Apache HBase - Apache Spark Integration Scope

busbey@apache.org - 20170718

Problem

Users of Apache HBase must rely on an external distributed processing framework when their use case requires either more throughput or resiliency than a single client process can provide. Currently, the de facto standard integration that comes with an HBase installation is with Apache Hadoop MapReduce. This integration includes both utilities for some administrative tasks as well as APIs that make writing custom MapReduce based applications easier.

Apache Spark is a natural successor to Apache Hadoop MapReduce. A succinct description can be found in the Apache Incubator proposal for the Livy project[1]:

> Apache Spark is a fast and general purpose distributed compute engine, with a versatile API. It enables processing of large quantities of static data distributed over a cluster of machines, as well as processing of continuous streams of data. It is the preferred distributed data processing engine for data engineering, stream processing and data science workloads.

We propose including a similar level of integration for Apache Spark as is currently done for Apache Hadoop MapReduce. Providing this integration as a first class component of the Apache HBase project allows us to centralize best practices and will ensure that those community members who perform their work using Apache Spark have a high quality experience.

Exit Criteria

The following criteria will be used to determine if this feature is ready to be included in our releases of Apache HBase:

- Our integration works at runtime with Spark releases in the 2.y major release line[2]. At this time, that includes Apache Spark 2.0 through 2.2.

[1] https://wiki.apache.org/incubator/LivyProposal
[2] http://spark.apache.org/versioning-policy.html

- Our integration works at runtime with Spark deployments that rely on whatever Scala version(s) a given Spark version is made to work with. At this time, that includes Scala 2.10 and Scala 2.11.
- Our integration works at runtime with secure HBase deployments whenever a Spark application that uses the traditional HBase client API would work.
- Our integration is present in one or more Maven modules, producing any needed jars for the above supported combinations of Spark and Scala versions in a single build. Such modules should ensure that no Scala dependencies are present outside of those modules specific to the integration work.
- Our convenience binary packaging ensures that by default no Scala code shows up in the classpath of HBase server processes nor non-Spark clients of HBase.
- Tests of added functionality, as described below, all run on ASF Jenkins[3] in our project view and pass.
- Our documentation covers all added functionality, as described below.

Future work

The following improvements are desirable, but in the interests of getting a viable tool into releases soon will not be considered necessary for this feature to be ready for downstream production use:

- Composite row keys
- Apache Spark PySpark integration
- Apache HBase administrative tool parity with those that currently run on MapReduce
- Apache Spark Structured Steaming integration
- Compatibility with the HBase DataType API[4]
- Compatibility with the encoding used by Apache Phoenix
- Handling of delegation tokens in the presence of multiple secure HBase deployments

Technical Approach

Spark Processing Abstractions

Spark has a large variety of APIs for interacting with data. Briefly, they can be described as:

- RDDs, essentially in-memory tabular data[5]

[3] https://builds.apache.org/view/H-L/view/HBase/

[4] http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/types/package-summary.html#package.description

[5] https://spark.apache.org/docs/latest/programming-guide.html

- Streaming, a series of the above RDDs over time[6]
- DataSource, an sql-oriented structured data processing that exposes computation info to the storage layer[7], usually used through Apache Spark SparkSQL
- Structured Streaming, essentially a series of DataSource results over time[8]

Our initial support will include integration that allows those using Spark with the RDD, Streaming, and DataSource abstractions via either the Scala or Java APIs to interact with HBase, either through a shared *Connection* object or using an tailored API for common operations like *Put*, *Get*, and *Delete*.

Bulkload

A very common use case for distributed computation frameworks like MapReduce and Spark is doing high-throughput insertion into HBase with latency requirements that are lax compared to the normal random read/write API. This is normally accomplished by writing to HFiles (the backing cold storage format for HBase) and then instructing HBase to perform a *bulk load* operation on these files. We will provide an API option geared to this use case.

Pluggable Layout and Data Encoding

Support for the Spark Dataset/Dataframe abstraction requires providing a Spark *catalog* mapping Spark's table / field abstraction to rows/columns in an HBase table. Additionally, it requires support for dealing with stored bytes in terms of the pre-defined types within SparkSQL.

Historically, HBase has maintained an agnostic position with respect to how end users encode their various data into the byte arrays used throughout the HBase API. To properly support SparkSQL, we will need a pluggable system that can deal with common user encodings. Initially, we should include implementations for:

- Java Native[9]
- Apache Avro

Build and Packaging

Adding Spark integration carries a bit of risk; the Spark project is relatively young and moves quickly with respect to APIs. Compounding this issue, the Spark project relies on Scala as a common implementation language and Scala has a track record of mutually incompatible releases. We will handle this combination of possible deployment environments by relying on

---

[6] https://spark.apache.org/docs/latest/streaming-programming-guide.html
[7] https://spark.apache.org/docs/latest/sql-programming-guide.html
[8] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
[9] http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/util/Bytes.html

the build system to create implementation artifacts specific to each of them. Where practical, we will avoid duplication of implementation source code. Release managers will not have to do any special additional steps when building our convenience binary tarballs.

Module layout

To minimize the impact on the rest of the HBase project, the integration work for this feature will be isolated in a set of new modules. These modules will depend on existing HBase modules, possibly making use of InterfaceAudience.Private internals. No existing modules will depend on these new modules, with exceptions of additions to existing Maven Reactor builds (so the modules will build) and our Maven Assembly definitions (so the modules will be included in our release and convenience binaries). No existing modules will depend on either the Spark project artifacts or Scala libraries.

Upgrade Impact

As a new feature, we do not expect this integration to have any impact on existing installations of Apache HBase releases.

Compatibility with existing third party solutions for accessing HBase from Spark applications is expressly not a consideration. If we can get contributions on how to migrate from those familiar with said third party works, we should include them in our documentation.

Test Strategy

To ensure that production users can rely on our Spark integration to provide performant, reliable, competent access to their HBase data, we will include the following test updates when labeling the feature as production ready: unit tests, an integration test, and updates for our automated contribution testing.

Unit Tests

We must have unit tests that cover the major functional parts of our spark integration. This should include both our mapping of Spark abstractions to HBase notions (e.g. RDD partitions to HBase regions) as well as our handling of data encodings. Testing should ensure we produce the expected exceptions in the face of error conditions.

Integration Tests

We must include an example use of our integration that can provide end-to-end verification that things work as expected when running on top of Spark (as opposed to using mocks or other

stand-ins for the Spark framework as we might do in unit tests). This example should work during our normal build process, as a part of the integration-test phase of the Maven lifecycle. Additionally, it should also work when deployed on a cluster to run against an existing HBase instance.

Versioning policies do not ensure perfect execution. In the event that our list of supported Spark versions include more than one minor release and/or our Scala versions include more than one incompatible version, we should run our integration test(s) on each of them to verify our assumptions about runtime compatibility. In the event that supporting a different deployment environment requires different source code, we should ensure any integration test that exercises that difference is run as a part of the build process. In all cases, we should have periodic runs of our integration test(s) against expected deployment environments that combine possible minor versions of Spark and Scala on the project's test hardware (e.g. builds.a.o). Release managers should verify the state of these test runs when evaluating branch readiness for a release.

Automated Contribution Testing

Given the goal of supporting use with the Spark 2.y major release line, we will update our automated testing via Apache Yetus Test Patch to test new contributions by compiling against a variety of Spark releases, similar to how we currently ensure that our source can be built with all of the Hadoop versions we list as supported.

Documentation

As a user-facing feature, it is vital that we have sufficient documentation describing both what users can expect to do with our API, best practices for running applications built on top of that API, as well as examples that show them end-to-end usage.

To ensure users can self-service use of our integration with supported Spark APIs, we'll include:

- An Apache Maven Archetype[10] for a Spark application, to supplement those we already have in progress[11].
- An additional section in the HBase Reference Guide's section on prerequisites[12] that details the versions of Apache Spark we expect our integration to work with.
- A dedicated section in the HBase Reference Guide that covers how we integrate with Apache Spark and examples of how users should build their applications on top of that integration. Coverage should include all data encoding options we include in our release.
- An equivalent to javadocs as present for the rest of the HBase API, either included in the main javadocs or if needed as a separate section. These should either break things up

[10] https://maven.apache.org/guides/introduction/introduction-to-archetypes.html
[11] http://hbase.apache.org/book.html#hbase.archetypes.development
[12] http://hbase.apache.org/book.html#basic.prerequisites

into the public / internals as we do with the rest of our API docs, or should expressly warn users that only those things marked as public will be stable across versions.
- A blog post on the project blog that introduces use of the feature for an intended audience of downstream users.

Milestones

Following are expected steps to get from the current work in the *master* branch to something that matches the above exit criteria for the feature.

Given schedules are estimates of task start to finish, which might include handling by different contributors. For example, the time on Milestone 1 includes time for some folks to provide reviews while others may be updating code based on consensus around implementation.

Milestone 0: Readiness review

Summary: check current implementation against exit criteria
Schedule: 3 days
Exit Criteria:

- JIRAs identified or filed for gaps against exit criteria
- Milestone(s) added for above JIRAs
- Details identified for Milestone 2: Unit Tests
- Details identified for Milestone 4: Reference Guide

Milestone 1: Spark 2.y update

Summary: complete update for supporting Spark 2 releases at runtime
Schedule: 1 week
Exit Criteria:

- Project updated to build against Spark 2.0
- Precommit updated to build against versions in ref guide

Milestone 2: Unit Tests

Summary: fill in gaps on unit coverage for supported features
Schedule: unknown prior to Milestone 0
Exit Criteria:

- Unit tests for each public API

Milestone 3: Maven Archetype

Summary: add hbase-spark-project to the set of project archetypes

Schedule: 1 week
Exit Criteria:

- Archetype added
- Archetype test added

Milestone 4: Reference Guide

Summary: fill in gaps for user facing documentation
Schedule: unknown until after Milestone 0.
Exit Criteria:

- Feature coverage in Spark section of reference guide
- Spark versions covered in prerequisites
- Ensure precommit covers versions given in prerequisites

Milestone 5: User facing API docs

Summary: ensure website has docs for downstream users
Schedule: 3 days
Exit Criteria:

- API docs created in build process (e.g. javadoc and/or scaladoc)
- If needed, website update for hosting said API docs

Milestone 6: HBase 1.y backport

Summary: Spark integration related modules backported to the next HBase 1.y release branch post Milestones 0-4.
Schedule: 3 days
Exit Criteria: It may not be feasible to keep commit history for this milestone, so we will accept a single monolithic backport instead.

Milestone 7: Evangelism

Summary: make downstream aware of our integration
Schedule: 1 week
Exit Criteria: Blog post that walks through example based on archetype generated project.

Revision History

- 2017-07-18 - busbey@apache.org - initial version

- 2017-07-19 - busbey@apache.org - update for review feedback from mdrob@apache.org on HBASE-18405. Fleshed out handling of ITs and deploy environments. Added precommit update to milestones. Added milestone for API docs.