HATS

**Hats stands for Highly Available Table Store.**

**HATS is an implementation of clever and simple protocol that aims at providing high availability for cloud based key value stores by building resiliency against a single data center unavailability. While the protocol itself can be implemented for any key value store that provides an expected interface, this implementation targets Azure Table Storage.**

**The implementation is a library that provides azure TableStore like APIs while also providing resiliency to a single datacenter unavailability. It achieves this by writing to multiple azure tables in background – yet maintaining consistency across them. HATS uses master-master strategy between stores and achieves consistency using 2 key concepts.**

1. **Multi-version values per key -> Store multiple versions of values for a given key**
2. **Quorum consensus -> Quorum of stores has to be available and they also have to agree on latest value.**

**Multi-version values per key**

**When an update needs to be made to a key, a new row is added to the table instead of overwriting the old value. Here is an illustration. Note that David has multiple versions stored while Seth and Ravi have only version.**

|  |  |  |
| --- | --- | --- |
| **Key** | **Version** | **Value** |
| Ravi | 1 | 23 |
| David | 1 | 24 |
| Seth | 1 | 28 |
| David | 2 | 55 |
| David | 3 | 76 |

**Quorum consensus**

Quorum = More than half.

Quorum consensus is different from quorum availability. Quorums consensus not only requires quorum availability but also requires that they quorum agreement on latest version. Let us define the term ‘Latest Committed version’ as below.

Latest committed version (aka LCV) **= common latest version that is present on quorum. Here are few examples.**

|  |  |  |
| --- | --- | --- |
| **Store 1** | **Store 2** | **Store 3** |
| 1 | 1 | 1 |
| 2 | 2 | 2 |

LCV = 2

|  |  |  |
| --- | --- | --- |
| **Store 1** | **Store 2** | **Store 3** |
| 1 | 1 | 1 |
| 2 | 2 | Write failed |

LCV = 2

|  |  |  |
| --- | --- | --- |
| **Store 1** | **Store 2** | **Store 3** |
| 1 | 1 | 1 |
| 2 | 2 | Write failed |
| Write failed | 3 | Write failed |

LCV = 2

Note -> Latest committed version may not be the latest version on any single replica.

|  |  |  |
| --- | --- | --- |
| **Store 1** | **Store 2** | **Store 3** |
| 1 | 1 | 1 |
| 2 | 2 | Write failed |
| Available | Available | Unavailable |

LCV = 2

Note -> We can still determine the LCV even if a store is unavailable if quorums agree on latest version.

|  |  |  |
| --- | --- | --- |
| **Store 1** | **Store 2** | **Store 3** |
| 1 | 1 | 1 |
| 2 | 2 | Write failed |
| Unavailable | Available | Available |

LCV = Indeterministic

Note -> We can’t determine LCV since quorum does not agree on latest version.

**Azure Table Illustration**

Azure table APIs use partition key and row Key in read/write APIs. HATS will also have similar APIs. However while storing into azure table, HATS appends the version to row key as shown below. For example if ‘David’ is the key that is passed into the API then HATS, the version number is appended (\_1, \_2, \_3) to the row Key before storing it into azure table. Why we need to append the version to the rowkey is explained in ‘racy Write’ paragraph below. The need for ‘TransactionId’ and ‘Deleted’ will be explained later.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PartitionKey** | **RowKey** | **TransactionId** | **Deleted** | **CustomValueColum(s)** |
| 100 | David\_1 | Guid1 | No | X |
| 100 | David\_2 | Guid2 | No | Y |
| 100 | David\_3 | Gud3 | No | Z |

**Read**

When all stores are available:

* Get all the version for a given key, do this on all stores.
* Find the biggest version across all stores. Check whether quorums agree on this. If yes this the latest version – if not previous version is the latest version (to see why previous version is the latest version, see the algorithm for write below).
* Find the biggest version that is common across all stores. That is the LCV.

When all stores are not available but quorum is available.

* Get all the version for a given key, do this on all stores.
* Find the biggest version across all stores. Check whether quorums agree on this. If yes this the latest version – if not we are can’t determine what is LCV – return an error.

**Write operation**

Write operation.

* Find the LCV by doing a Read first (as described above)
* Determine the new LCV to write by incrementing the LCV by 1
* If there are some uncommitted version rows > LCV, then we may have to delete them before writing. We can only delete them if the current Read could visit all the stores and all the stores have dirty rows. If only few of the stores have dirty rows then we can simply try to write without deleting (irrespective of whether few = quorum).
* Try writing the new LCV to all stores in parallel.
* If quorum of them succeed then report success with new LCV as HATS ETag (more below).

