The New Era Begins
WINDOWS AZURE STORAGE: A HIGHLY AVAILABLE CLOUD STORAGE SERVICE WITH STRONG CONSISTENCY

Rana Shahout
Maria Abu-Sini
AGENDA

➤ Motivation
➤ Highlight Of Design Goals
➤ High Level Architectures
  ➤ Front End Layer
  ➤ Partition Layer
  ➤ Stream Layer
➤ Consistency Achievement
➤ Failures
➤ Load Balancing
➤ Overhead vs. performance
➤ Evaluations
➤ Design Choices
WHAT IS CLOUD STORAGE

➤ Cloud storage is a model of data storage
➤ The data is stored in logical pools, the physical storage spans multiple servers (and often locations).

WINDOWS AZURE STORAGE

➤ Scalable cloud storage system
➤ Provide storage data abstractions to program applications.
ANYWHERE AT ANYTIME ACCESS TO DATA
**CAP Theorem**

- **Consistency**
- **Availability**
- **Partition Tolerance**

*Pick 2*

*Windows Azure Storage*
DESIGN GOALS

➤ Highly Available with Strong Consistency
➤ Durability
➤ Scalability
DATA ABSTRACTIONS

➤ Blobs (Binary Large Object) include images, text files, videos and audios.

➤ Tables - Structured storage

➤ Queues - Message delivery
HOW AZURE PROVIDES ACCESS AROUND THE WORLD

➤ Through Global Partitioned NameSpace

http(s)://AccountName.<service>.core.windows.net/PartitionName/ObjectName

selected account name  locates the data  identifies objects in partition

service - Blob/Table/Queue
HIGH LEVEL ARCHITECTURE
Access blob storage via the URL: http://<account>.blob.core.windows.net/

Samp: cluster of 10-20 racks
INTRA-STAMP REPLICATION

➤ Synchronous replication
➤ Make sure all the data written into a stamp is kept durable
➤ On critical path of customer write request

*Durability against hardware failures*
INTER-STAMP REPLICATION

➤ Asynchronous replication
➤ Replication data across stamps
➤ OFF critical path of customer write request
➤ Replication is at object level (the whole object or delta changes)
➤ Used for: Disaster recovery, migrating an account’s data between stamps

*Durability against geo-disasters*
ARCHITECTURE LAYERS INSIDE STAMPS

Front-End Layer:

➤ Login
➤ Authentication
➤ Routing to Partition Server
**ARCHITECTURE LAYERS INSIDE STAMPS**

Partition Layer:
- Understand data abstractions
- Provide Optimistic concurrency
- Massively scalable index
- Log structured merge tree
ARCHITECTURE LAYERS INSIDE STAMPS

Stream Layer

➤ Streams are large files - Extents
Replicated 3 times across nodes Why?

➤ Append Only Distributed File System
➤ Cons? Pros?

➤ Operations:
➤ Open/Close/Delete/Streams
➤ Rename Streams
➤ Append writing/Random reads
STREAM LAYER
STREAM LAYER

➤ Stores bits on disk
➤ Distribute and replicate the data across servers
➤ An extent is replicated 3 times across different fault and upgrade domains
➤ Checksum all stored data
STEAM LAYER

Sequence of Blocks ~ 1GB

Unit of replication

Extent

Sequence of Blocks ~ 1GB

Unit of replication

Extent

Extent

Stream

Smallest Unit for read/write
(≈4MB)

➤ ordered List of pointer to extents

➤ Operations: Append
STREAM LAYER ARCHITECTURE

➤ Stores extents (has N disks)
➤ Maintains the storage for a set of extent replicas
➤ Contains where the replicas for given extent
STREAM LAYER ARCHITECTURE

- Monitoring the health of EN
- Creating/assigning extents to EN
- re-replication of extent replicas that are lost
- Garbage collection of extents
- Schedule Erasure code of extent

Diagram:

- Stream Master
- Extent Node - EN
CREATING EXTENTS

Partition Layer

Create Extent

EN1 - Primary
EN2, EN3 - Secondary

Allocate Extent at replica set

Stream Master

Primary EN1
Secondary EN2
Secondary EN3

Stream Layer
PRIMARY ROLE

➤ Determining the offset of the append in the extent
➤ Ordering all of the appends if there are concurrent appends
➤ Sending the append with its chosen offset to the secondaries.
➤ Returning success for the append to the client

