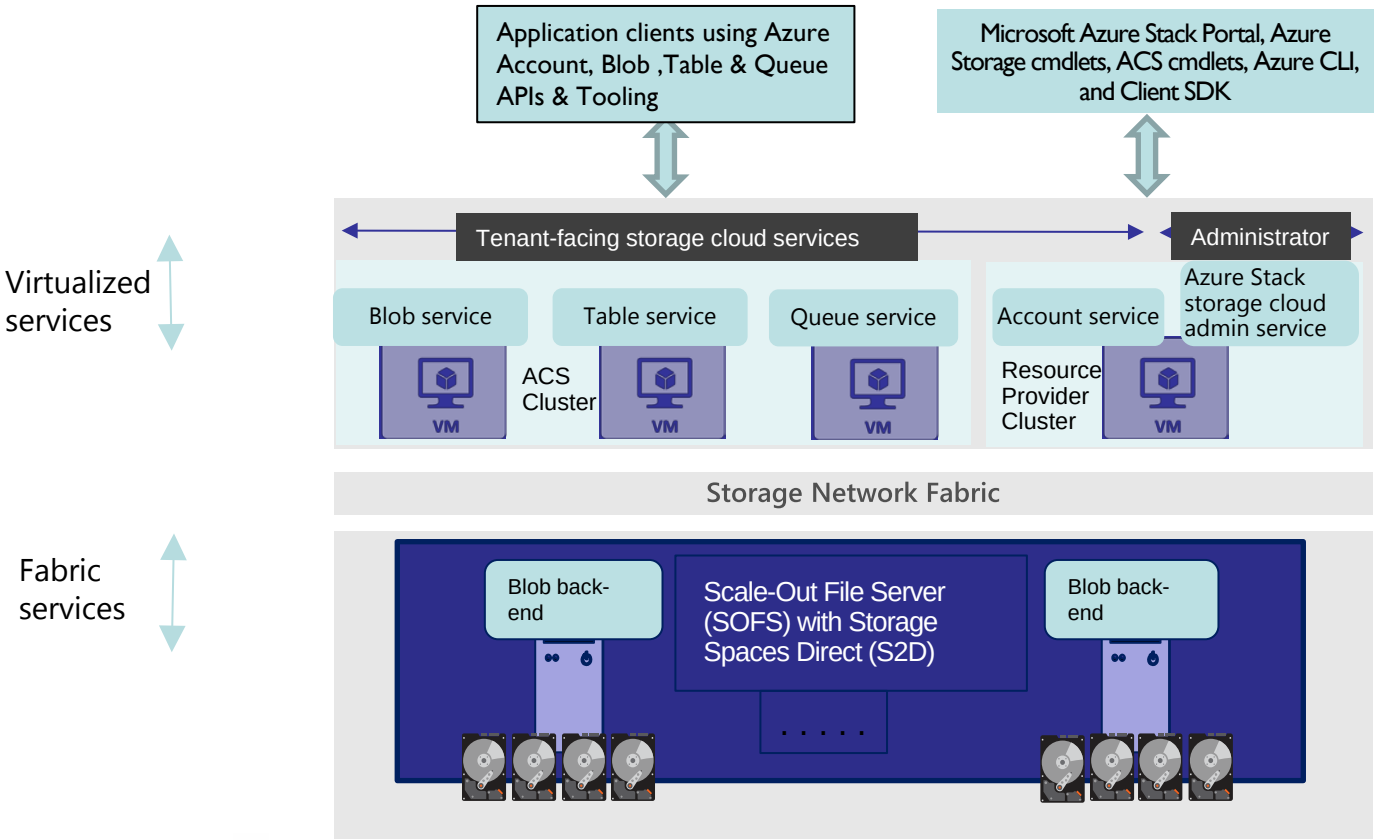
Agenda

- Context, Solution Stack, Architecture, ARM & SRP
- ACS Architecture Deep Dive
- Blob Service Architecture & Design
- Questions/Discussion

Azure-consistent storage

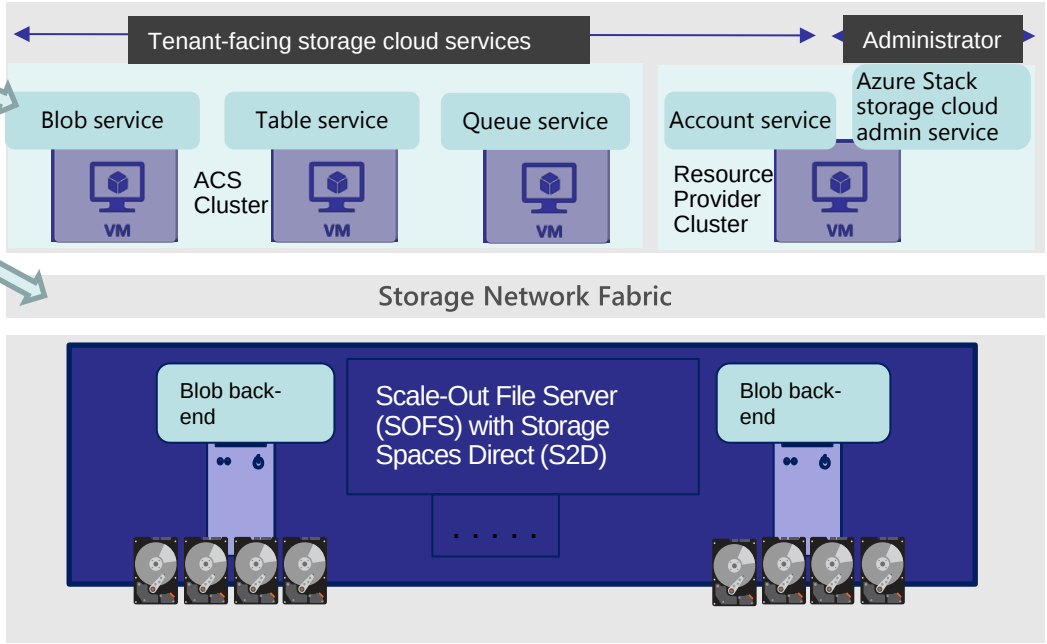
- Cloud storage for Azure Stack
- Azure-consistent blobs, tables, queues, and storage accounts
- Administrator manageability
- Enterprise-private clouds or hosted clouds from service providers
- IaaS (page blobs) + PaaS (block blobs, append blobs, tables, queues)
- Builds on & enhances WS2016 Software-Defined Storage (SDS) platform capabilities