**HATS Etag**

Since HATS uses multiple azure table, it needs to come up with a way to use a common ETag based write mechanism across all stores and expose this ETag as part of the HATS API. HATS uses the version number as the ETag. HATS Etags are monotonically increasing numbers starting with 1.

**Conditional Write**

Conditional Writes are done with HATS ETag. If an HATS ETag is provided during write (or delete) we need to make sure that HATS ETag matches with LCV before proceeding with writes.

**Racy Write**

If we have racy writes with ETag then only one of them should succeed and the other one should fail with ‘ETag Mismatch’ exception. Let us say ‘x’ and ‘y’ are the 2 racy transactions on the same key ‘David’ and same ETag = 1. Note that each version update is implemented by adding a new row (Calling TableOperation.Insert) and the underlying store will only allow one insert for a given partition-key/row-key combination. This means that at the end of ‘x’ and ‘y’ trying once, we will likely have one (say x) succeeding on 2 stores and other (say y) succeeding on the 3rd store. In this case ‘x’ will return success (because quorum was achieved) and y will fail with ‘ETagMisMatch’ exception.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Store 1** | | | | |
| **PartitionKey** | **RowKey** | **TransactionId** | **Deleted** | **CustomValueColum(s)** |
| 100 | David\_1 | Guid1 | No | 10 |
| 100 | David\_2 | Guid2 | No | 20 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Store 2** | | | | |
| **PartitionKey** | **RowKey** | **TransactionId** | **Deleted** | **CustomValueColum(s)** |
| 100 | David\_1 | Guid1 | No | 10 |
| 100 | David\_2 | Guid2 | No | 20 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Store 3** | | | | |
| **PartitionKey** | **RowKey** | **TransactionId** | **Deleted** | **CustomValueColum(s)** |
| 100 | David\_1 | Guid1 | No | 10 |
| 100 | David\_2 | Guid3 | No | 25 (Written by Y) |

**Transaction ID**

Examine the above tables carefully. All the 3 stores have version 2 for key David – however store 3 has the incorrect value. If a read operation comes now, it is incorrect to return 25 as the new value though it is written as version 2. We should find a way to return 20 as the correct answer. To achieve this we need to be able to attach a ‘TransactionId’to each write operation that comes into HATS. So when a write operation comes into HATS, a unique ‘TransactionId’ is generated and is written as a HATS specific column called ‘TransactionId’ into Table in every store. You can see that store1 and store2 have Guid2 as the ‘TransactionId’ where as Store3 has Guid3 as the ‘TransactionId’ for version 2. Now when determining LCV, HATS should not only compare the versions but also the ‘TransactionId’. If the quorum does not agree on both version and ‘TransactionId’– then we don’t have a match. Following is an illustration of how 3 HATS instances which all have various availability to different stores end up determining LCV in this state.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PartitionKey** | **Store1** | **Store2** | **Store3** | **LCV** |
| HATS instance123 | Available | Available | Available | 2 |
| HATS instance12 | Available | Available | Unavailable | 2 |
| HATS instance13 | Available | Unavailable | Available | Indeterministic |

**AVOIDING DEADLOCK**

It is possible that multiple competing writers can all fail – this means that latest version on each store is in uncommitted state. This can lead to read deadlock or write deadlock. Here is how we recover from such situation and make progress.

Race – but all three are available :

Just chose one of them as winner (based on timestamp or transactionId)

Race – but only 2 out of 3 are available:

This is the tricky. Here are some options that can be implemented out of the box and app can chose the one it likes

* Option 1 : Don’t do anything but implement a work around – this may sound lame – but I do have a scenario where this option makes sense. For example app logic needs only one out of many equivalent keys to be available. So if a key ends up in race- the app logic simply tries some other key and succeeds.
* Option 2 -> The app receives NoConsenusException with details of the exception object providing 2 values that are currently in 2 stores – The app logic can now commit one of those values with InsertOrReplace with ETag = \*
* Option 3

1. Where writer succeeds partially but can’t reach quorum, it asynchronously marks the succeeded store as ‘indoubt’
2. When a writer succeeds in quorum but fails on one, then it marks the succeeded  as ‘committed’
3. When a reader succeeds in quorum and finds one of the stores out of sync, then it marks the one with quorum as ‘committed’ and fixes up the other.

With above 3 in place – when in a racy situation we can chose the winner based on timestamp/transaction id given  none of the 2 available are in committed state, but both of are in ‘indoubt’ state. Well – this is theoretically not strong consistency given there is a still a theoretical chance that a reader succeeded in reading a value that was written by a writer that thinks it failed – and eventually a different winner was chosen – but for this to occur the reader must also have failed to mark as committed.   We can also solidify this even further by delaying the ‘winner chosing’ algorithm until it is a minute or 2 after the racy write occurred – and also by making sure that if the ‘sneak in reader’ tries to write back it fails (there are ideas in my mind how we can achieve this – but I will save that for later).

**Fix up**

Let us say we have a key that gets updated on the last day of the month – but read many times in a day. While writing version 3 the store 1 failed – however quorum was achieved hence operation succeeded. Now store 1 came back up next day however store 2 went down for a week right after. In this case the, though more than 2 stores are available, we cannot determine the LCV for the period when store 2 was down. Because the available quorum cannot agree on latest value. Since writes depend on reads, they won’t succeed either. This won’t be rectified until store2 comes back up. (Example below).

|  |  |  |
| --- | --- | --- |
| **Store 1** | **Store 2** | **Store 3** |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| Write failed | 3 | 3 |
| Available | Unavailable for a week | Available |

This situation can be improved by doing a ‘fixup Write’ to store 2 during Read – i.e. if we find that we can determine a quorum but we have a store that is not in consensus with the LCV, then we force write the LCV to that store. If we have a key that is frequently written, then this lazy fix up feature is not necessary.

**Delete**

Delete simply inserts another row while also setting column ‘Deleted’ to true. Delete also should make sure the HATS ETag match with LCV before proceeding with Delete (see conditional write above).

**Purging Stale entries**

Stale entries can be deleted in a batch. This can be done asynchronously after a commit or this can be done as a admin activity.

**HATS API**

HATS API will be similar to Table APIs.

It will provide features that azure Table already provides like EntityGroup transaction, 5 flavors of write (insert/replace/merge/InsertOrMerge/InsertOrReplace), multi column values etc. It is unclear at this point whether we can provide 100% features that azure table has – but is clear we can do majority of them. Keeping API surface similar to azure TableStore will make HATS attractive all teams across Microsoft as well as other customer outside.

ReadDirty(bool onlyWhenNoQuorumAvailable)

When onlyWhenNoQuorumAvailable is true, HATS reads latest value when no quorum is available - if quorum is available then returns quorum value.

When onlyWhenNoQuorumAvailable is false, HATS reads the latest value ignoring the state of quorum consensus.

Please look at API usage for current state of APIs.

**DATA CORRUPTION RECOVERY - AS A FEATURE**

A buggy application can sometime corrupt data. Because of this reason applications take backup so that they can do point in time recovery. With HATS we get this feature for free. This is possible because HATS never overwrites old data. Let us imagine an application started corrupting the data on Jan 10th. Once you discover this, you could initialize HATS to Jan 10th – what this would make HATS do is to only read entries that were written before Jan 10th  and hence read uncorrupted data. Meanwhile all new entries that were made after Jan 10th could be deleted out of band.

**PERMANENT DISASTER**

If a datacenter that has gone down completely is never coming back, then we need special handling to build a new datacenter with all the keys that exist in other datacenters. This needs be done outside of regular read/write operations – because regular HATS API has no way of knowing which datacenter is not coming back up. For this reason, an API like this will be useful.

BuidNewCloudTable(existing tables, new table, bool strongConsistency)

Following table explains the logic that will be used for each key to build the new table.

|  |  |  |
| --- | --- | --- |
| **Existing Tables are in quorum for the given key** | **Quorum is the latest value** | **Action** |
| Yes | Yes | Write the quorum/latest value |
| Yes | No | If strongConsistency is true then write quorum value, if not write the latest value into new table |
| No | NA | If strongConsistency is true then delete the key from existing tables, if not write the latest value |

**API Usage**

HatsTable hatsTable = null;

public BasicTests()

{

this.hatsTable = new HatsTable(new List<CloudTable>() { Table1, Table2, Table3 });

}

[TestMethod]

public async Task BasicRetrieve()

{

DynamicTableEntity entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve("Unknown" + Guid.NewGuid().ToString(), Guid.NewGuid().ToString()));

Debug.Assert(entityReturned == null);

}

[TestMethod]

public async Task BasicInsert()

{

DynamicTableEntity entity = DynamicTableEntityExtensions.GetRandomEntity();

DynamicTableEntity entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Insert(entity));

Debug.Assert(entity.IsSameAs(entityReturned));

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve(entity.PartitionKey, entity.RowKey));

Debug.Assert(entity.IsSameAs(entityReturned));

}

[TestMethod]

public async Task BasicInsertReplace()

{

DynamicTableEntity entity = DynamicTableEntityExtensions.GetRandomEntity();

DynamicTableEntity entityReturned = await hatsTable.ExecuteAsync(HatsOperation.InsertOrReplace(entity));

Debug.Assert(entity.IsSameAs(entityReturned));

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve(entity.PartitionKey, entity.RowKey));

Debug.Assert(entity.IsSameAs(entityReturned));

}

[TestMethod]

public async Task BasicInsertOrMerge()

{

DynamicTableEntity entity = DynamicTableEntityExtensions.GetRandomEntity();

DynamicTableEntity entityReturned = await hatsTable.ExecuteAsync(HatsOperation.InsertOrMerge(entity));

Debug.Assert(entity.IsSameAs(entityReturned));

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve(entity.PartitionKey, entity.RowKey));

Debug.Assert(entity.IsSameAs(entityReturned));

}

[TestMethod]

public async Task BasicReplace()

{

DynamicTableEntity entity = DynamicTableEntityExtensions.GetRandomEntity();

DynamicTableEntity entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Insert(entity));

entity.Properties[entity.Properties.First().Key] = new EntityProperty(Guid.NewGuid().ToString());

entity.ETag = entityReturned.ETag;

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Replace(entity));

Debug.Assert(entity.IsSameAs(entityReturned));

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve(entity.PartitionKey, entity.RowKey));

Debug.Assert(entity.IsSameAs(entityReturned));

}

[TestMethod]

public async Task BasicMerge()

{

DynamicTableEntity entity = DynamicTableEntityExtensions.GetRandomEntity();

const string keyToRetain = "keyToRetain";

const string valueToRetain = "valueToRetain";

entity.Properties.Add(keyToRetain, new EntityProperty(valueToRetain));

DynamicTableEntity entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Insert(entity));

// Add a property, cahnge a property, remove a property - make sure all these properties exist eventually

entity.Properties.Remove(keyToRetain);

entity.Properties[entity.Properties.First().Key] = new EntityProperty(Guid.NewGuid().ToString());

entity.Properties.Add("NewProperty", new EntityProperty(Guid.NewGuid().ToString()));

entity.ETag = entityReturned.ETag;

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Merge(entity));

entity.Properties.Add(keyToRetain, new EntityProperty(valueToRetain));

Debug.Assert(entity.IsSameAs(entityReturned));

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve(entity.PartitionKey, entity.RowKey));

Debug.Assert(entity.IsSameAs(entityReturned));

}

[TestMethod]

public async Task BasicDelete()

{

DynamicTableEntity entity = DynamicTableEntityExtensions.GetRandomEntity();

DynamicTableEntity entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Insert(entity));

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Delete(entityReturned));

Debug.Assert(entity.IsSameAs(entityReturned));

Debug.Assert(entityReturned.ETag == null);

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve(entity.PartitionKey, entity.RowKey));

Debug.Assert(entityReturned == null);

}

[TestMethod]

public async Task BasicGetAndSet()

{

DynamicTableEntity entity = DynamicTableEntityExtensions.GetRandomEntity();

DynamicTableEntity entityReturned = await hatsTable.ExecuteAsync(HatsOperation.InsertOrReplace(entity));

Debug.Assert(entity.IsSameAs(entityReturned));

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve(entity.PartitionKey, entity.RowKey));

Debug.Assert(entity.IsSameAs(entityReturned));

entity.Properties[entity.Properties.First().Key] = new EntityProperty(Guid.NewGuid().ToString());

entity.ETag = entityReturned.ETag;

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.InsertOrReplace(entity));

Debug.Assert(entity.IsSameAs(entityReturned));

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.Retrieve(entity.PartitionKey, entity.RowKey));

Debug.Assert(entity.IsSameAs(entityReturned));

entity.Properties[entity.Properties.First().Key] = new EntityProperty(Guid.NewGuid().ToString());

entity.ETag = entityReturned.ETag;

entityReturned = await hatsTable.ExecuteAsync(HatsOperation.InsertOrReplace(entity));

Debug.Assert(entity.IsSameAs(entityReturned));

}