Optimization:

Caching the extents at the client for the client can go directly to the EN without the SM
REPLICATION WITHIN STREAM LAYER

➤ Stream created

➤ SM assign three replicas for first extent
  ➤ One primary and two secondary (never changed for unsealed extent)

➤ Write to primary EN

➤ Primary is in charge of coordination the write to secondaries
ERASURE CODING SEALED EXTENTS

➤ WAS breaks an extents into N equal sized fragments
➤ Adds M error correcting code fragments using Reed-Solomon
➤ if there are NO MORE THAN M loses fragments
  ➤ Recreate the full extent

This reduce the cost of storing data from three full replicas within the stamp
APPEND FLOW

Partition Layer

Append

Ack

Stream Master

Stream Layer

Primary EN1

Secondary EN2

Secondary EN3
READ FLOW

Partition Layer

Read

From Any Replica!
Even Unsealed

Stream Master

Primary EN1
Secondary EN2
Secondary EN3

Stream Layer
Providing Bit-Wise Identical Replicas

➤ All replicas for an extent to be bit-wise the same, up to a committed length

➤ Store pointers from the partition layer index to an extent + offset

➤ In order to be able to read from any replica
PROVIDING BIT-WISE IDENTICAL REPLICAS

Replication flow

➤ All appends to an extent go to the Primary

➤ Primary orders all incoming appends and picks the offset for the append in the extent

➤ Primary then forwards offset and data to secondaries

➤ Primary performs in-order acks back to clients for extent appends

  Primary returns the offset of the append in the extent

  An extent offset can commit back to the client once all replicas have written that offset and all prior offsets have also already been completely written

  This represents the committed length of the extent
READ FAST!

➤ Submit the read with a “deadline”
➤ If the read cannot be fulfilled within the deadline EN replay it to client IMMEDIATELY
➤ Allows client to select different EN so the read is faster
➤ Do reconstruction rather than reading the data fragment
➤ Read are issued to all fragments of an erasure coded extent
➤ First N responses are used to reconstruct the requested fragment. Parallel!!
PARTITION LAYER
PARTITION LAYER ARCHITECTURE

➤ Partition Master - PM
➤ Partition Server - PS
➤ Lock Server
PARTITION LAYER

- Data model for the different types of objects
- logic and semantics to process the different types of objects
- Scalable namespace for the objects
- Load balancing to access objects across partition servers
- Transaction ordering and strong consistency to objects
PARTITION LAYER

➤ Provide an efficient way to query/get/update objects

➤ Provide a dynamically load balance

➤ Maintains an internal Object Index Table for each data abstraction

➤ Each Object Index is scalable
### Split Index Range Partition

<table>
<thead>
<tr>
<th>Account</th>
<th>Counter</th>
<th>Blob</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>pics</td>
<td>flower</td>
</tr>
<tr>
<td>Rana</td>
<td>video</td>
<td>GOT</td>
</tr>
<tr>
<td>Bob</td>
<td>video</td>
<td></td>
</tr>
<tr>
<td>Maria</td>
<td>pics</td>
<td>codes</td>
</tr>
</tbody>
</table>

**Assigned to one server**

- Front-End Server
  - A-C: PS-1
  - C’-F: PS-2
  - F’-Z: PS-2

**Cached in front-End also**

- PS-1
- PM
- PS-2
- PS-3
Each RangePartition – Log Structured Merge-Tree

- Memory Table
- Row Cache
- Index Cache
- Bloom Filters
- Load Metrics

**Writes**

**Read/Query**

**Memory Data**

**Persistent Data (Stream Layer)**

- Commit Log Stream
- Metadata Log Stream

**Row Data Stream**

- Checkpoint File Table
- Checkpoint File Table
- Checkpoint File Table

**Blob Data Stream**

- Blob Data
- Blob Data
- Blob Data
Each RangePartition – Log Structured Merge-Tree

READ HOT DATA

Persistent Data (Stream Layer)
- Commit Log Stream
- Metadata log Stream

Row Data Stream
- Checkpoint File Table
- Checkpoint File Table
- Checkpoint File Table

Blob Data Stream
- Blob Data
- Blob Data
- Blob Data

Memory Data
- Memory Table
- Row Cache
- Index Cache
- Bloom Filters
- Load Metrics