Azure-consistent Storage: Big Picture



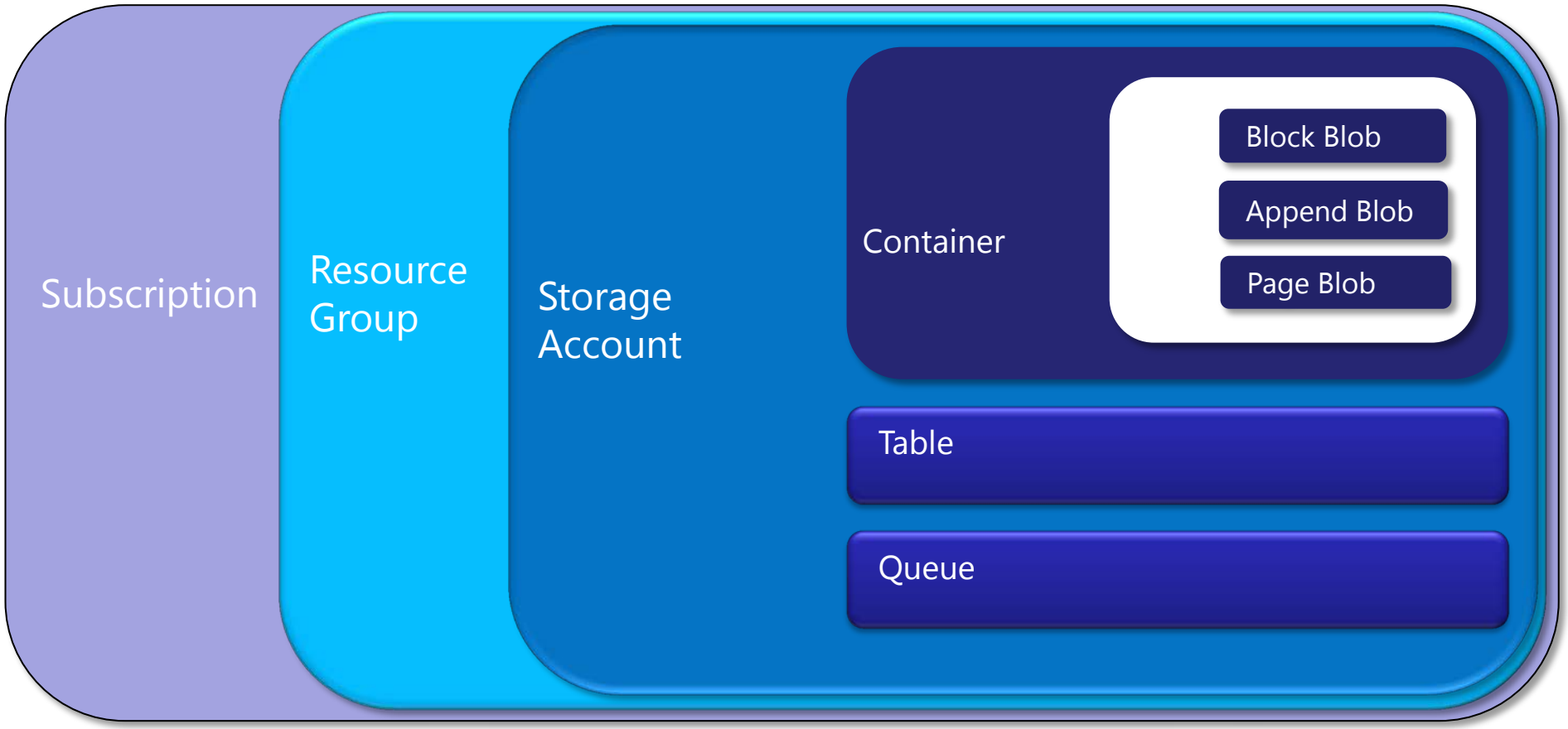
Clustering Architecture

- Azure Service Fabric (ASF) guest clusters for cloud services
- Hyper-converged Windows Server Failover Cluster (WSFC) Host fabric
- WSFC enhances ASF cluster resiliency
  - HA VMs via Hyper-V host clustering
  - Anti-affinity policies on VMs to ensure all VMs never failover to same host
  - Application health monitoring on Service Fabric service for timely detection of service hang situations
- WSFC & CSVFS* provide the basis for blob service HA model



*Cluster Shared Volume File System

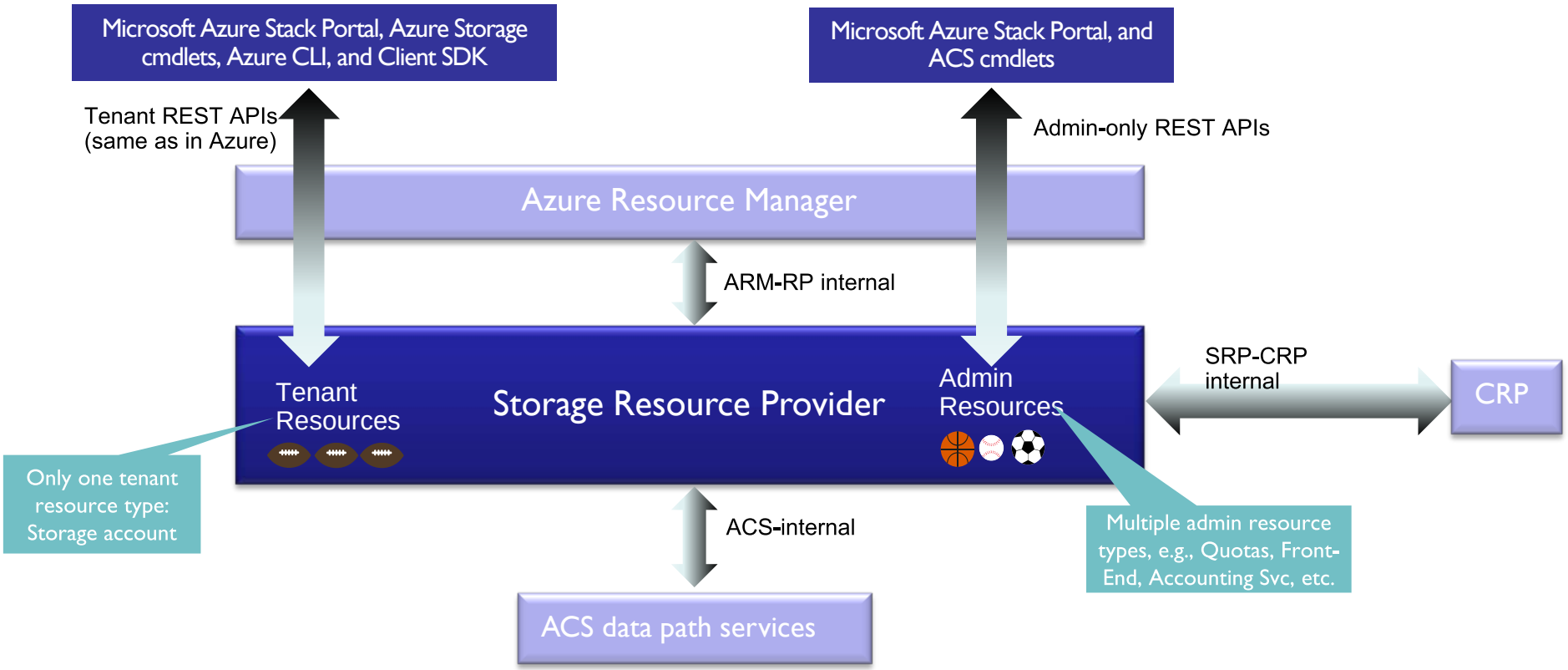
Relating Azure Storage Concepts



ARM & Resource Providers

- Azure Resource Manager (ARM) in Azure Stack
  - Provides an Azure-consistent resource management model
  - Clients can use REST APIs, PS cmdlets, or Portal experiences to manage an Azure Stack deployment
- Resource Provider (RP) is a plug-in which enables Azure-consistent management of a type of infra
  - Compute Resource Provider (CRP)
  - Network Resource Provider (NRP)
  - Storage Resource Provider (SRP)
  - ….
- Users express desired state to ARM via templates
  - Template is a declarative statement about what the user wants
  - ARM drives all necessary orchestration, including imperative directives to RPs

ACS Management Model



IaaS VM Storage

- All VM storage in Azure Stack resides in blob store
- Every OS or Data Disk is a page blob
  - Page blob → ReFS file
  - Starts in REST API access mode
- CRP and SRP collaborate behind the scenes
  - Page blob toggles to ‘SMB-only’ at VM run time
  - Super-optimized Hyper-V-over-SMB I/O path

