SFDC : HBase 2/Phoenix 5 in-place rolling upgrade

Written by: Viraj Jasani and Andrew Purtell

Introduction

What is Apache HBase? HBase stores rows of data in *tables*. Tables can be grouped into *namespaces*. A table can belong to only one namespace at a time. Tables are split into groups of lexicographically adjacent rows. These groups are called *regions*. By lexicographically adjacent we mean all rows in the table that sort between the region's start row key and end row key are stored in the same region. Regions are distributed across the cluster, hosted, and made available to clients by *regionserver* processes. Regions are the physical mechanism used to shard and distribute the write and query load across the fleet of regionservers. Regions are non-overlapping. A single row key belongs to exactly one region at any point in time. Together with the special META table, a table's regions effectively form a b-tree for the purposes of locating a row within a table. HBase is properly described as a *distributed ordered tree*.

What is Apache Phoenix? Phoenix is a relational toolkit and SQL parser built on top of HBase, providing two key components. The first is a client library and a JDBC driver. The driver bundles JDBC APIs, a SQL parser, a query planner, and a typed data encoding library. The second is server-side integrated HBase Coprocessors. Leveraging the powerful Coprocessor API of HBase, Phoenix utilizes these extension points for high performance integration. This integration enables Phoenix to offer structured metadata in the SQL model, optimize queries by leveraging push-down predicates and statistics collection, and implement strongly consistent secondary indexes, as well as hierarchical and tenant-specific views.

In terms of compatibility, it's important to note that Phoenix 4.16 is designed to work with HBase 1.6. Whereas Phoenix 5.1 is compatible with recent versions of HBase 2.

Why move to HBase 2?

Here are some of the notable new features contributed to HBase 2.x release line:

1. AMv2 (AssignmentManagerV2) HBASE-14350: What regionserver among the fleet is given responsibility for serving a given region is determined by the currently active HBase master process. The master's AssignmentManager (*or AM*) orchestrates the transfer of region serving responsibility from one server to another. In HBase 1, ZooKeeper is used for state synchronization between the active master and regionservers. A comprehensive redesign of the Assignment Manager was undertaken in HBase 2. The dependency on ZooKeeper for orchestrating region transitions has been eliminated. The logic for assignment, crash, split and merge handling has been recast as *procedures* (or Pv2, see HBASE-13202 and its detailed overview).
2. AsyncWAL HBASE-14790 An asynchronous HDFS output stream implementation: An HDFS cluster consists of a redundant set of NameNodes, a coordinating service that manages the file system namespace and regulates access to files by clients; and typically an order of magnitude or more instances of another redundant set of DataNodes, which manage locally attached storage devices. Internally, a file is split into one or more blocks and multiple replicas of these blocks are distributed over the fleet of DataNodes in a fault-domain-aware manner. The durability and availability of HBase data files are guaranteed by HDFS's data availability model. WAL (or write-ahead-log) is a standard way to ensure data integrity and reliability for HBase data. Any changes made on the given table are first logged in an append-only WAL files. The advanced implementation in HBASE-14790 excels in fanning out WAL writes across multiple DataNodes while exclusively accommodating the writing of only a single block to the file. This solution not only achieves

reduced latency but also promptly detects and responds to DataNode connection issues, ensuring a high degree of fail-fast capability.