Writes
Read/Query
READ COLD DATA

Each RangePartition – Log Structured Merge-Tree

- Memory Table
- Row Cache
- Index Cache
- Bloom Filters
- Load Metrics

Persistent Data (Stream Layer)
- Commit Log Stream
- Metadata log Stream

Row Data Stream
- Checkpoint File Table
- Checkpoint File Table
- Checkpoint File Table

Blob Data Stream
- Blob Data
- Blob Data
- Blob Data
WRITE DATA

Each RangePartition – Log Structured Merge-Tree

- Memory Table
- Row Cache
- Index Cache
- Bloom Filters
- Load Metrics

Writes

Read/Query

Memory Data

Persistent Data (Stream Layer)

- Commit Log Stream
- Metadata log Stream

Row Data Stream

- Checkpoint File Table
- Checkpoint File Table
- Checkpoint File Table

Blob Data Stream

- Blob Data
- Blob Data
- Blob Data
CHECKPOINT PROCESS – NOT IN CRITICAL PATH

Each RangePartition – Log Structured Merge-Tree

- Memory Table
- Row Cache
- Index Cache
- Bloom Filters
- Load Metrics
- Writes
- Read/Query
- Memory Data

Persistent Data (Stream Layer)
- Commit Log Stream
- Metadata log Stream

Row Data Stream
- Checkpoint File Table
- Checkpoint File Table
- Checkpoint File Table

Blob Data Stream
- Blob Data
- Blob Data
- Blob Data
GEO-REPLICATION

Each RangePartition – Log Structured Merge-Tree

Persistent Data (Stream Layer)

Commit Log Stream

Metadata log Stream

Blob Data Stream

Row Data Stream

Read/Query

Write

Memory Data

Table

Row Cache

Index Cache

Blob Filters

Load Metrics

Checkpoint

File

Table

Blob Data

Commit Log Stream

Metadata log Stream

Blob Data Stream

Checkpoints

File Table

File Table

File Table
Consistency is The Key
READ – AVAILABILITY WITH CONSISTENCY

➤ **Read consistency:**
  ➤ Can read from any replica for an extents since all is bit-wise identical

➤ **Read Availability:**
  ➤ Send out parallel real requests if first read is taking higher than 95% latency
WRITE – AVAILABILITY WITH CONSISTENCY

➤ **Write consistency:**
  ➤ Appends are ordered the same across all 3 replicas
  ➤ Only return success if all 3 replicas are committed to storage
  ➤ When failure, seal extent replica set and never append anymore

➤ **Write Availability: handle failures during write**
  ➤ Seal extent replica set
  ➤ Append to new replica set
WHEN WRITE OPERATION COULD FAILS?
WRITE FAILURES

➤ Ack from primary lost when going back to partition layer

Retry from partition layer Cause duplicate records - multiple blocks to be appended

➤ Unresponsive/Unreachable Extent Node (EN) (Ack will not be backed)

Seal the failed extent, Allocate a new extent append immediately
SECENARIO 1 - WE HAVE 2 REPLICA

Partition Layer

Stream Layer

Stream Master

Append

120 at 120
120 at 120
120

Primary EN1
Secondary EN2
Secondary EN3

current len?
SCENARIO 2 - WE HAVE 1 REPLICAS

Append

Partition Layer

Primary EN1

Secondary EN2

Secondary EN3
For Data Streams, Partition Layer only reads from offsets returned from successful appends

Offset valid on any replica
DEALING WITH NETWORK PARTITION - LOGS

- Check commit length first
- Only read from:
  - Unsealed replica if all replicas have the same commit length
  - A sealed replica

![Diagram showing network partition handling]

- Partition Layer
  - Unsealed replica
  - A sealed replica

- Stream Layer
  - Use EN1/EN2

- Check commit length
AGENDA

➤ Motivation
➤ Highlight Of Design Goals
➤ High Level Architectures
  ➤ Front End Layer
  ➤ Partition Layer
  ➤ Stream Layer
➤ Consistency Achievement
➤ Failures
  ➤ Load Balancing
  ➤ Evaluations
➤ Design Choices
DYNAMICALLY LOAD BALANCING - PARTITION LAYER

➤ Partition Manager maintain index/transactions processing across PS based on load/resource utilization