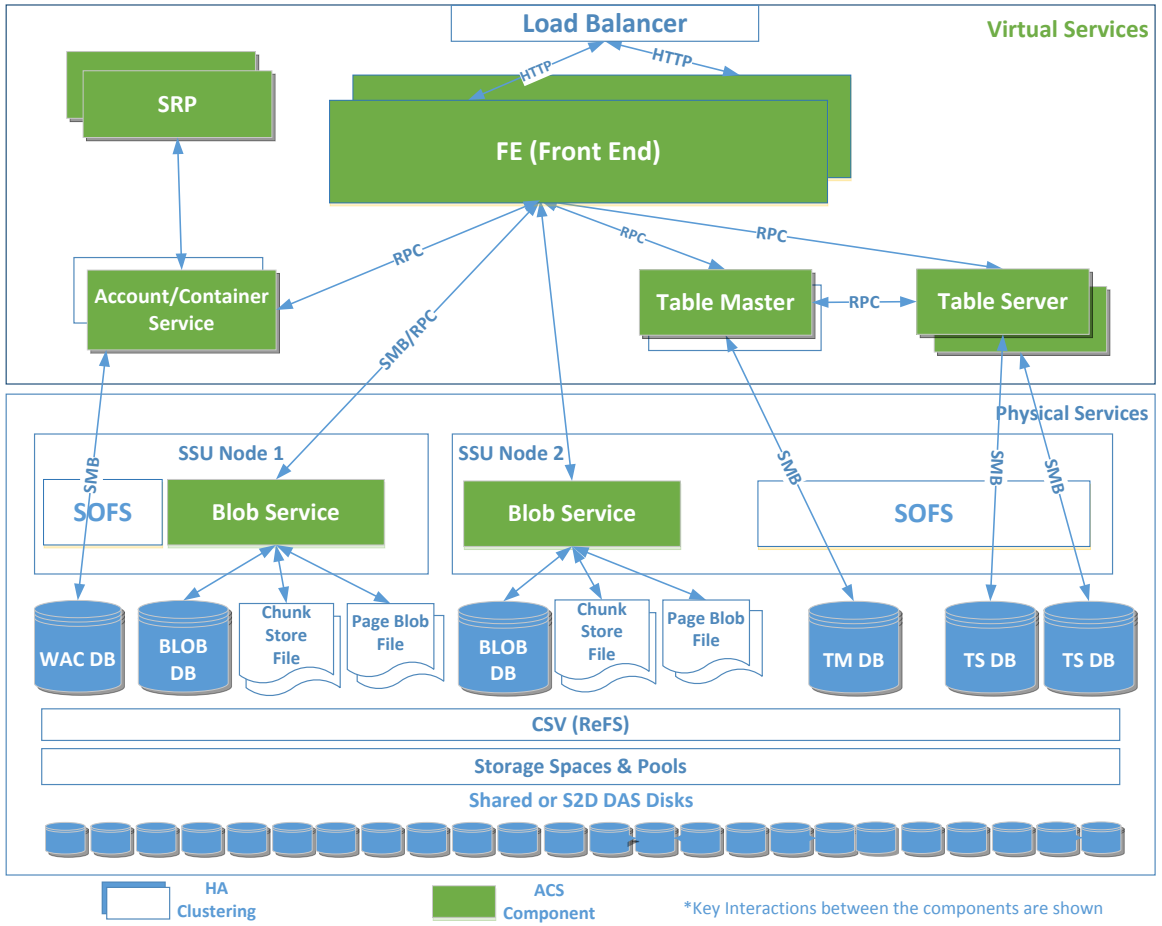
ACS Architecture Deep Dive

Key Requirements and Challenges for Object Storage on MAS

- **Atomicity** guarantees, 4MB put page, 4MB put block.
- **Data Consistency** guarantees, after a successful atomic write, any reads after that write gets the latest value.
- Immutable blocks that allow re-composition
- **Snapshot isolated** copy for read operations.
- Case sensitive blob names
- Historical 512 byte page **alignment** for page blobs (vs. ReFS 4k cluster size).
- Expensive list blob/enumeration including blob metadata.
- **Durable**: Synchronously stores 3 replications before write completes.
- **Scalable**: Up to millions of blobs under a container and millions of containers
- **Highly Available**: 99.9% read/write for local region
- Built with **Fault Tolerance** in mind at every component. Verified crash consistent implementation.
- Specific to MAS architecture, Hyper-V access to page blobs over SMB is required.
- Public vs. Private cloud scale differences:
  - I.e. Azure min deployment stamp is ~80 racks. Typical small scale on premise needs 4 nodes – few racks.

ACS Architecture

- **SRP (Storage Resource Provider):** Integrates with ARM and exposes resource management REST interfaces for storage service overall.
- **FE**: Provides REST interface Front End, consistent with Azure.
- **WAC:** Storage account management, user requests authentication, and container operations.
- **Blob Service:** Implements the blob service backend. Stores block and page blob data in file system/Chunk Store, and metadata in ESENT DB.
- **Table Master:** Maps user table to database/TS instances.
- **Table Server:** Handles table query & transactions in databases.
- **Storage:** SOFS exposes a unified view of all tiered storage to compute nodes as CA shares. Provides fault tolerance & local replication.



Blobs - Semantic Requirements: See MSDN

BLOCK BLOBS

- Client uploads individual **immutable blocks** with PUT-BLOCK for future inclusion in a block blob. Block size may be up to **4 MB**. A blob can have up to 100,000 uncommitted blocks. Maximum size of uncommitted block list is 400 GB.
- Followed by a PUT-BLOCK-LIST call to assemble the blob. Maximum size supported for Block Blob is **200 GB** and **50,000 committed blocks**.
- Blocks must retain their identity to permit later PUT-BLOCK-LIST calls to re-arrange.
- Unused blocks are lazily cleaned up after a PUT-BLOCK-LIST request. In the absence of PUT-BLOCK-LIST, uncommitted blocks are garbage collected after 7 days.
- Blob names are **case-sensitive**. At least one character long, and at most 1024 characters.
- All Blob operations **guarantee atomicity** where it either happened as a whole, or it has not happened at all. There is no undetermined state at failure.
- For Block Blobs each GET-BLOB request gets a **snapshot isolated copy** of the Blob data (or request fails if this cannot be accommodated.

