Cloudera Runtime 7.2.6

Developing Applications with Apache Kudu

Date published: 2020-02-28
Date modified: 2020-12-01

CLOUDERA

Legal Notice

Contents

View the API documentation

This topic provides you information on how to find the API documentation for C++ and Java.

⚠️ **Warning:** Use of server-side or private interfaces is not supported, and interfaces which are not part of public APIs have no stability guarantees.

C++ API Documentation

The documentation for the C++ client APIs is included in the header files in /usr/include/kudu/ if you installed Kudu using packages or subdirectories of src/kudu/client/ if you built Kudu from source. If you installed Kudu using parcels, no headers are included in your installation. and you will need to build Kudu from source in order to have access to the headers and shared libraries.

The following command is a naive approach to finding relevant header files. Use of any APIs other than the client APIs is unsupported.

```
find /usr/include/kudu -type f -name *.h
```

Java API Documentation

View the *Java API documentation* online. Alternatively, after building the Java client, Java API documentation is available in java/kudu-client/target/apidocs/index.html.

Related Information
Java API documentation

Kudu example applications

Several example applications are provided in the examples directory of the Apache Kudu git repository. Each example includes a README that shows how to compile and run it. The following list includes some of the examples that are available today. Check the repository itself in case this list goes out of date.

cpp/example.cc:

A simple C++ application which connects to a Kudu instance, creates a table, writes data to it, then drops the table.

java/java-example:

A simple Java application which connects to a Kudu instance, creates a table, writes data to it, then drops the table.

java/insert-loadgen:

A small Java application which listens on a TCP socket for time series data corresponding to the Collectl wire protocol. The commonly-available collectl tool can be used to send example data to the server.

python/dstat-kudu:

A Java application that generates random insert load.

python/graphite-kudu:

An example program that shows how to use the Kudu Python API to load data into a new / existing Kudu table generated by an external program, dstat in this case.

python/graphite-kudu:

An example plugin for using graphite-web with Kudu as a backend.

These examples should serve as helpful starting points for your own Kudu applications and integrations.

Related Information
Kudu examples

Kudu Python client

The Kudu Python client provides a Python friendly interface to the C++ client API. To install and use the Kudu Python client, you need to install the Kudu C++ client libraries and headers.

Before you begin
See Install Using Packages topic for installing the Kudu C++ client libraries.

Procedure

1. Update all the packages on your system by using the following command:

   yum -y update

2. Install the extra packages for the Enterprise Linux distribution:

   sudo yum -y install epel-release

3. Install the Python package manager:

   sudo yum -y install python-pip

4. Verify the version of the PIP manager that you just installed:

   pip --version

5. Install Cython:

   sudo pip install cython

6. Download the following files using wget:

   • Kudu artifact: http://username:password@archive.cloudera.com/p/cdp-public/*RUNTIME_VERSION*/redhat7/
     yum/kudu/*KUDU_ARTIFACT*
   • Kudu-client artifact: http://username:password@archive.cloudera.com/p/cdp-public/*RUNTIME_VERSION*/red
     hat7/yum/kudu/*KUDU-CLIENT_ARTIFACT*
   • Kudu-client-devel artifact: http://username:password@archive.cloudera.com/p/cdp-publ
     ic/*RUNTIME_VERSION*/redhat7/yum/kudu/*KUDU-CLIENT-DEVEL_ARTIFACT*