➤ Only reassigns what part of the index a partition server responsible for and DOES NOT MOVE DATA
Read/Write Load Balancing

- Monitor load on each node
- On Read: select the replica to read from, start parallel reads on 95% capacity
- On Write: Seal the replica set with overloaded node, switch to new extent on another set
## OVERHEAD VS PERFORMANCE

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replication</td>
<td>Better performance, several copies to read from</td>
</tr>
<tr>
<td></td>
<td>Write to several copies</td>
</tr>
<tr>
<td></td>
<td>High Storage Overhead</td>
</tr>
<tr>
<td></td>
<td>Used with hot data</td>
</tr>
<tr>
<td>Erasure Codes</td>
<td>Specific source to read from, reconstruction is used as optimization</td>
</tr>
<tr>
<td></td>
<td>Computations needed upon failure</td>
</tr>
<tr>
<td></td>
<td>At least several updates, yet can be optimized</td>
</tr>
<tr>
<td></td>
<td>Low storage overhead</td>
</tr>
<tr>
<td></td>
<td>Used with cool data</td>
</tr>
</tbody>
</table>

How many redundant symbols?
EVALUATION
GOAL

➤ Throughput: rate at which something can be processed

➤ Main goal is to examine throughput, how many requests can the system deal with in one second

➤ Will also answer the following questions:

Achieve scalability?

Single operation vs batch operation
Given a number of VMs, for each of the following operations:

➤ Random 1KB single entity get
➤ Random 1KB single entity put
➤ Batch insert of 100 entities

compute the number of entities modified in one second
RESULTS

➤ Achieve scalability

➤ Batch operation vs Single operation

Figure 6 Table Entity Throughput for 1-16 VMs
RESULTS

➤ Achieve scalability

Figure 7: Blob Throughput for 1-16 VMs
WORKLOAD PROFILES

➤ Structured data saved in tables. Unstructured data saved in blobs.

➤ Ingress, egress is the incoming traffic (bytes)

Table 1: Usage Comparison for (Blob/Table/Queue)

<table>
<thead>
<tr>
<th></th>
<th>%Requests</th>
<th>%Capacity</th>
<th>%Ingress</th>
<th>%Egress</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>All</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blob</td>
<td>17.9</td>
<td>70.31</td>
<td>48.28</td>
<td>66.17</td>
</tr>
<tr>
<td>Table</td>
<td>46.88</td>
<td>29.68</td>
<td>49.61</td>
<td>33.07</td>
</tr>
<tr>
<td>Queue</td>
<td>35.22</td>
<td>0.01</td>
<td>2.11</td>
<td>0.76</td>
</tr>
<tr>
<td><strong>Bing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blob</td>
<td>0.46</td>
<td>60.45</td>
<td>16.73</td>
<td>29.11</td>
</tr>
<tr>
<td>Table</td>
<td>98.48</td>
<td>39.55</td>
<td>83.14</td>
<td>70.79</td>
</tr>
<tr>
<td>Queue</td>
<td>1.06</td>
<td>0</td>
<td>0.13</td>
<td>0.1</td>
</tr>
<tr>
<td><strong>XBox GameSaves</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blob</td>
<td>99.68</td>
<td>99.99</td>
<td>99.84</td>
<td>99.88</td>
</tr>
<tr>
<td>Table</td>
<td>0.32</td>
<td>0.01</td>
<td>0.16</td>
<td>0.12</td>
</tr>
<tr>
<td>Queue</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>XBox Telemetry</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blob</td>
<td>26.78</td>
<td>19.57</td>
<td>50.25</td>
<td>11.26</td>
</tr>
<tr>
<td>Table</td>
<td>44.98</td>
<td>80.43</td>
<td>49.25</td>
<td>88.29</td>
</tr>
<tr>
<td>Queue</td>
<td>28.24</td>
<td>0</td>
<td>0.5</td>
<td>0.45</td>
</tr>
<tr>
<td><strong>Zune</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blob</td>
<td>94.64</td>
<td>99.9</td>
<td>98.22</td>
<td>96.21</td>
</tr>
<tr>
<td>Table</td>
<td>5.36</td>
<td>0.1</td>
<td>1.78</td>
<td>3.79</td>
</tr>
<tr>
<td>Queue</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
DESIGN CHOICES
Customers run their applications as a tenant on VMs. Nodes running a customer’s service code are separate from nodes providing their storage.