PAGE BLOBS

- Client creates a new *empty* page blob by calling PUT-BLOB. A page Blob starts as sparse and its size can be up to 1 TB.
- Random Read/Write access
- Client than calls PUT-PAGE to add content to the Page Blob. PUT-PAGE operation writes a **range of pages** to a Page Blob. **Put-page operation must guarantee atomicity.**
- Calling Put Page with the **Update** option performs an in-place write on the specified page blob. Any content in the specified page is overwritten with the update.
- Calling Put Page with the **Clear** option releases the storage space used by the specified page. Pages that have been cleared are no longer tracked as part of the page blob.
- Each range of pages submitted with Put Page for an update operation may be up to **4 MB in size**. The start and end range of the page must be aligned with **512-byte boundaries**

Azure Blobs Object API : See MSDN for details

- Common Blob Operations

  Put Blob, Get Blob, Get/Set Blob Properties, Get/Set Blob Metadata, Lease Blob, Snapshot Blob, Copy Blob, Abort Copy Blob, Delete Blob

- Operations on Block Blobs

  Put Block, Put Block List, Get Block List

- Operations on Append Blobs

  Append Block (operation commits a new block of data to the end of an existing append blob)

- Operations on Page Blobs

  Put Page, Get Page Ranges

Block Blob Object Design Challenges on Traditional File System

- Why not implement Block Blobs as file objects?
  - Isolation/atomicity and unique composition requirements are the key offenders.
  - Lack of any form of acceptable atomicity support on NTFS.
    - Rename/Create path on file system is prohibitively expensive
  - API semantics does not map to files, immutable vs. random access
  - Enumeration & Rich Metadata operations requires Index and DB, and transactions.
  - SMB access to block blobs are not needed
  - Namespace should be in database, but not in file system, to meet scale demands, and other requirements
  - Kernel mode filter driver would have been needed to implement stream maps, and atomicity, but even then we would still need an index and transactional DB in kernel mode, thus increasing the cost and complexity.
  - Dedup integration complexity

Key Design Decisions for Blob Service

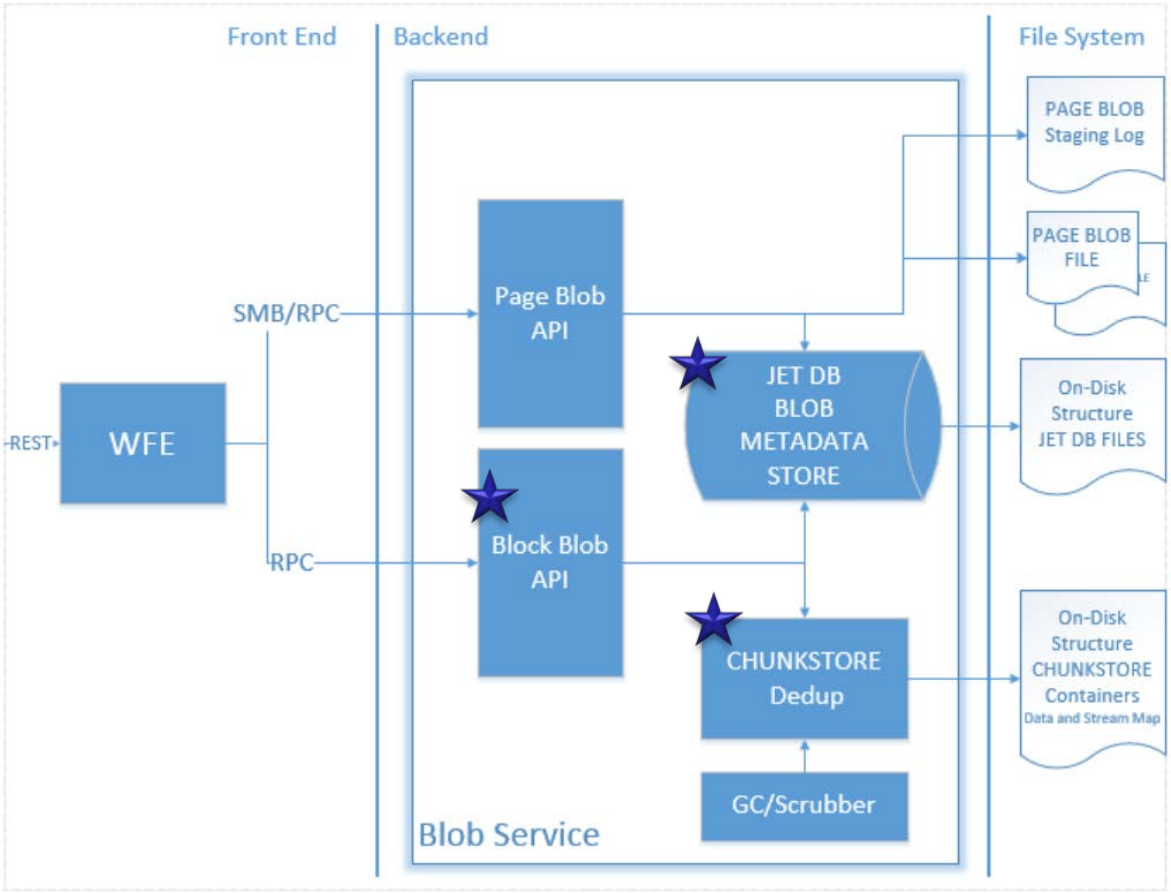
- ReFS is used for Page Blob Implementation for Page Blob Snapshots and Page Blob 4 MB atomic write via ReFS duplicate extent (FSCTL_DUPLICATE_EXTENTS_TO_FILE)
- Page Blob and Block Blob implementations must be in same service to
  - Share the DB implementation, Metadata operations
  - Meet the REST API requirement to have block and page blobs that could be under same container.
- SMB is the Hyper-V data transport for Page Blobs because
  - Page Blobs are designed as (backed by) files
  - Highly optimized data path for Hyper-V over SMB
- RPC is chosen as data transport from WFE to Blob service because
  - WFE cannot write directly to ChunkStore container files. Extending ChunkStore implementation for this deemed costly.
  - Alternatively staging Put Blob/Block writes via a file would cause additional write penalty.
  - Block Blobs are NOT files.

ACS Service Design Principles

- Keep it simple
  - Choose a good achievable limit in V1. Build a solid base and iterate/optimize from there.
- More than API consistency
  - Guaranteed strong data consistency, durability, and transaction.
  - Build with high availability and fault tolerance in every component.
  - Design for management & architecture simplicity.
- Build on available technologies and not reinvent the wheel
  - Depend on SOFS (with ReFS, Storage Spaces Direct), CSVFS for highly available, redundant storage. No application-level data replication.
  - Use ESE/NT (JetDB) engine for table data and blob metadata storage, transaction and indexing. Proven through Exchange deployments.
  - Leverage Dedup Chunk Store for block blob data storage/implementation.
  - Build on ServiceFabric/WSFC for high availability, scaling out and load balancing.

Block Blob Service Design

- **Global namespace** exists in the DB.
- **User mode only access**/store. No file system access to the block blobs; neither for namespace nor for data.
- **Use *Dedup ChunkStore** implementation to store committed and uncommitted blocks, and the stream maps.
- **Design for Azure Parity** from get go.
- Azure blob metadata is stored in ESE/NT DB.
  - To optimize metadata only operations
  - To implement Blob API GC semantics.
- DB Scope is set of containers and their blobs metadata.



Why ChunkStore?