3. In-memory compaction HBASE-14918: HBase region consists of one or more stores (*or HStore*). Each store represents one Column Family (*or CF*). A Column Family is a logical grouping of columns. Each Column Family can contain an arbitrary number of columns, and each column is identified by a unique qualifier within its column family. Each HBase store consists of one memstore and one or more store files (*or HFile*). When a write operation is initiated by a client on the specified table, the new or updated data is first written to the memstore — an in-memory data structure backed by ConcurrentSkipListMap — to retain the sorted order of data in memory according to the row-key, column-family, column-qualifier and timestamp values. When the data stored by memstore reaches configurable threshold, the data is flushed to the disk as persistent store file. Every flush operation creates a new store file. HBase compaction is a process of combining multiple store files within the given region into one big store file, this operation (Compactor) helps reduce number of disk seek operations required to read the data. In-memory compaction involves the process of compacting the data while it is still held in the memstore. This enhanced solution provides improved performance by reducing disk operations during flush and compaction.
4. Alternative Block cache implementations: Each HFile consists of a series of blocks (*or HFileBlock*). A block is the smallest unit of data that can be read from an HFile. When a block is read from the filesystem (HDFS), it is cached in BlockCache. Frequent access to the rows in a block causes the block to be kept in cache, improving read performance.
    a. AdaptiveLRU HBASE-23887: HBase BlockCache p99 read latency improves by 3x.
    b. TinyLFU HBASE-15560: TinyLfuBlockCache records the frequency in a counting sketch, ages periodically by halving the counters, and orders entries by SLRU. An entry is discarded by comparing the frequency of the new arrival to the SLRU's victim, and keeping the one with the highest frequency.
5. End-to-end offheap read path HBASE-11425: In Java, the heap is the region of memory managed by the Java Virtual Machine (*or JVM*) where objects are allocated and deallocated. Off-heap memory, on the other hand, is memory that is allocated outside of this managed heap, often directly by the application or through native libraries. Off-heap memory has some notable advantages over on-heap memory: No contributions to JVM's garbage collection (*or GC*), reduced heap fragmentation, larger memory capacity. The workflow provided by HBASE-11425 from reading blocks to sending cells to client does its best to avoid on-heap memory allocations reducing the amount of work the GC has to do.
6. End-to-end offheap write path HBASE-15179: In the write path, the request packet received from client will be read in on a pre-allocated offheap buffer and retained offheap until those cells are successfully persisted to the WAL and Memstore. The memory data structure in Memstore does not directly store the cell memory, but references the cells encoded in the offheap buffers.
7. Serial replication HBASE-20046: HBase replication is a feature that allows data to be asynchronously (by default) copied (or replicated) from one HBase cluster (*or source cluster*) to another HBase cluster (*or sink cluster*). This feature is designed to provide data redundancy, disaster recovery, and remote access to data. HBase replication is often used in scenarios where high availability, data backup, and data distribution across geographically distributed clusters are important requirements. When serial replication provided by HBASE-20046 is enabled, the chronological sequence of log propagation to the replication sink cluster maintains strict correspondence with the order in which incoming client requests are received by the source cluster.
8. FileSystem quotas HBASE-16961: It is possible to establish allocation limits for both tables and namespaces. The computation of region sizes is undertaken by the regionservers, and subsequently relayed to the active master. The master daemon evaluates the dimensions of individual regions, aggregating these measurements to derive comprehensive table and namespace sizes. This analysis forms the basis for determining the specified quota thresholds.
9. Direct insert HFiles and persist in-memory HFiles tracking HBASE-24749: New store engine to provide lower write latency and high throughput for HBase flush and compaction when HFiles are deployed to S3.
10. OpenTelemetry based tracing HBASE-22120: Tracing in distributed systems refers to the practice of capturing and tracking the flow of requests or transactions as they move through various components and services of the distributed systems. The goal of tracing is to provide visibility into the interactions and behaviors of the different components that collaborate to fulfill a client request, and to help diagnose performance issues, bottlenecks, and errors that can arise in

the complex systems. HBASE-22120 replaces HTrace with OpenTelemetry for the tracing.

11. Remove dependency on zookeeper to store meta location HBASE-26193: Apache ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. By default, HBase stores the meta table location in Zookeeper. When meta region transitions from one regionserver to another, the location of meta changes in Zookeeper. HBase clients can put significant pressure on Zookeeper when it comes to searching for the location of meta table. HBASE-26193 introduces Master local region to store the location of meta region and hence no longer depends on Zookeeper to store and retrieve meta location.

HBase 1 is not compatible with Java 11+ and Hadoop 3+ versions. HBase 2 release line has been active since 2018.

Scope of the upgrade:

In distributed systems, compatibility between client and server is critical so both can understand each other's communication protocols, data formats, features, and other requirements. For instance, protocol compatibility means that the client and server must use compatible communication protocols to exchange data. This includes agreement on the methods, headers, and message formats used in the communication. As our customers expect a high level of service availability (*SLA*), our primary objective is to achieve in-place rolling upgrade without downtime. It also requires ensuring that clients maintain protocol compatibility with servers both during and after the entire upgrade procedure.

Prior to the upgrade, our overall system consisted of Phoenix 4.16 combined with HBase 1.6 at the client side, interacting with Phoenix 4.16 server running on HBase 1.6 clusters. For the purpose of the upgrade, using Phoenix 4.16 with HBase 1.6 at the client is protocol compatible with Phoenix 5.1 running on HBase 2.4 at the server side.

Furthermore, the disparity between HBase 2 and HBase 1 in terms of region assignment management requires a different approach for performing a rolling upgrade. Conventional rolling restarts of master components (active and backups) followed by regionservers are insufficient in achieving an upgrade without experiencing downtime.

Prerequisite features:

Before we explore the in-place rolling upgrade, it would be good to emphasize important features that greatly help with the overall upgrade process.

RSGROUPS

HBase and Phoenix include many system tables designed to store the metadata related to the user tables. For instance, hbase:meta table stores the status and destination regionserver details, among other information, for every region of each table within the cluster.

HBase system tables:

- hbase:meta
- hbase:namespace
- Optional tables based on various features enabled (hbase:rsgroup, hbase:quota, hbase:slowlog, hbase:acl etc)

Phoenix system tables:

- SYSTEM.CATALOG
- SYSTEM.CHILD_LINK
- SYSTEM.STATS

- SYSTEM.SEQUENCE
- SYSTEM.TASK
- SYSTEM.FUNCTION
- SYSTEM.LOG
- SYSTEM.MUTEX

In order to maintain the high availability for the overall cluster, system table regions should be provided with higher *availability priority* than the other regions. This is also an important consideration for HBase 2 upgrade.

What is RSGroup? Regionserver grouping (or rsgroup) is an advanced feature for partitioning regionservers into distinctive groups for strict isolation.

Advantages of the RSGroup Feature:

By allocating only a few dedicated regionservers to the system rsgroup, we can restrict the target regionservers for assigning system table regions. This helps minimize the frequency of the system table region movements during various patching and deployment activities. By doing so, the exposure to the availability issues related to the region movements (caused by the auto load balancing as well as operator interventions - patching/deployments) for the system tables are greatly reduced, thereby reducing the possibility of prolonged availability loss for the overall cluster.

By utilizing rsgroup, we can target upgrade of system rsgroup that consists of only system table regions, before upgrading user rsgroup regionservers that only make user table regions available.

ZOOKEEPER-LESS REGION ASSIGNMENT:

The rationale behind our interest in ZooKeeper-less (*or zk-less*) region assignment originates from the different assignment mechanisms used in HBase 1 and HBase 2. Due to these discrepancies, we found it necessary to utilize zk-less assignment within HBase 1 as an interim measure to facilitate the transition from one version to another. This transitional approach introduced many complexities and challenges.

We would cover more about transitioning from zk to zk-less region assignment on HBase 1 in the second part of this blog post series.

Prerequisite cluster checks:

Before proceeding with the in-place rolling upgrade, it is essential to ensure that the HBase 1.x cluster is in a clean state. The clean state is characterized by the following conditions:

- RITs (Regions In Transition): 0, indicating that there are no regions in transition or under movement. The process of opening a region or closing a region is transitional by nature. Therefore we term regions currently in the process of opening or closing as *regions in transition*, or RIT.
- hbck (HBase fsck) inconsistencies: 0, meaning there are no inconsistencies detected by the hbck tool, which validates data integrity for HBase.
- Clean fsck (File System Check) report: Ensuring that the HBase file system is in a healthy and consistent state. HDFS fsck helps operators identify issues with the HDFS, such as missing blocks, under-replicated blocks, over-replicated blocks, corrupt blocks etc.
- Passing of Synthetic test checks: HBase Canary or Synthetic Tests is a type of diagnostic tool that is designed to detect issues related to data availability and health of every region of each table within the cluster.

Once all these conditions are met, the cluster will be considered in a clean state and ready for the in-place rolling upgrade.

Upgrade in detail

We leverage a combination of open source technologies, including Apache Ambari and Bigtop, supplemented by our proprietary tooling, to establish robust CI/CD (Continuous Integration/Continuous Deployment) capabilities within our first-party datacenters. Similarly, we use Kubernetes with a specific workload resource called StatefulSets, to ensure similar capabilities within our public cloud infrastructure (more details here). We have carefully developed custom rolling upgrade workflows within our tooling, enabling us to achieve in-place rolling upgrades without any disruption in first party datacenters as well as public cloud infrastructures.

Upgrade steps:

If HBase 1 is already operating with Zookeeper-less region assignment mode, step 3 to 6 can be skipped.

STEP 1.

This step is same as the prerequisite step defined in the previous blog post i.e. ensure the cluster is in healthy state:

- No RITs
- No missing/under-replicated blocks
- No synth test failures
- Run hbck and fix any reported inconsistencies
- Ensure regions are well balanced across the fleet of regionservers. Run the balancer if required.

Also be sure to disable any auto-start capabilities of the deployment tooling (f.e. ambari or kubernetes).

STEP 2.

HBase background housekeeping is unwanted during the upgrade procedure. Disable the balancer, the region normalizer, and region split and related janitor operations. This step provides very important stability for the remaining upgrade steps and failing to make these changes is unnecessarily risky. Throughout this upgrade, we are going to deal with different region assignment mechanisms between HBase 1 and HBase 2. Operations like region split and merge are partially incompatible when HBase master and regionservers are operating with a mixed versions of HBase 1 and 2.

```
> balance_switch false
> normalizer_switch false
> splitormerge_switch 'SPLIT', false
> catalogjanitor_switch false
> splitormerge_switch 'MERGE', false
```

Check for RITs and manually resolve if any. We might encounter region split RITs if certain region is in the middle of splitting and we disable split. If it happens, reenable split, restart active master and wait until region going through split transition completes. Disable split again.

STEP 3.

Update hbase-site configs:

```
<property>
  <name>hbase.assignment.usezk</name>
  <value>true</value>
</property>
<property>
  <name>hbase.assignment.usezk.migrating</name>
  <value>true</value>
</property>
```

The intention behind making this change is to signal the AssignmentManager to get ready for the ZooKeeper-less region assignment mode. With this migrating state, region state updates are additionally stored in the meta table by AssignmentManager.

STEP 4.

The active HBase master is the primary instance responsible for managing and coordinating various activities within the cluster. It handles tasks such as region assignment, load balancing, disaster recovery for any regionserver and other administrative operations. The standby HBase Master is a backup instance of the HBase master service that is ready to take over in case the active master becomes unavailable due to any reasons. In a production environment, it's advisable to deploy a minimum of three master service components to ensure the presence of at least two standby masters for the backup.

Restart all HBase masters. For this step, it would be beneficial to restart all standby masters followed by active master restart, or restart all masters simultaneously. This would ensure that active master failover happens only once. This is an important consideration to reduce the complexity of the overall upgrade procedure, considering the different region assignment mechanisms involved in each step of master restarts.

STEP 5.

Update hbase-site configs:

```
<property>
  <name>hbase.assignment.usezk</name>
  <value>false</value>
</property>
```

The config change implies that we are ready to completely migrate to zookeeper-less region assignment on hbase 1 servers.

STEP 6.

Critical order of rolling restarts (system RSGroup → default RSGroup → masters): Rolling restart all regionservers first. We need to ensure no master is restarted with updated config. Disable auto-restart if required. Ensure system RSGroup regionservers are restarted before default RSGroup servers. This is required because the deployment of user tables in the default RSGroup depends on the health and functionality of the system RSGroup serving the system tables.

Without this strict order, we could potentially observe many regions-in-transition (RITs) in the cluster as the network calls from master to regionservers are not compatible in this state.

Rolling restart all masters. Nice to follow the order of restarting all standby masters before restarting active master so that

active master fails-over only once.

Once done, we are on zookeeper-less region assignment. This means we are ready to perform hbase 2.x upgrade.

STEP 7.

Initiate HBase 2 rolling upgrade:

The classpath is a system parameter or environment variable that defines the location where the JVM can find the compiled bytecode i.e. classes and resources that are required to run the application. HBASE_CLASSPATH is used by HBase to include all necessary HBase libraries and dependencies required for various HBase operations.

For users who are already using Phoenix, HBASE_CLASSPATH would have previously included Phoenix server jar location. For the purpose of this deployment, update HBASE_CLASSPATH to include Phoenix 5.1 server jar location.

Critical order of rolling restarts (system RSGroup → default RSGroup → masters): Similar to above step, follow the same order of rolling restart starting with the system RSGroup regionservers, followed by the default RSGroup regionservers, followed by masters.

Rolling restart all regionservers:

First regionserver that restarts must belong to system RSGroup. The first regionserver that comes online on hbase 2 must online the regions of the system tables (HBASE-17931).

Once the key system tables come online on hbase 2 regionserver, let rolling restart move forward and bring all regionservers to hbase 2.

Rolling restart all masters:

It is very important that all standby masters are restarted and upgraded to hbase 2 before active master.
Restart active master (which is on 1.6.0 version so far) and active master fails over to any of standby masters. When it does, it tries to follow series of steps required to become active and bypass some of the steps specifically for this upgrade, because it would not be able to find missing CFs (table and repl_barrier) on meta table.
Active master tries to add these missing CFs in meta after becoming active. It submits ModifyTableProcedure to perform this task. However, since few of tasks were bypassed, the active master aborts itself (please check release notes on HBASE-25902 for detailed info).
Due to this abort, new master becomes active and this time, the process goes smooth
Because of the above mentioned steps, it is really critical that we upgrade all standby masters to 2.4 before upgrading 1.6 active master. We don't want to keep switching from 2.4 to 1.6 and from 1.6 to 2.4 and hence, restart all standby to bring them on 2.4 before restarting active master. As an alternative, we can stop all masters running old version (1.6) simultaneously and then start them all with hbase 2.4 simultaneously.

Verify the active master daemon is running on hbase 2.

STEP 8.

Re-enable balancer, normalizer and region split/merge:

```
> balance_switch true
> normalizer_switch true
> splitormerge_switch 'SPLIT', true
```

```
> catalogjanitor_switch true
> splitormerge_switch 'MERGE', true
```

Note: Throughout this entire upgrade procedure, it is crucial to execute rolling restarts of regionservers with maximum care. This involves cautiously relocating all regions from the designated host using the region_mover tool, and thereby allowing the active master to handle region assignments. Once the regionserver is brought back online, the regions can be reloaded back onto the host using the same region_mover tool. By adhering to this approach, we can efficiently prevent the ServerCrashProcedure (SCP) from having to manage region assignments. This precaution holds significant importance during the transitional phase of the upgrade, wherein the master operates on hbase 1 while regionservers operate in mixed mode, incorporating both hbase 1 and 2 versions. It has been observed that the SCP on the hbase 1 master is particularly susceptible to fragility in this transitional phase, which could potentially result in regions lingering in the Region In Transition (RIT) state for prolonged periods or, in some severe cases, remaining offline throughout the remainder of the upgrade procedure. Hence, following this careful rolling restart strategy is very important to ensuring a smooth and successful upgrade.

In the next part of the blog post series, we will cover the array of challenges we encountered during our upgrades and provide the strategies employed to successfully surmount them.

Learnings

Let's dive deep into how we tackled some of the challenges. We have contributed the fixes back to the opensource HBase 2.x and Phoenix 5.x codebase.

1) HBASE-26021 HBase 1.7 to 2.4 upgrade issue due to incompatible deserialization

Those who are using HBase 1.7.0 in production and are planning to upgrade to HBase 2.x versions are going to encounter this issue of incompatible deserialization described in the issue tracked by HBASE-26021. The incompatible deserialization of table descriptiors issue is fixed by reverting HBASE-7767 for branch-1 and by fixing the ConnectionRegistry accordingly https://github.com/apache/hbase/pull/3435

We faced this problem because our production version during that period was more aligned with 1.7 release than with 1.6 release. For those using HBase 1.7.0 within their production environment, it is advised to initially undertake an upgrade to version 1.7.1 or 1.7.2 prior to initiating the migration to HBase 2.x. If anyone is using HBase 1.6.x in production, migrating to HBase 2.x can be accomplished without performing any additional release on 1.x release line.

2) HBASE-26433 Rollback from Zookeeper-less to Zookeeper-based region assignment could produce inconsistent state - doubly assigned regions

As part of the migration from using Zookeeper for the region assignment workflows to not using it, we go through two-phase process. First phase requires enabling config *hbase.assignment.usezk.migrating*. Nonetheless, reverting from this state back to the Zookeeper-based region assignment mode will pose challenges. Any subsequent attempts to restart active master after this rollback is likely going to lead to multiple doubly assigned regions i.e. region inconsistencies and even data loss.

This has been fixed by https://github.com/apache/hbase/pull/3826. Anyone using HBase 1.6 are strongly recommended to apply this patch to their fork before HBase 2.x migration, or first migrate to HBase 1.7.2 before performing HBase 2.x upgrade.

3) HBASE-27502 Regionservers aborted as mvcc read point is less than max seq id derived from .seqid files

After upgrading one of our initial production clusters from HBase 1.6 to 2.4, we observed 144 out of 150 regionservers getting aborted with java.io.IOException: The new max sequence id {} is less than the old max sequence id {}.

The root cause of the issue: MapReduce job created snapshot using HBase 1.x binary, which internally creates snapshot scanner and eventually it creates new sequence id for the region as part of region open. Having new sequence id created as part of the this workflow is not compatible behavior with HBase 2.x server.
The recommendation is to let MapReduce jobs create or restore snapshot using HBase 2.x binary while the rolling upgrade is in progress.

4) HBASE-26708 Netty "leak detected" and OutOfDirectMemoryError due to direct memory buffering with SASL implementation

Conducting a series of extensive performance tests helped identify a notable concern: Over the course of a few hours, a few regionservers were observed to enter a stalled state, rendering them incapable of accommodating further requests from clients and intermittently encountering out-of-memory errors. Comprehensive explanations, accompanied by error stack traces are available on HBASE-26708. The issue turned out to be on the Sasl Netty decoder path. The fix: https://github.com/apache/hbase/pull/4596 is present on latest HBase 2.4 and 2.5 release lines.

This also provided motivation for additional fixes: https://github.com/apache/hbase/pull/4597

5) HBASE-25902 Add missing CFs (Column Families) in hbase:meta table during HBase 1 to 2.3+ Upgrade

The inclusion of the 'table' Column Family (CF) to the hbase:meta (or meta) table was done by HBASE-12035 for HBase 2.0.0. Subsequently, HBASE-23782 (2.3.0) introduced the reversal of hardcoding of the meta table schema, enabling the schema to be derived from the filesystem. It is important to note that the creation of the meta table schema occurs exclusively during the initial installation phase only. During the upgrade, HBase 1.x meta table schema would not be suitable for HBase 2.3+ versions due to the absence of required Column Families. Nonetheless, had we maintained a closer alignment with the community's official releases, this problem would not have occurred. In such a scenario, the process of upgrading from HBase 1.x to versions 2.0, 2.1, or 2.2 would have been more straightforward. This, in turn, would have facilitated a smoother transition to versions 2.3 and beyond without encountering significant complications.

To avoid the necessity of undergoing an intermediate upgrade via version 2.2 (<2.3), significant enhancements have been made to the active master initialization workflow. The resolution, available at https://github.com/apache/hbase/pull/3417, provides modifications to the active master's initialization workflow, ensuring the incorporation of required Column Families (CFs) into the meta table. Moreover, this alteration prompts the active master to initiate an orderly termination, thereby paving the way for a subsequent active master to seamlessly execute the complete initialization process. This refinement guarantees the successful execution of an uninterrupted in-place rolling upgrade from HBase 1.6/1.7 to 2.4, without any adverse impact on the continuity of operations related to the meta table.

6) HBASE-26797 HBase 1.x clients will choke on rep_barrier rows when scanning hbase 2.x meta

Upon completing the upgrade process for the initial clusters to HBase 2.4, an unexpected issue surfaced: the normalizer activity resulted in the merging of several regions. Consequently, a subset of client requests encountered failures for the affected table keyspace, encompassing the merged regions. Investigation revealed a distinction in region location extraction from the meta table between HBase 2.x and HBase 1.x clients. Specifically, HBase 2.x clients exclusively search for the 'info' Column Family (CF) within the meta table, whereas HBase 1.x clients explore any available CF for the corresponding region.

Even after the successful region split/merge done by HBase 2.x server, HBase 1.x client can still misinterpret 'rep_barrier' CF as 'info' CF. This misinterpretation results in operational failures, as the region in question is no longer in *ONLINE* state after the completion of merge or split operations.

Since this is fixed by https://github.com/apache/hbase/pull/4161, it is advisable for individuals utilizing HBase 1.6 to consider either integrating this patch into their repository prior to performing the HBase 2.x migration or alternatively, proceeding with the migration to HBase 1.7.2 before initiating the HBase 2.x upgrade process.

7) PHOENIX-6500 Allow 4.16 client to connect to 5.1 server

For Phoenix users, the exclusive and recommended version for seamless upgrade from HBase 1.x to 2.x is 4.16. Our compatibility can facilitate Phoenix 4.16 clients executing queries against Phoenix 5.1 server operating on HBase 2.4/2.5 environment. Phoenix versions 4.16 and 5.1 offer functional parity, despite the fact that 4.16 is compatible with HBase 1.x, while 5.1 is compatible to run with HBase 2.x versions and not the other way around.

8) PHOENIX-6758 During HBase 2 upgrade Phoenix Self healing task fails to create server side connection before reading SYSTEM.TASK

During in-place rolling upgrade, where the HBase master operates on version 1.x while the regionservers, specifically system rsgroup regionservers, are functioning on version 2.x, the TaskRegionObserver encounters issues in establishing connections for the purpose of accessing and reading SYSTEM.TASK records.
e.g. `org.apache.hadoop.hbase.regionserver.NoSuchColumnFamilyException: Column family table does not exist in region hbase:meta,,1.1588230740 in table`

This exception occurs because the meta table is missing required schema changes, for certain administrative APIs to be operative, in the middle of the rolling upgrade. The meta table schema gets updated only after HBase masters are upgraded to HBase 2.

The fix, https://github.com/apache/phoenix/pull/1477, involves the creation of backward-compatible administrative APIs, thereby ensuring uninterrupted functionality during the rolling upgrade process. This solution has been integrated into Phoenix 5.1.3, thus making it the recommended version for the upgrade procedure.

9) HBASE-28042 and HBASE-28081 Snapshot corruption due to non-atomic rename, atomic rename could lead to ACL retention issues

HBase provides capability for clients to create snapshot on a given table. The client can perform scan operation on the snapshot to directly read from HFiles and avoid having to rely on HBase server resources that are otherwise used to serve RPC requests on the live RegionServer serving the corresponding regions of the table. For HDFS file system, due to a bug, the snapshot commit phase no longer uses atomic rename while performing copy of snapshot metadata files from the snapshot working directory to the final commit directory. Due to the non-atomic rename, snapshot commit phase and snapshot create with the same snapshot name could create conflict and cause corruption in the snapshot metadata as discussed in HBASE-28042.

With this fix, the snapshot working directory can no longer retain HDFS ACLs from the snapshot parent directory. This requires additional fix from HBASE-28081.

10) HBASE-28050 RSProcedureDispatcher to fail-fast for krb auth failures

AssignmentManager V2 (AMv2) has been redesigned to implement all region open, close, move, re-open, split, merge processes based on ProcedureV2 (Pv2) framework. As part of this implementation, when the active master needs to dispatch the region open or close requests to a RegionServer, it can lead to infinite retries if the RPC requests could not be processed by the server due to kerberos authentication failures. In this case, it is safe to fail-fast the whole procedure because the server has not yet received the request from the master yet.

11) HBASE-26814 Default StoreHotnessProtector to off, with logs to guide when to turn it on

Store hotness protector, introduced by HBASE-19389 started causing issues while replicating WAL entries of specific table e.g. `org.apache.hadoop.hbase.RegionTooBusyException: StoreTooBusy,{table},{region}:{CF} Above parallelPreparePutToStoreThreadLimit(20)`

Nonetheless, the old default threshold for determining hotness is not what a usual user would consider hot and so this should always be disabled and only enabled after careful consideration of a correct threshold for the user's actual environment and workload. For anyone migrating to versions less than 2.5.0, the recommendation is to provide value `0` for config key `hbase.region.store.parallel.put.limit` (HBASE-26814).

12) Delays observed during the graceful restarts of regionservers:

Graceful restarts of regionservers heavily rely on the region mover functioning:

- use region mover to unload all the regions
- stop regionserver
- start regionserver
- use region mover to load back all previously help regions

The Region Mover utility in HBase is used a) to transfer or relocate all regions from a specified server (or RegionServer) to other operational (*or online*) servers, and b) to bring all those regions back from these servers to the original server where the region mover command was initiated. The default options provided by the region mover are available here.

Following the upgrade to HBase 2.4, we encountered notable delays — exceeding twice the duration of what it used to take with HBase 1.6 version — while executing the orderly restart of regionservers gracefully. By default, this script utilizes an acknowledgment mode, wherein, prior to the region transfer, a scan is performed on the region to verify a single row with cache blocks disabled. After the region is moved to the target server, the script once again initiates a scan, with caching disabled. Only upon the successful completion of both scans is the region considered to have been relocated successfully. While this approach is robust in terms of affirming successful transitions across the server fleet, it imposes significant overheads when dealing with a high volume of regions and will worsen the latency associated with the region movements. This mode does not scale for our clusters. Therefore, a transition to the non-acknowledgment mode becomes crucial for us, wherein regions are transferred through administrative APIs without any significant IO overheads. Non-ack can be achieved with option `-n` (reference: here).

Some useful references:

1. https://engineering.salesforce.com/evolution-of-region-assignment-in-the-apache-hbase-architecture-part-1-c43b1becc522/
2. https://engineering.salesforce.com/evolution-of-region-assignment-in-the-apache-hbase-architecture-part-2-9568fb3790b/
3. https://engineering.salesforce.com/evolution-of-region-assignment-in-the-apache-hbase-architecture-part-3-e03b814ae92/