Advantages:

Isolation between compute and storage.

Both systems can load balance independently.

Tradeoff: additional network bandwidth between VMs and storage.
Main goals: objects should be partitioned to servers so that each object is assigned to exactly one server.

Besides, need to achieve:

- Isolation: keep customer’s objects together to throttle and isolate potentially abusive accounts.
- Efficient objects enumeration.
- Load balance.
FIRST APPROACH, HASHING

• Objects are assigned to servers based on hash values of their keys.

• Advantages: easy load balance.

• Disadvantages: lose the locality of objects for isolation and efficient enumeration.
SECOND APPROACH, RANGE PARTITION

• Account’s objects are stored together within a set of RangePartitions.

• Advantages:
  1. Easier isolation.
  2. Efficient objects enumeration.

• Disadvantages: load balance system is needed.
SECOND APPROACH, RANGE PARTITION

• Works well when writes are distributed across a large number of PartitionNames.

• Downside: scaling out access to sequential patterns. For example, accessing only the very end of a table’s key range.
THROTTLING / ISOLATION, PROBLEM

- Server might be overloaded by a specific customer requests.
- Need to identify which storage accounts should be throttled.
- Motivation: well-behaving accounts are not affected.
THROTTLING / ISOLATION, SOLUTIONS

• Naïve solution: keep track of request rate for all AccountNames and PartitionNames.
  Not practical, large number of AccountNames and PartitionNames.

• Alternative: Sample-Hold algorithm:
  ✓ Track the request rate history of the top N busiest AccountNames and PartitionNames.
  ✓ First, try load balancing
  ✓ Second, throttle requests of specific patterns (high traffic to a single PartitionName, high sequential access traffic, repetitive sequential scanning).
AUTOMATIC LOAD BALANCING

➤ The system should quickly adapt to various traffic conditions.

➤ Main goal: maintain availability in the multi-tenancy environment.

➤ First Solution: use single number to quantify “load” on each RangePartition and each server.

➤ Product of request latency and request rate.

➤ Advantages:
  ➤ Easy to compute
  ➤ Reflects load incurred by requests on a server.

➤ Disadvantage: should consider high CPU and network utilization.
AUTOMATIC LOAD BALANCING, ALGORITHM

➤ First Step:

PM sorts all RangePartitions based on several parameters (request throttling, request timeout, size of partition ...).

Then PM picks a small number of RangePartitions to split.

➤ Second Step:

PM sorts PSs based on load balancing metrics (request load, CPU load, network load).

Identify which PSs are overload and which are lightly loaded.

PM chooses one of the heavily loaded PSs. If one of its RangePartitions were recently splitted, then it is moved to another lightly loaded PS.

If there are still highly loaded PSs, offload RangePartitions from them to other lightly loaded PSs.
AUTOMATIC LOAD BALANCING, TWO STEPS?

➤ Suppose we are offloading a whole RangePartition.
   The loaded RangePartition will move all of the traffic to another PS.
   Did not solve the problem.

➤ Second step?
   One PS might be heavily loaded due to several RangePartitions.
   Need to find which PSs are heavily loaded and which are lightly loaded.
AUTOMATIC LOAD BALANCING

➤ Can customize load balancing logic at runtime.
➤ Define what triggers partition split (which metrics).
➤ Try new algorithms according to traffic pattern observed.
**SEPARATE LOG FILES PER RANGEPARTITION**

➤ Motivation: performance isolation for storage accounts.
➤ Thus, use separate log streams for each RangePartition.
➤ Isolate load time of RangePartition to just recent object update in that RangePartition.
APPEND-ONLY SYSTEM

- The system is based on:
  - Append-only system.
    - Suppose data can be overwritten after it is committed to replica, how can we read from several replicas?
  - Seal and extent upon failure.
    - Suppose extent is not sealed
  - Consistency is enforced across all replicas up to committed length
APPEND-ONLY SYSTEM

➤ Advantages:
   ➤ Provide snapshot / versioning features at no cost.
   ➤ Erasure coding
   ➤ Diagnosing issues
   ➤ Recovering the system

➤ Disadvantage:
   ➤ Garbage collector is needed
END-TO-END CHECKSUMS

• Once Front-End server receives the user data, it computes the checksum and send it along with the data to the backend servers. Partition servers and stream servers verify it before processing the data.
END-TO-END CHECKSUM

➤ Advantages:

➤ Prevent corrupted data from being committed into the system.