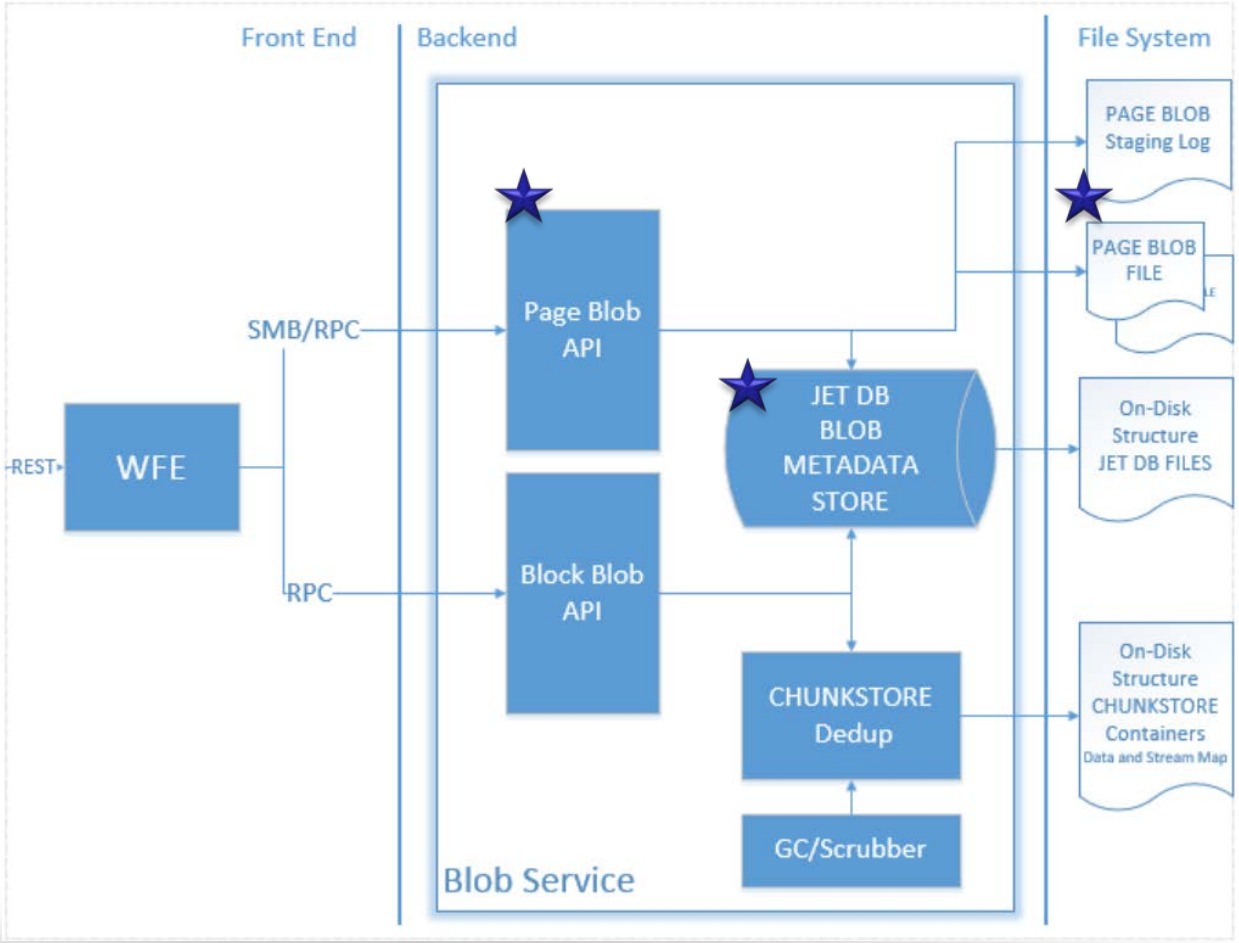
- ChunkStore implements immutable file containers for chunk and stream maps.
- Append Only Chunk Insertion
- Used also as part of Deduplication Feature and proven.
- Integrity checks for detecting page corruptions, and read from a replica for in place patching.
- Shallow and Deep Garbage Collection.
- Data chunks shared by root blob and snapshots (deduplication)
- Guarantee crash-consistent file commit (Precise order of operations guarantees data integrity at each stage)
- Various chunk size support (up to 4MB)
- Efficient self contained chunk id referencing

Why ESE/NT?

- **ESE**
  - Extensible Storage Engine (ESE), also known as JET Blue, is an ISAM (Indexed Sequential Access Method) data storage technology. It provides transacted data update and retrieval
- **Transactions & Index**
  - The ESE database engine provides Atomic Consistent Isolated Durable (ACID) transactions.
- **Logging and crash recovery**
  - ESE has write ahead logging and snapshot isolation for guaranteed crash recovery.
  - The application-defined data consistency is honored even in the event of a system crash.
- Good **Backup/Restore** support
  - ESE supports on-line backup where one or more databases are copied, along with log files in a manner that does not affect database operations
- **Page corruption detection** and read from replica and patch in place (FSCTL_MARK_HANDLE / MARK_HANDLE_READ_COPY)
  - Automatic **DB scan** and corruption detection
  - ESENT engine self-detecting and auto-correcting checksum errors on a Jet database stored on a Spaces triple-replica remotely accessed via SMB/CSVFS
- **Used at scale** in exchange workloads.

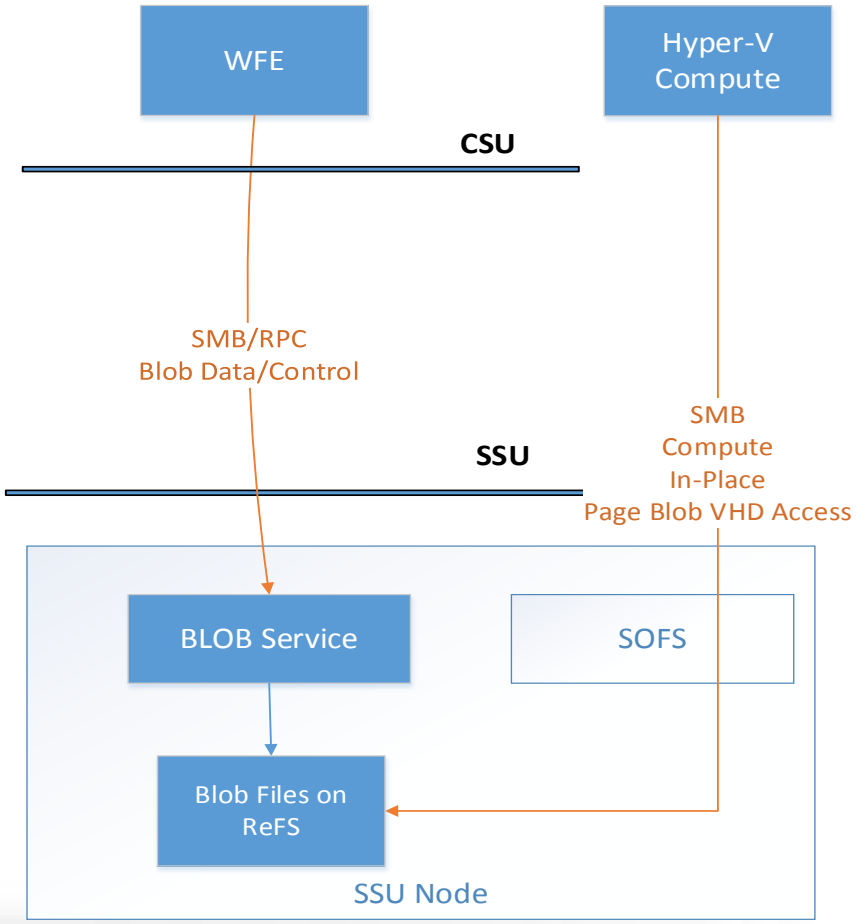
Page Blob Service Design

- **Global namespace** exists in the DB.
- **Page blobs are files** stored in ReFS volume.
- Blob is a linear mapping to the file. There is no stream map.
- **Page blobs also use DB** to store azure blob metadata.
- To enable exclusive in-place access (direct via SMB) for Hyper-V, check-out, and check-in semantics supplied.
- No concurrent REST vs. File System access.



