Timeline Service v2 Backend Storage Performance Test Summary

Introduction

The next generation YARN Timeline Service is a work in progress, as noted in YARN-2928. HBase was chosen to be the backend store for capturing this information. Phoenix was one of the natural choices that could be used for querying the Timeline Service data stored in HBase. But we wanted to ascertain whether Phoenix could be used efficiently in the direct write path or if an hybrid approach should be taken where in plain HBase tables would be used in the write path in an OLTP-like schema and a Phoenix based schema would be used in the SQL like querying path as part of an OLAP schema. So we decided to evaluate both approaches in terms of their performance, scalability, usability, maintenance: YARN-3134 (Phoenix based HBase schema) and YARN-3411 (hybrid HBase schema - vanilla HBase tables in the direct write path and phoenix based tables for reporting).

This write-up describes how we ended up choosing the approach of writing to vanilla HBase tables in the direct write path.

Description

YARN-3134 considers a Phoenix based approach for storing Timeline Entity information emitted by applications. YARN-3411 considers storing the Timeline Entity information in a vanilla HBase table. Refer to jira attachments (Phoenix based schema and vanilla HBase schema) for detailed descriptions of tables in both schemas.

Testing details

Test driver description

MAPREDUCE-6335 and MAPREDUCE-6337 describe the work done for implementing a driver to test drive the timeline entity information storage. We ran the test driver in two modes:

1. Generate synthetic data: In this mode, the driver code generated timeline entity data as per the input parameters and emitted them to the backend via Timeline Writer APIs. The driver generated one entity with a randomly generated string payload of specified size attached to the info field of the entity (configurable via command line parameters), one event, one single-valued metric, and one config key-value.
2. Load data from existing job history & job conf files on hdfs: In this mode, the driver code read job history and job conf files from hdfs, created timeline entity objects with the data read from files and emitted them to the backend via Timeline Writer APIs. These history and conf files belonged to jobs that had been run on Twitter's production and ad-hoc clusters. This dataset had several variations of job history data like jobs whose setup itself had failed (thus no tasks) to jobs with just 1 container to jobs with ~20k containers. The history file sizes ranged from 10KB to 200MB. The job conf file sizes ranged from 142KB to 1.8MB.

Test environment description

A Hadoop/HBase cluster was setup at a Twitter datacenter to test these frameworks. For simplicity we used a single cluster to both drive workload as well as run HBase to record data.

The cluster had 33 worker nodes spread over 3 racks. Each node was a datanode, a compute node and a regionserver node. Both the DN and NM had a 1GB JVM heap. The region servers were given a 10G heap. Each node had assigned 10G for containers (nodemanager.container.memory). We set the mapper task size to 6G and set the number of mapper tasks to the number of worker nodes so that the mapper tasks spread out to all nodes evenly.

Each node consisted of a single socket, 8-core CPU, 32G of RAM and contained 12 1T drives. The nodes had a 10Gbps NIC. Due to the rack diversity, a 4x TOR over-subscription

was not a bottleneck during these tests. To read more about the hadoop machines used at Twitter, check out this Hadoop Summit Talk by @joep and @eecraft.

Versions

- The Hadoop cluster was based on the 3.0 snapshot build based on this commit on YARN-2928.
- The HBase cluster that was setup on this 3.0 hadoop cluster had the HBase 1.0.1 release.
- Phoenix driven code used jars phoenix-server-4.5.0-20150506.001937-1.jar and phoenix-core-4.5.0-20150506.001901-1.jar (SNAPSHOTs).

Testing process

Once we had confirmed the hadoop cluster was working correctly by running example applications, we created the schemas in HBase and looked at them via the HBase shell. We then launched the loader jobs from the command line one by one and captured runtime stats reported by the loader, validated this against metrics shown by the MR jobs through the RM/AM pages. After each run of a test, we confirmed the number of records loaded in HBase and reads and writes reported by HBase through the HBase UI and noted the counters emitted by the driver.

Results

Test description	**# Map tasks**	**# Entities per mapper**	**# total entities written**	**Phoenix Transaction Rate (per mapper) ops/sec**	**HBase Transaction Rate (per mapper) ops/sec**	**Phoenix Write Time (job counter TIMELINE_SERVICE_WRITE_TIME)**	**Hbase Write Time (job counter TIMELINE_SERVICE_WRITE_TIME)**
synthetic data	170	1k	170k	112.83	2285.13	1,506,704	74,394
synthetic data	170	10k	1.7M	53.029	636.41	32,057,957	2,671,241
synthetic data	1	50k	50k	196.67	19770.66	254,225	2,529
9 history files	33	-	85k	319.19 *(some write errors)**	962.32	265,460	88,049
555 history files	33	-	810k	206.25 *(some write errors)**	927.62	4,102,364	874,151

*Note: it's highly possible that the write errors of the Phoenix writer came from some special characters in the configs, which may cause errors with SQL statements. We calculate the write throughput by (*N_entity_planned_to_write*) / *time*, but the total number of entities

actually written is less than *N_entity_planned_to_write* due to the errors. Based on our observation, the actual write throughput should be 50%~60% of the reported value.

Observations and Discussions

- Write throughput of HBase is one order of magnitude greater than of Phoenix, even though both writers worked on exactly the same HBase cluster.
- The region pre splits used by the HBase approach helped distribute the load across region servers. It took a few attempts to get the distribution going. The table was created with splits based on alphabets like ‘a’, ‘ad’, ‘an’, ‘b’, …..‘r’,’s’,’se’...’z’. A KeyPrefixRegionSplitPolicy was set as the split policy for the table at schema creation time.
- With the Phoenix writer, no pre-splits were created for tables. We observed that the data distribution did not spread beyond 3-4 nodes. During the test we would like to apply the same pre-split strategy as the HBase approach. However, the only way we discovered to achieve this is to use SPLIT ON substatement, which requires us to rewrite the HBase pre-splitting strategy.
- The used heap size on the region servers did not exceed 50% of the max heap (11520 MB).
- During the Phoenix runs we noticed that both the client and the HBase region servers were running under much larger IO wait and CPU loads than with the standalone HBase client.
- For the load generator side, we noticed mappers (writers) spent around 2x time on CPU for Phoenix compare to HBase. Meanwhile, from the throughput data, Phoenix writers spent around 10x time on writing the same number of timeline entities. From our timing, creating a “cached” Phoenix connection (“cached” means an underlying HBase connection exists in the system) took about 0.2 millisecond. So our hunch for Phoenix here is that its run time was actually dominated by the I/O time rather than CPU time.

Summary

The results we got from these tests were consistent with earlier runs on an standalone environment. We found the standalone HBase client to outperform the Phoenix implementation by a factor of about 8x.

On average, we saw that the HBase client was able to write about 1 entity per ms, and that the average job (consisting of ~1,900 entities) were loaded in less than a couple of seconds. Some of the tests with the Phoenix client were not able to load all data, due to a small bug with parsing data that resulted in incorrect SQL getting generated.

Outcome

Based on the performance evaluation, the current approach chosen is that the Timeline Service information will be written to vanilla HBase tables and Phoenix based tables are to be considered in schemas that are to be used for reporting and analytics for Timeline Service data. The idea is to populate these Phoenix tables with aggregated data from the Timeline Service entity information either via coprocessors or via some aggregation processes which will not be in the direct write path for Timeline Service entities.

Appendix

Experiences on Phoenix

Originally we thought Phoenix is simply an additional SQL layer that can automatically “attach” to an existing HBase writer design. It turned out we were wrong. In order to support SQL, it replaces, rather than extends, the normal workflow of the HBase writer. For example, almost every operation to Phoenix needs to be done through SQL but not HBase API. This fact brings in limitations for both semantics and implementations. Here we list our problems.

- Performance-wise, most, if not all, operations to Phoenix need to go through the JDBC interface. For simple writes this may be an overkill (the HBase writer logic is also quite straightforward). Performance tuning to Phoenix appears to be less direct compare to

HBase, such as pre-splitting. As a suggested future work item, it will be very helpful to have documents about Phoenix performance tuning, for users familiar with HBase.

- We are using dynamic columns in Phoenix, but it appears that there's no simple way to iterate available dynamic columns within a column family. Iterating dynamic columns is one key use case for timeline service v2, so missing this feature causes some design challenges. We understand that SQL itself may not work well with HBase's dynamic columns (that's why HBase is NoSQL), but it will certainly be helpful to have some extension on this. After all, what we need for Phoenix is a SQL-*like* query engine for HBase. For us it does not need to stick with *standard* SQL.
- Storing time series data in Phoenix is also a challenge. Unlike HBase, we cannot directly access the timestamp of each cell in Phoenix. With the Phoenix writer, right now timeline service v2 only supports single data metrics but we do plan to add support to time series data soon. If we're still using Phoenix storage in some sort of "aggregation tables" for timeline v2, we need to carefully design the schema for time series metric data.
- When deploying Phoenix, we need to find out one version that works well with the latest or a reasonably late HBase release. This may not be a trivial work in real life deployments, especially when Phoenix is deployed side-by-side on an HBase cluster where the HBase version is new.

Of course, the ability to support SQL queries in Phoenix is attractive. However a general impression for the pure Phoenix writer is that we're some sort of "forced" too far away, by current Phoenix, to pursue the conventional SQL way on NoSQL DB. It will be very helpful to have SQL-like query language *together with* original HBase APIs.