➤ Maintain data integrity.

➤ Identify servers that consistently have hardware issues.
➤ Rack domain orthogonal to upgrade domain.

➤ Servers of the three layers are spread evenly across different fault and upgrade domain.

➤ Upgrade a single upgrade domain at a time.
UPGRADES

➤ During upgrade, storage nodes goes offline for few seconds.

➤ Naïve approach: treat upgrades as abrupt failures.

➤ Another approach:
  1. Notify PM to move partitions out of the upgrade domain.
  2. Notify SM not to allocate extents in the upgrade domain.
  3. Ensure there are enough replicas available for each extent.
  4. Validation test before proceeding to the next domain.
MULTIPLE DATA ABSTRACTIONS FROM A SINGLE STACK

<table>
<thead>
<tr>
<th></th>
<th>Massive disk capacity</th>
<th>I/O spindles</th>
<th>Run in memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blobs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queues</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

➤ All supported by the same set of storage nodes.

➤ All benefit from improvements in stream layer and partition layer.
OFFERING STORAGE IN BUCKETS OF 100TBS

➤ Limit amount of storage for an account to no more than 100TBs.

➤ All of the storage account data fit within a given storage stamp.
OFFERING STORAGE IN BUCKETS OF 100TBS

➤ How customers deal with it?
   ➤ Create several accounts in one data centers.
   ➤ Some customers already use multiple accounts to partition storage across different regions.
      ➤ Thus, allow local access to data for their customers.
      ➤ Still not all customers.
   ➤ Future plans: increase amount of storage held within a given storage account.
CAP THEOREM

- Consistency and availability in face of specific failures.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Availability</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream layer</td>
<td>three replicas within different fault and upgrade domain.</td>
<td>append only system</td>
</tr>
<tr>
<td>Partition layer</td>
<td>reads from specific replicas</td>
<td>RangeParitions to another PS</td>
</tr>
</tbody>
</table>

Layers are co-designed to achieve the goal.

Network partitions issues that are likely to occur, TOR.
HIGH-PERFORMANCE DEBUG LOGGING

➤ Write logs to the local disks of the storage nodes.
➤ Provide grep-like utility to do a distributed search across all storage node logs.
➤ Two approaches:
  ➤ Reduce amount of logging.
  ➤ Optimize logging system, increase its performance, reduce disk space overhead by tokenizing and compressing output.
➤ Importance:
  ➤ Retention of many days of verbose debug logs across cluster.
  ➤ Investigate problems in production in detail.
  ➤ No need to reproduce problem or deploy special code.
PRESSURE POINT TESTING

- Programmable interface for main operations and points to create fault. Examples:
  1. Checkpoint a RangePartition
  2. Combine a set of RangePartition checkpoints
  3. Garbage collect a RangePartition
  4. Split / merge / load balance RangePartitions
  5. Erasure code or un-erasure code an extent
  6. Crash each type of server in a stamp
  7. Inject network latencies
  8. Inject disk latencies

- Main goal: find or reproduce issues from complex interactions that might have taken years to occur on their own.
Erasure Coding in Windows Azure Storage
Motivation for using erasure codes:
1. Data durability
2. Low storage cost

Reed Solomon?
1. Need to read from many fragments
2. Field size
3. Complicated computation

Another solution:
LRC, Local Reconstruction Codes
Not MDS

Other uses for erasure codes:
Reconstruction when reading data seems to take a lot of time (no failure occurred).
Definition and Goal

- **Tradeoff:**
  
  storage overhead (MDS, RS), performance (LRC)

- **Reconstruction cost:**
  
  Number of fragments required to reconstruct unavailable fragment.

- **Main goal:**
  
  Minimize reconstruction cost while still maintaining low storage overhead.
Example

- Given 6 fragments of data
  - Divide to 6/3 groups, each has its own local parity
  - Two more global parities
  - Finally, \( n = k + l + r = 6 + 2 + 2 = 8 \)
  - Capable of decoding arbitrary 3 failures.

Figure 1: A (6, 2, 2) LRC Example. (\( k = 6 \) data fragments, \( l = 2 \) local parities and \( r = 2 \) global parities.)
MDS?