Page Blobs - Compute vs. REST data access path

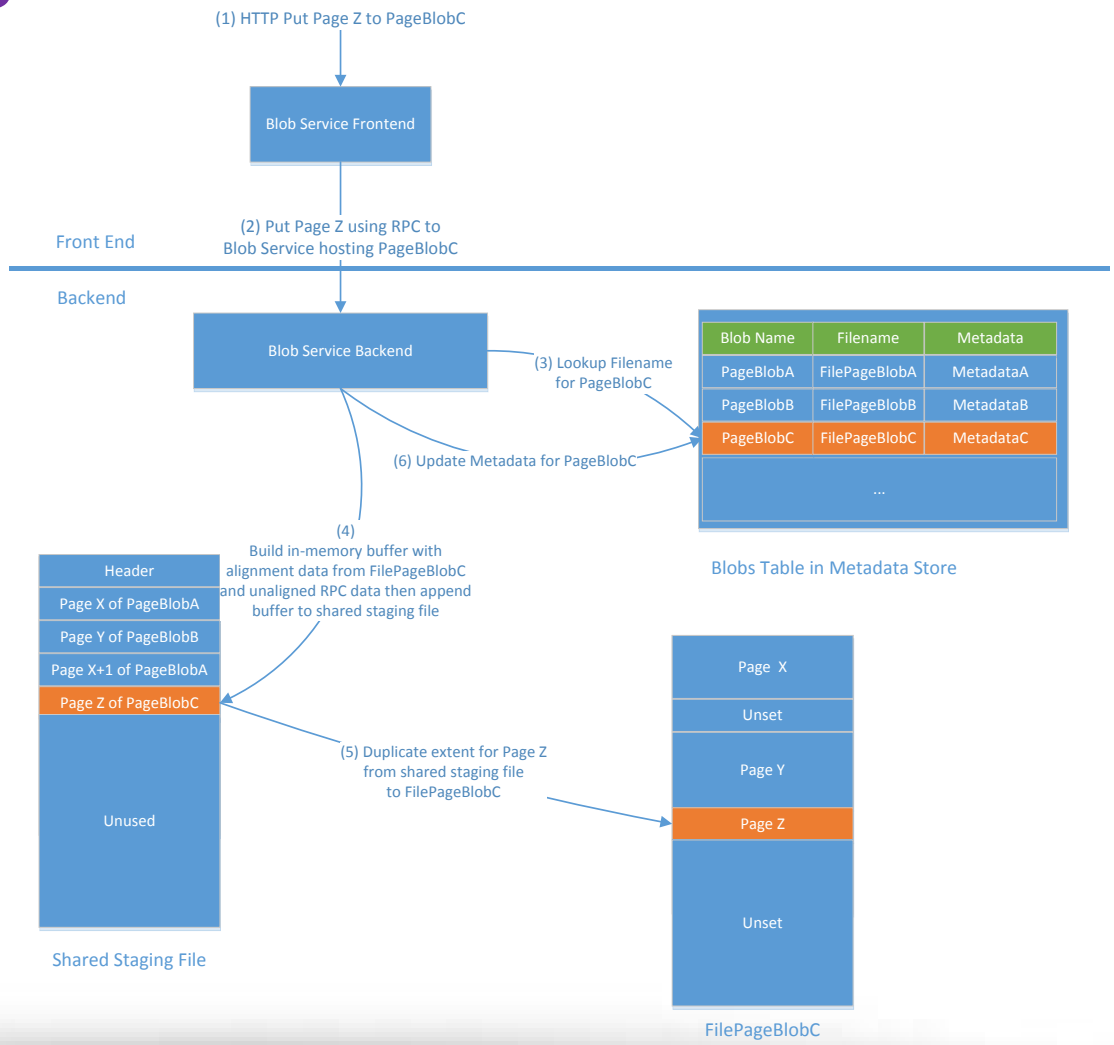
- Blob REST access is via the Blob service.
- Compute page blob access (Hyper-V accessing VHDs) is direct to the file over SMB, same path as today with RDMA etc.
- Once a blob is "checked out" for compute access, it is not accessible through REST.



Blob Service Design Data Flow & Representation

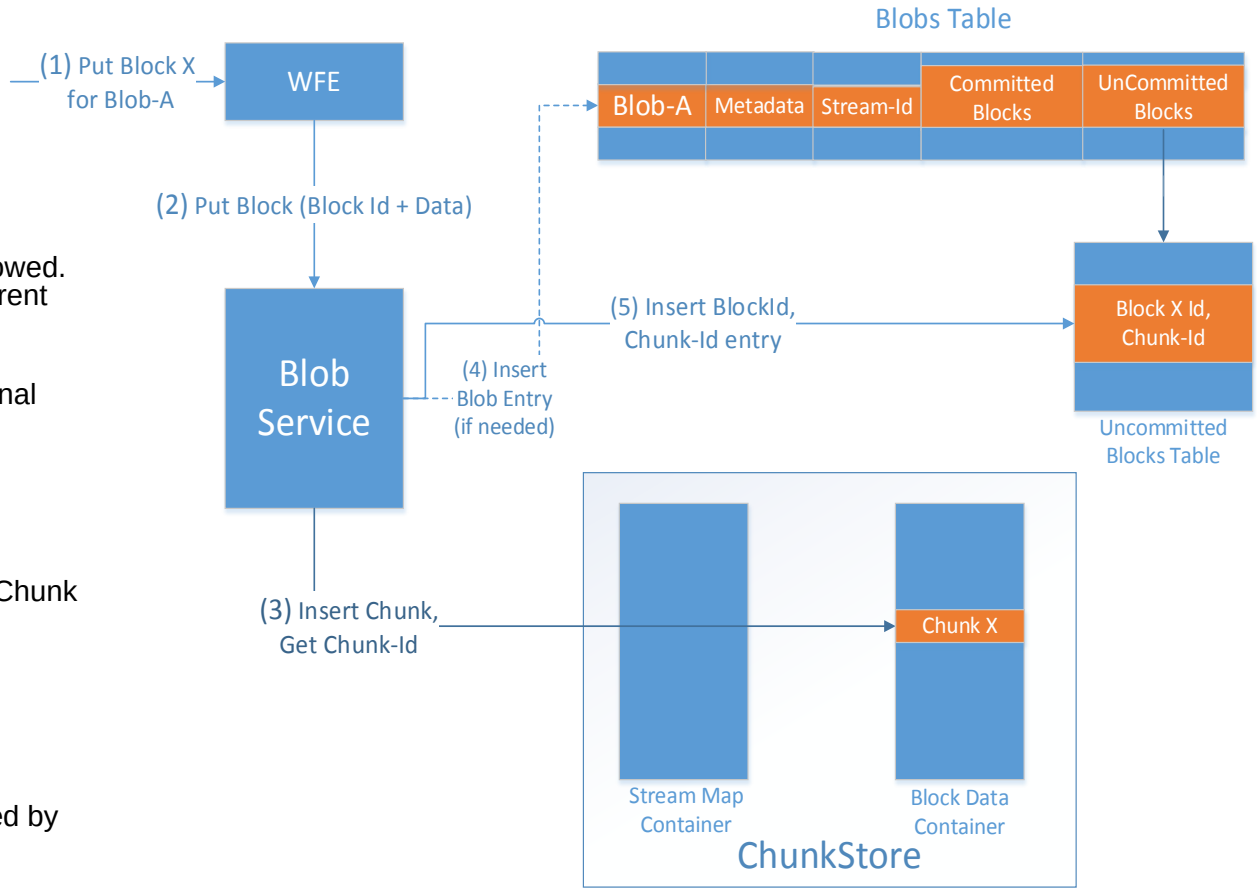
PAGE Blob Design on ReFS

- Blob Service backend maintains its private log per managed volume to “stage” page writes. Single log for all page writes within a single volume. Arbitrary log write size exceeds 4MB REST requirement .
- Append-only writes to staging log with control, request parameters and page write data use standard logging techniques to achieve atomic log writes.
- ReFS duplicate extents feature used to atomically insert write data from staging log into target page blob.
- Final Commit call updates metadata and returns ETAG/LastModified
- Crash Consistency achieved via staging log write, ReFS extent duplication, sequenced metastore update with log replay from last valid checkpoint on a service restart.
- Staging log uses checkpointing to manage space consumption on volume and volume restart recovery.



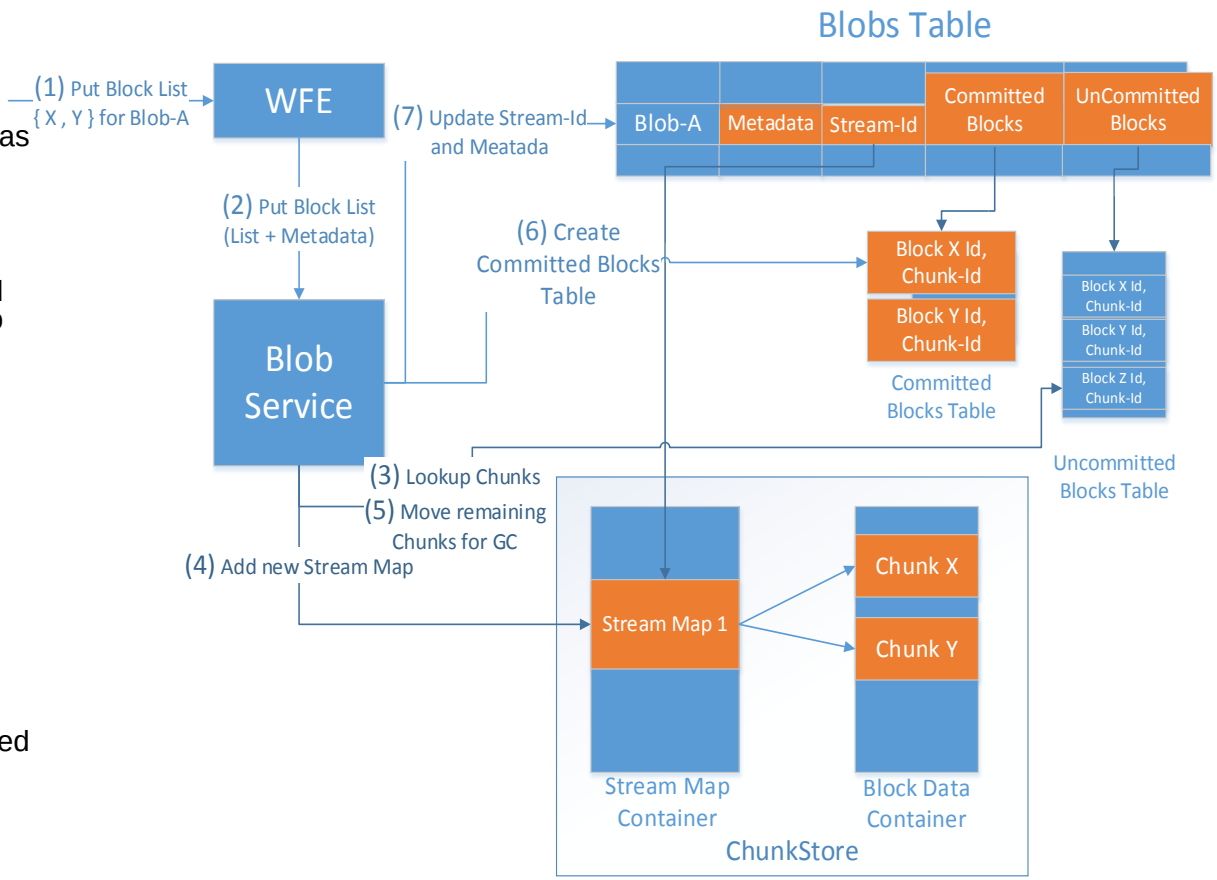
BLOCK BLOB – PUT BLOCK

- Blocks are uploaded individually first. Followed by Put-Block-List call later.
- Put Block operation does not alter the composition of the blob and it does not change its existing stream map.
- Concurrent put block operations are allowed. However the Block Sharing across different Blobs is not allowed .
- Crash Consistency achieved via additional Log Write (not shown)
- Service needs to maintain uncommitted blocks LV Column in DB:
- To Persist Azure Block Id (64 bytes) to Chunk Id mapping for uncommitted blocks.
- To search uncommitted blocks in a performant way at Put-Block-List (or at recovery)
- To enforce/implement GC policy required by the Blob API.



BLOCK BLOB – PUT BLOCK LIST

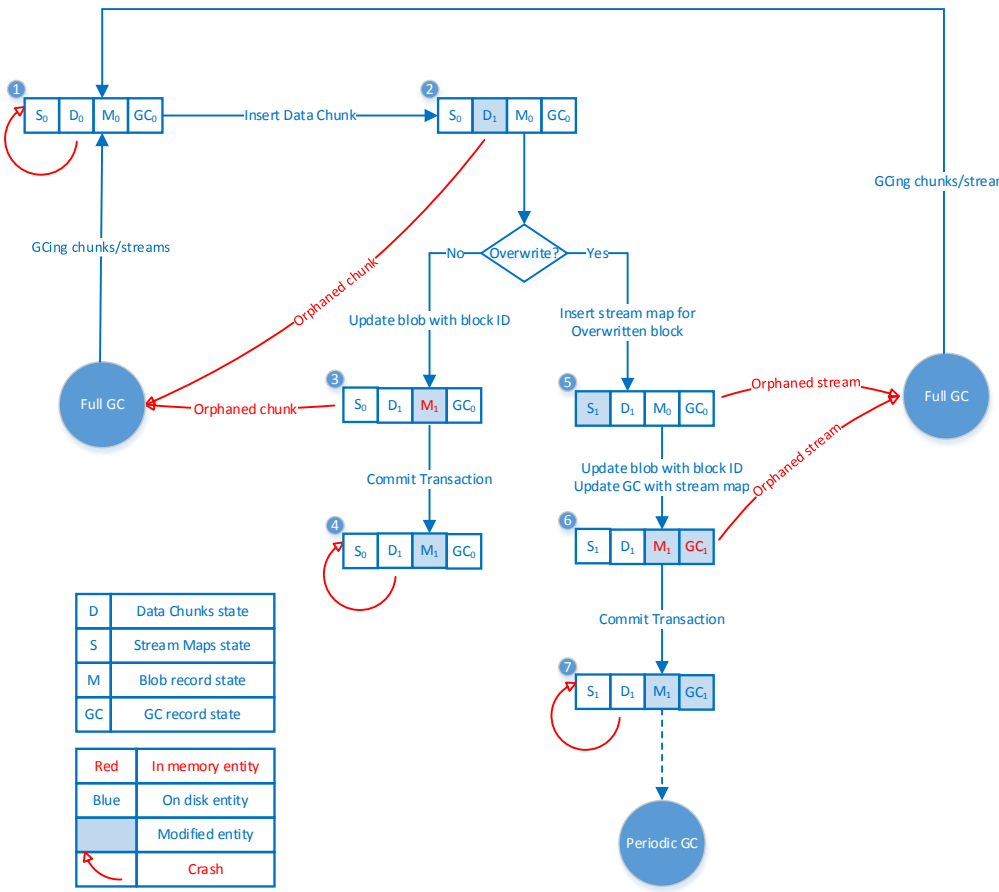
- Concurrent Put Block List calls to the same block blob are serialized by the blob service.
- Put Block List modifies the metadata as well, not just the composition of the blob.
- Put Block List searches among committed and uncommitted sets and allocates and inserts a stream map to ChunkStore.
- Blob entry in the Blobs table has the "current" stream map ID. This shows the current composition of the blob.
- Crash Consistency is achieved by additional log writes (not shown)
- Committed Block List LV (long value) column is needed to
- Efficiently search referenced committed blocks at put block list.
- To GC no longer needed previously committed blocks at Modifying Put-Block-List.



Blob Service Crash Consistency

Example Crash Consistency Approach

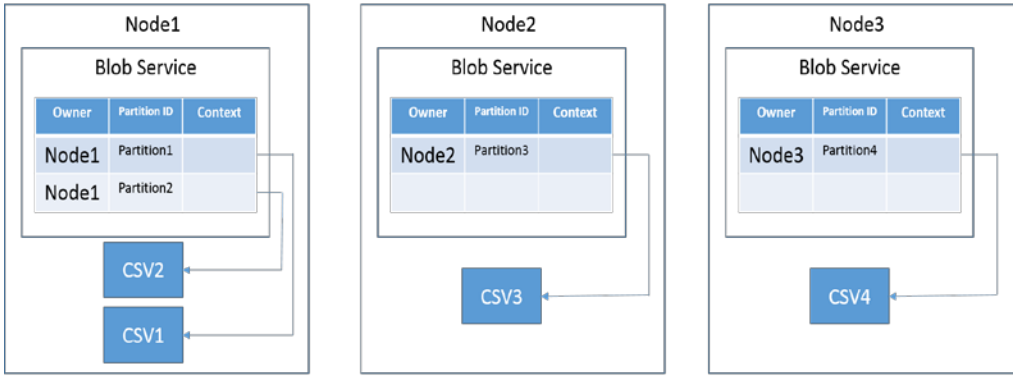
State	Crash just before normal action	Normal action	Coverage ID
1. The state consists of a valid metadata blob record that might have a pre-existing blob record or not (M_0), valid stream map state that might have a stream for the existing blob or no stream in case the blob doesn't exist (S_0) and a valid GC metadata table that has no records for related to this blob (GC_0).	We remain in this state.	If the operation conditions succeed, then insert the new data chunk into the chunk store. This takes us to state 2. Otherwise, if the operation conditions fails, then remain in this state	1
2. The state consists of a valid metadata blob record (M_0) that is not yet updated with the valid inserted data chunk ID (D_1).	On demand full GC. For more details about the full GC, please refer to the full GC crash consistency section.	If there is no uncommitted block with the same block id, then begin transaction to update the metadata store with the newly inserted data chunk id. This takes us to state 3. Otherwise, if there is an uncommitted block with the same block id, then create a stream map for it. This takes us to state 5.	If blob exists: 2, 301 Else 2, 100
3. The state consists of in memory metadata update for the blob record (M_1) that is updated with the valid data chunk id (D_1).	On demand full GC. For more details about the full GC, please refer to the full GC crash consistency section.	Commit the DB transaction. This takes us to state 4.	101, 102
4. The state consists of a valid metadata update for the blob record (M_1) that is updated with the valid inserted data chunk id (D_1).	Will remain in this state.	This is a successful terminal state.	103



Blob Service HA Model

Blob Service HA model on WSFC & CSVFS

- Multiple instances of Blob Service, each running on a single physical machine.
- Blobs are stored in a cluster file system (CSVFS).
- Blob namespace is partitioned among the blob service instances.
- Each Blob Service maintains a partition mapping table.
- CSV volumes can move between nodes to remain highly available.
- A Blob client maintains a mapping of the partition ID to the node name on which the partition is hosted.
- The mapping can change due to node failover and CSV failover.



Container ID	Partition ID
Container1	Partition3
Container2	Partition2
Container3	Partition4

Container ID	List of Blob IDs
Container1	Blob1,Blob2
Container2	Blob3,Blob4
Container3	Blob1,Blob3

Questions?

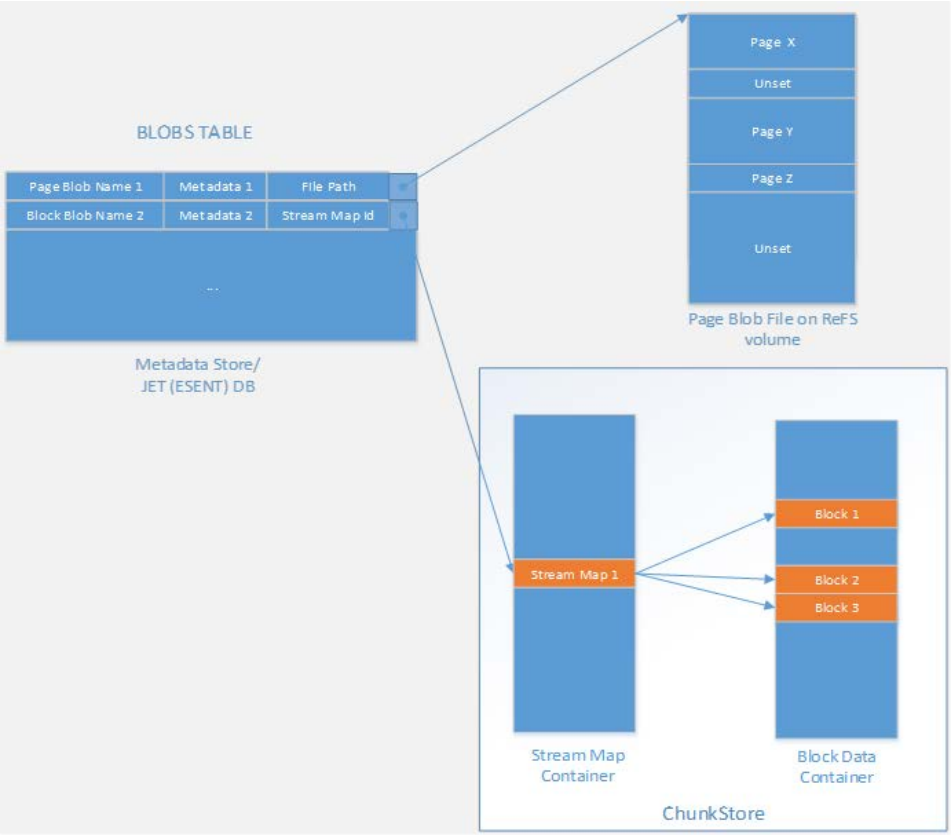
Appendix

Differences against Azure Storage

- No Standard_RAGRS or Standard_GRS
- Premium_LRS: no performance limits or guarantees
- No Azure Files yet
- Usage meter differences
  - No IaaS transactions in Storage Transactions
  - No internal vs. external traffic distinction in Data Transfer
- No account type change, or custom domains

Blob Service Metadata Persistence Model

- Stream Maps being evaluated to move to DB
- Single table with mixed row/blob types



ACS HA Model

- Service Fabric is used by WFE, TM/TS, WAC, SRP instances for failover, load balance, deployment/upgrade .
- All ACS service roles are co-locatable.
- SRP collocates with other RPs in the same cluster.
- Compute cluster VM health monitoring complements in-guest Service Fabric by providing VM-level recovery.
- Blob service is running as an HA service on the SSU cluster directly on physical nodes, as a peer of SMB. It monitors CSV movements in the cluster and attach/detach to them.
- Blob service does not failover. It is multiple active configuration.