7. Install the kudu package from the local directory:

   sudo yum -y localinstall ./kudu-*

8. Install the package used for developing Python extensions:

   sudo yum -y install python-devel

9. Upgrade the setup tools:

   sudo pip install --upgrade pip setuptools

10. Install the C++ compiler:

    sudo yum -y install gcc-c++

11. Install the Kudu-python client:

    sudo pip install kudu-python==<kudu-version>

12. Install kudu-python: sudo pip install kudu-python.

The following sample demonstrates the use of part of the Python client:

```
import kudu
from kudu.client import Partitioning
from datetime import datetime

# Connect to Kudu master server
client = kudu.connect(host='kudu.master', port=7051)

# Define a schema for a new table
builder = kudu.schema_builder()
builder.add_column('key').type(kudu.int64).nullable(False).primary_key()
builder.add_column('ts_val', type_=kudu.unixtime_micros, nullable=False,
 compression='lz4')
schema = builder.build()

# Define partitioning schema
partitioning = Partitioning().add_hash_partitions(column_names=['key'],
num_buckets=3)

# Create new table
client.create_table('python-example', schema, partitioning)

# Open a table
table = client.table('python-example')

# Create a new session so that we can apply write operations
session = client.new_session()

# Insert a row
op = table.new_insert({'key': 1, 'ts_val': datetime.utcnow()})
session.apply(op)
# Upsert a row
op = table.new_upsert({'key': 2, 'ts_val': "2016-01-01T00:00:00.000000"})
session.apply(op)
# Updating a row
op = table.new_update({'key': 1, 'ts_val': ("2017-01-01", "%Y-%m-%d")})
session.apply(op)
# Delete a row
op = table.new_delete({'key': 2})
session.apply(op)
# Flush write operations, if failures occur, capture print them.
try:
    session.flush()
except kudu.KuduBadStatus as e:
    print(session.get_pending_errors())

# Create a scanner and add a predicate
scanner = table.scanner()
scanner.add_predicate(table['ts_val'] == datetime(2017, 1, 1))
# Open Scanner and read all tuples
# Note: This doesn't scale for large scans
result = scanner.open().read_all_tuples()
```

Kudu integration with Spark

Kudu integrates with Spark through the Data Source API as of version 1.0.0. Include the kudu-spark dependency using the --packages option.

Use the kudu-spark_2.10 artifact if using Spark with Scala 2.10. Note that Spark 1 is no longer supported in Kudu starting from version 1.6.0. So in order to use Spark 1 integrated with Kudu, version 1.5.0 is the latest to go to.

```
spark-shell --packages org.apache.kudu:kudu-spark_2.10:1.5.0-cdh5.13.91 --re
positories https://repository.cloudera.com/artifactory/cloudera-repos/
```

Use kudu-spark2_2.11 artifact if using Spark 2 with Scala 2.11.

> **Note:** kudu-spark versions 1.8.0 and below have slightly different syntax. See the documentation of your version for a valid example. Versioned documentation can be found on the releases page.

```
spark2-shell --packages org.apache.kudu:kudu-spark2_2.11:1.9.0-cdh6.2.0 --re
positories https://repository.cloudera.com/artifactory/cloudera-repos/
```

Below is a minimal Spark SQL "select" example for a Kudu table created with Impala in the "default" database. We first import the kudu spark package, then create a DataFrame, and then create a view from the DataFrame. After those steps, the table is accessible from Spark SQL. You can also refer to the Spark quickstart guide or this Kudu-Spark example.

> **Note:** You can use the Kudu CLI tool to create table and generate data by kudu perf loadgen kudu.master:70 51 -keep_auto_table for the following two examples:

```
import org.apache.kudu.spark.kudu._
// Create a DataFrame that points to the Kudu table we want to query.
val df = spark.read.options(Map("kudu.master" -> "kudu.master:7051",
                                "kudu.table" -> "default.my_table")).format(
"kudu").load
// Create a view from the DataFrame to make it accessible from Spark SQL.
df.createOrReplaceTempView("my_table")
// Now we can run Spark SQL queries against our view of the Kudu table.
spark.sql("select * from my_table").show()
```

Below is a more sophisticated example that includes both reads and writes:

```
import org.apache.kudu.client._
import org.apache.kudu.spark.kudu.KuduContext
import collection.JavaConverters._

// Read a table from Kudu
val df = spark.read
  .options(Map("kudu.master" -> "kudu.master:7051", "kudu.table" -> "kudu_ta
ble"))
  .format("kudu").load

// Query using the Spark API...
df.select("key").filter("key >= 5").show()

// ...or register a temporary table and use SQL
df.createOrReplaceTempView("kudu_table")
val filteredDF = spark.sql("select key from kudu_table where key >= 5").show
()

// Use KuduContext to create, delete, or write to Kudu tables
val kuduContext = new KuduContext("kudu.master:7051", spark.sparkContext)
// Create a new Kudu table from a DataFrame schema
// NB: No rows from the DataFrame are inserted into the table
kuduContext.createTable(
    "test_table", df.schema, Seq("key"),
    new CreateTableOptions()
        .setNumReplicas(1)
        .addHashPartitions(List("key").asJava, 3))
```

```
// Check for the existence of a Kudu table
kuduContext.tableExists("test_table")

// Insert data
kuduContext.insertRows(df, "test_table")

// Delete data
kuduContext.deleteRows(df, "test_table")

// Upsert data
kuduContext.upsertRows(df, "test_table")
// Update data
val updateDF = df.select($"key", ($"int_val" + 1).as("int_val"))
kuduContext.updateRows(updateDF, "test_table")
// Data can also be inserted into the Kudu table using the data source, tho
ugh the methods on
// KuduContext are preferred
// NB: The default is to upsert rows; to perform standard inserts instead, s
et operation = insert
// in the options map
// NB: Only mode Append is supported

df.write
  .options(Map("kudu.master"-> "kudu.master:7051", "kudu.table"-> "test_t
able"))
  .mode("append")
  .format("kudu").save

// Delete a Kudu table
kuduContext.deleteTable("test_table")
```

Spark integration known issues and limitations

Here are the limitations that you should consider while integrating Kudu and Spark.

- Spark 2.2 (and higher) requires Java 8 at runtime even though Kudu Spark 2.x integration is Java 7 compatible. Spark 2.2 is the default dependency version as of Kudu 1.5.0.
- Kudu tables with a name containing upper case or non-ASCII characters must be assigned an alternate name when registered as a temporary table.
- Kudu tables with a column name containing upper case or non-ASCII characters must not be used with SparkSQL. Columns can be renamed in Kudu to work around this issue.
- <> and ORpredicates are not pushed to Kudu, and instead will be evaluated by the Spark task. Only LIKE predicates with a suffix wildcard are pushed to Kudu. This means LIKE   "FOO%" will be pushed, but LIKE "FO O%BAR" won't.
- Kudu does not support all the types supported by Spark SQL. For example, Date and complex types are not supported.
- Kudu tables can only be registered as temporary tables in SparkSQL.
- Kudu tables cannot be queried using HiveContext.

Spark integration best practices

It is best to avoid multiple Kudu clients per cluster.

A common Kudu-Spark coding error is instantiating extra KuduClient objects. In kudu-spark, a KuduClient is owned by the KuduContext. Spark application code should not create another KuduClient connecting to the same cluster. Instead, application code should use the KuduContext to access a KuduClient using KuduContext#syncClient.

To diagnose multiple KuduClient instances in a Spark job, look for signs in the logs of the master being overloaded by many GetTableLocations or GetTabletLocations requests coming from different clients, usually around the same time. This symptom is especially likely in Spark Streaming code, where creating a KuduClient per task will result in periodic waves of master requests from new clients.

Upsert option in Kudu Spark

The upsert operation in kudu-spark supports an extra write option of ignoreNull. If set to true, it will avoid setting existing column values in Kudu table to Null if the corresponding DataFrame column values are Null. If unspecified, ignoreNull is false by default.

```
val dataFrame = spark.read
   .options(Map("kudu.master" -> "kudu.master:7051", "kudu.table" -> simpl
eTableName))
   .format("kudu").load
dataFrame.createOrReplaceTempView(simpleTableName)
dataFrame.show()

// Below is the original data in the table 'simpleTableName'
+---+---+
|key|val|
+---+---+
|  0|foo|
+---+---+

// Upsert a row with existing key 0 and val Null with ignoreNull set to true
val nullDF = spark.createDataFrame(Seq((0, null.asInstanceOf[String]))).t
oDF("key", "val")
val wo = new KuduWriteOptions
wo.ignoreNull = true
kuduContext.upsertRows(nullDF, simpleTableName, wo)
dataFrame.show()
// The val field stays unchanged
+---+---+
|key|val|
+---+---+
|  0|foo|
+---+---+

// Upsert a row with existing key 0 and val Null with ignoreNull default/set
 to false
kuduContext.upsertRows(nullDF, simpleTableName)
// Equivalent to:
// val wo = new KuduWriteOptions
// wo.ignoreNull = false
// kuduContext.upsertRows(nullDF, simpleTableName, wo)
df.show()
// The val field is set to Null this time
+---+----+
|key| val|
+---+----+
|  0|null|
+---+----+
```

Use Spark with a secure Kudu cluster

The Kudu-Spark integration is able to operate on secure Kudu clusters which have authentication and encryption enabled, but the submitter of the Spark job must provide the proper credentials. For Spark jobs using the default 'client' deploy mode, the submitting user must have an active Kerberos ticket granted through kinit. For Spark jobs

using the 'cluster' deploy mode, a Kerberos principal name and keytab location must be provided through the --princi
pal and --keytab arguments to spark2-submit.

Spark tuning

In general the Spark jobs were designed to run with minimal tuning and configuration. You can adjust the number of executors and resources to increase parallelism and performance using Spark's configuration options.

If your tables are super wide and your default memory allocation is fairly low, you may see jobs fail. To resolve this, increase the Spark executor memory. A conservative rule of thumb is 1 GiB per 50 columns.

If your Spark resources drastically outscale the Kudu cluster, then you may want to limit the number of concurrent tasks allowed to run on restore.