Satisfies \( d = n - k = l + r = 2 + 2 = 4 \) ? Can decode arbitrary four failures.

No, not MDS 😞
However, ...

- We can still decode several case of 4 failures. Thus, defined:
  1. Theoretically decodable
  2. Theoretically non-decodable

- Challenge: code that can deal with any theoretically decodable case ⇒ Maximally recoverable property (MR)
Finally, definition

- Parameters:
  1. \( k \): data symbols
  2. \( l \): local parities
  3. \( r \): global parities

- Total number of fragments:
  \[ n = k + l + r \]

- Storage overhead:
  \[ \frac{n}{k} = \frac{k+l+r}{k} = 1 + \frac{l+r}{k} \]

Challenge: Determine set of equations to achieve MR property.
Determine Set of Equations

Set the parity functions to be:

\[ q_{x,0} = \alpha_0 x_0 + \alpha_1 x_1 + \alpha_2 x_2 \]
\[ q_{x,1} = \alpha_0^2 x_0 + \alpha_1^2 x_1 + \alpha_2^2 x_2 \]
\[ q_{x,2} = x_0 + x_1 + x_2 \]

\[ q_{y,0} = \beta_0 y_0 + \beta_1 y_1 + \beta_2 y_2 \]
\[ q_{y,1} = \beta_0^2 y_0 + \beta_1^2 y_1 + \beta_2^2 y_2 \]
\[ q_{y,2} = y_0 + y_1 + y_2 \]

\[ p_0 = q_{x,0} + q_{y,0}, \quad p_1 = q_{x,1} + q_{y,1}, \]
\[ p_x = q_{x,2}, \quad p_y = q_{y,2} \]
Determine Set of Equations

\[ G'' = \begin{pmatrix} \alpha_i & \beta_s \\ \alpha_i^2 & \beta_s^2 \end{pmatrix} \]
\[ \text{Det}(G'') = \alpha_i \beta_s (\beta_s - \alpha_i). \]

\[ p_y, p_x, \text{one failure in group } x, \text{ one failure in group } y \]

\[ G = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ \alpha_i & \alpha_j & \beta_s & \beta_t \\ \alpha_i^2 & \alpha_j^2 & \beta_s^2 & \beta_t^2 \end{pmatrix} \]
\[ \text{Det}(G) = (\alpha_j - \alpha_i)(\beta_t - \beta_s)(\alpha_i + \alpha_j - \beta_s - \beta_t). \]

\[ 4 \text{ failures equally divided between groups} \]

\[ G' = \begin{pmatrix} 1 & 1 & 0 \\ \alpha_i & \alpha_j & \beta_s \\ \alpha_i^2 & \alpha_j^2 & \beta_s^2 \end{pmatrix} \]
\[ \text{Det}(G') = \beta_s (\alpha_j - \alpha_i)(\beta_s - \alpha_j - \alpha_i). \]

\[ p_y, \text{2 failures in group } x, \text{ one in group } y \]
Determine Set of Equations

- Matrices should not be non-singular, it is enough to require:

\[
\begin{align*}
\alpha_i, \alpha_j, \beta_s, \beta_t & \neq 0 & (9) \\
\alpha_i, \alpha_j & \neq \beta_s, \beta_t & (10) \\
\alpha_i + \alpha_j & \neq \beta_s + \beta_t & (11)
\end{align*}
\]

- Assign \(\alpha\)'s and \(\beta\) the elements from a finite field \(GF(2^4)\), elements are represented by 4 bits. 
  \(\alpha\)'s are chosen among elements whose lower order 2 bits are zero. 
  \(\beta\)'s are chosen among elements whose higher order 2 bits are zero.
Which Parameters to Choose?

- Repair cost:
  Calculated by averaging across all the fragments.
  For example (6,2,2) LRC:
  1. 3 fragments to repair any of the 6 data fragments, and the 2 local parities
  2. 6 fragments to repair the 2 global parities
  \[ C = \frac{3 \cdot 8 + 6 \cdot 2}{10} = 3.6 \]

- Storage overhead
- Each fragment should be placed in different fault domain.
- (6,3) RS was replaced with (12,4,2) LRC
  Reconstruction cost is reduced from 6 to 3.
  Storage overhead 1.5x
- Another option, (12,2,2) LRC
  Storage overhead reduced from 1.5x to 1.33x.
Based on: