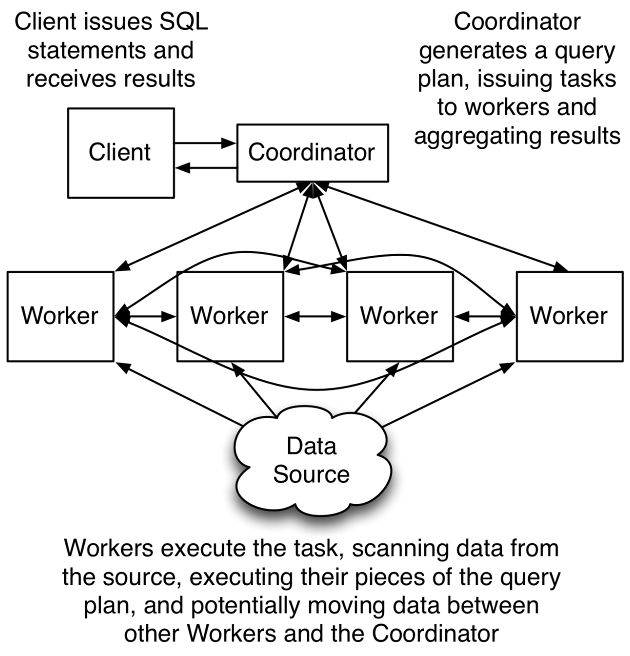
Apache Accumulo is based on Google's BigTable design, built on the Hadoop, Zookeeper, and Thrift projects (also from Apache), with strong support for data security built right in. Accumulo is a scalable, sorted, distributed key/value store. It stores relational rows of data as a collection of key/value pairs, which are sorted on their keys. Accumulo provides very fast retrieval of data when specifying either an individual key or a small range of keys.

We use Accumulo in a number of applications at Bloomberg Vault, as a database of communication events, as a triple store for entity relationships, and as a file store abstraction over HDFS. Applications that leverage data in Accumulo are typically written in the Java programming language. The Accumulo API is relatively simple to use, but is lacking a robust query framework. Applications are limited to writing data using Accumulo's *Mutation* object and reading data via iterating over *Scanner* or *BatchScanner* objects, extracting information from the raw key/value entries in the Accumulo table. This requires a lot of complex Java code and low-level data management, which is often similar in structure across many applications using Accumulo for data storage and retrieval We implemented an Accumulo connector for Presto to address these issues and to reduce the application development time.

We recently published our Presto-Accumulo connector, and this blog covers how it can be used to retrieve data from Accumulo using SQL. We'll also look at some performance metrics from the TPC-H benchmark suite, and wrap up by providing an overview of the functionality supported by the Presto-Accumulo connector.

Presto is a distributed ANSI SQL query engine for running interactive queries over very large data sets -- from gigabytes to petabytes. Originally built by Facebook, Presto supports a pluggable storage layer, allowing users to implement a connector to virtually any data storage system, big or small.

When building a connector, Presto allows users to split large data sets into pieces, called *splits*, which are then scanned in parallel to answer queries. The connector code uses the split to read a chunk of the data from the source, providing rows of data to a Presto worker. After the data is read, Presto takes over to execute the remaining query plan -- filters, joins, groups, and sorting -- once again in parallel.



Client issues SQL statements and receives results
Coordinator generates a query plan, issuing tasks to workers and aggregating results
Client
Coordinator
Worker
Worker
Worker
Worker
Data Source
Workers execute the task, scanning data from the source, executing their pieces of the query plan, and potentially moving data between other Workers and the Coordinator

Clerk Search - An Accumulo Application

In this example we build an application that enables looking up point-of-sale orders that were rung up by a given clerk. This application will use the TPC-H "orders" table, for which the data can be generated using tpch-dbgen. To create this application, we need to:

1. Create an ingestion application to
    a. Create the Accumulo data tables and index tables
    b. Parse a flat file containing generated TPC-H order data
    c. Encode each field of each row using Accumulo's lexicoder API, adding entries to a *Mutation*
    d. Create an index *Mutation* for mapping the clerk ID to the order ID
    e. Write the *Mutations* to their appropriate tables
2. Create a query application to
    a. Batch scan the index table for all order IDs rung up by a given clerk or clerks
    b. Batch scan the data table using the order IDs from the index table and a *WholeRowIterator*
    c. Iterate over the row, decoding the Accumulo *Value* objects into Java types via Accumulo's lexicoder API
    d. Print the resulting data

Some of the code is omitted for brevity -- and it is still so much code! The full source code for the application can be found in the presto-accumulo-examples sub-project in the presto-accumulo repository on GitHub.

Ingesting Data using the Accumulo APIs

Let's break down the ingestion application section by section. We first start by creating a `ZooKeeperInstance` and `Connector` to interact with Accumulo.

```
ZooKeeperInstance inst = new ZooKeeperInstance("default", "localhost:2181");
Connector conn = inst.getConnector("root", "secret");
```

We then create our data and index tables.

```
conn.tableOperations().create(DATA_TABLE);
conn.tableOperations().create(INDEX_TABLE);
```

We'll create a `MultiTableBatchWriter` and get `BatchWriter` objects for the data and index tables.

```
MultiTableBatchWriter mtbw = conn.createMultiTableBatchWriter(
            new BatchWriterConfig());
BatchWriter mainWrtr = mtbw.getBatchWriter(DATA_TABLE);
BatchWriter indexWrtr = mtbw.getBatchWriter(INDEX_TABLE);
```

Next, open the file to be read and begin reading lines of data. For each line of text, we split the line using a '|' field delimiter and convert the fields into their corresponding Java types.

```
BufferedReader rdr = new BufferedReader(new FileReader(ORDER_FILE);
String line;
while ((line = rdr.readLine()) != null) {
    // Split the line into fields
    String[] fields = line.split("\\|");

    Long orderkey = Long.parseLong(fields[0]);
    Long custkey = Long.parseLong(fields[1]);
    String orderstatus = fields[2];
    Double totalprice = Double.parseDouble(fields[3]);
    Date orderdate = sdformat.parse(fields[4]);
    String orderpriority = fields[5];
    String clerk = fields[6];
    Long shippriority = Long.parseLong(fields[7]);
    String comment = fields[8];
```

We then create a `Mutation` for each field in the row of data. The encode function used will take the given Java object, inspect the type, and then use the corresponding Accumulo lexicoder to convert it to a byte array. This is an efficient way to store data and is the preferred storage method for Accumulo.

```
    // Create mutation for the row
    Mutation mutation = new Mutation(encode(orderkey));
    mutation.put(CF, CUSTKEY, encode(custkey));
    mutation.put(CF, ORDERSTATUS, encode(orderstatus));
    mutation.put(CF, TOTALPRICE, encode(totalprice));
    mutation.put(CF, ORDERDATE, encode(orderdate));
    mutation.put(CF, ORDERPRIORITY, encode(orderpriority));
    mutation.put(CF, CLERK, encode(clerk));
    mutation.put(CF, SHIPPRIORITY, encode(shippriority));
    mutation.put(CF, COMMENT, encode(comment));
    mainWrtr.addMutation(mutation);
```

Finally, we index the data by creating a new `Mutation` with the row ID set to the value of the `clerk` field. We encode the `orderkey` as the key's column qualifier with an empty value. This is then written to the index table and we loop through the remaining rows of text.

```
    // Create index mutation for the clerk
    Mutation idxClerk = new Mutation(encode(clerk));
    idxClerk.put(CF, encode(orderkey), EMPTY_BYTES);
    indexWrtr.addMutation(idxClerk);
}
```

Once all the `Mutations` are created, we close the file reader and our `MultiTableBatchWriter`. We've now ingested and indexed the TPC-H orders data set.

```
mtbw.close();
rdr.close();
```

We can then build this program and use it to load data into the Accumulo data and index tables.

Searching for Data using the Accumulo APIs

Now that we have ingested our data set, we move our attention to the second application. We'll be creating a Java application that accepts clerk IDs via the command line. Using these clerk IDs, we search the index table for order IDs containing clerk data, then we use another `BatchScanner` to search the main orders table for the full row. This is a common Accumulo design pattern, often referred to as *secondary indexing.* By duplicating data in another table, we can drastically reduce the runtime of a query by first querying the index table to get all row IDs that contain a specific field value. Using these IDs, we then go to the main data table and retrieve rows by their unique identifier.

We start as we did before, by creating a `Connector` from a new `ZooKeeperInstance`.

```
ZooKeeperInstance inst = new ZooKeeperInstance("default", "localhost:2181");
Connector conn = inst.getConnector("root", "secret");
```

We then move on to creating and configuring a `BatchScanner` to search for data in our index table where the row ID is equal to the search term. We scan the index table and extract the column qualifier (which is a row ID in the orders table) and add it to a `List` of `Range` objects that we will use shortly.

```
BatchScanner idxScanner = conn.createBatchScanner(INDEX_TABLE,
            new Authorizations(), 10);
LinkedList<Range> searchRanges = new LinkedList<Range>();

String[] searchTerms = // ... retrieved from the command line
for (String searchTerm : searchTerms) {
    searchRanges.add(new Range(searchTerm));
}

// Set the search ranges for our scanner
idxScanner.setRanges(searchRanges);

// A list to hold all of the order IDs
List<Range> orderIds = new ArrayList<Range>();
String orderId;

// Process all of the records returned by the batch scanner
for (Map.Entry<Key, Value> record : idxScanner) {
    // Get the order ID and add it to the list of order IDs
    orderIds.add(new Range(record.getKey().getColumnQualifier()));
}

// Close the batch scanner
idxScanner.close();
```

Now that we have the row IDs, we create another `BatchScanner` against the main data table and set the ranges to our list of order IDs and create a bunch of local Java objects to grab the field data. We also configure a `WholeRowIterator` to simplify the scanning task. This iterator encodes all of the key/value pairs that share the same row ID into a single entry. We can then decode this on the client side to provide row isolation as well as simplify the iteration process.

```
BatchScanner dataScanner = conn.createBatchScanner(DATA_TABLE,
            new Authorizations(), 10);
dataScanner.setRanges(orderIds);
dataScanner.addScanIterator(new IteratorSetting(1, WholeRowIterator.class));
```

```
Long orderkey = null;
... // bunch of other local variables to store the fields
```

We then begin scanning data, decoding the row into a `SortedMap` for us to iterate over. We then get the column qualifier and, based on the value of the qualifier, we decode the value into the local variable.

```
Text row = new Text();
Text colQual = new Text();
for (Map.Entry<Key, Value> entry : dataScanner) {
    // Get the orderkey from the row
    entry.getKey().getRow(row);
    orderkey = decode(Long.class, row.getBytes(), row.getLength());

    // Decode the row into a map of entries and iterate over these
    SortedMap<Key, Value> rowMap =
            WholeRowIterator.decodeRow(entry.getKey(), entry.getValue());
    for (Map.Entry<Key, Value> record : rowMap.entrySet()) {
        // Get the column qualifier from the record's key
        record.getKey().getColumnQualifier(colQual);

        // Switch on the column qualifier and decode the value
        switch (colQual.toString()) {
            case CUSTKEY_STR:
                custkey = decode(Long.class, record.getValue().get());
                break;
            case ORDERSTATUS_STR:
                orderstatus = decode(String.class, record.getValue().get());
                break;
            case TOTALPRICE_STR:
                totalprice = decode(Double.class, record.getValue().get());
                break;
            case ORDERDATE_STR:
                orderdate = decode(Date.class, record.getValue().get());
                break;
            case ORDERPRIORITY_STR:
                orderpriority = decode(String.class,
                  record.getValue().get());
                break;
            case CLERK_STR:
                clerk = decode(String.class, record.getValue().get());
                break;
            case SHIPPRIORITY_STR:
                shippriority = decode(Long.class, record.getValue().get());
                break;
            case COMMENT_STR:
                comment = decode(String.class, record.getValue().get());
                break;
            default:
                throw new RuntimeException("Unknown qualifier " + colQual);
        }
    }
```

After we have processed the row, we format our output string and print it. Once all of the rows have been processed, we close the data scanner. That is our data retrieval application.

```
    System.out.println(format("%d|%d|%s|%f|%s|%s|%s|%d|%s",
            orderkey, custkey, orderstatus, totalprice, orderdate,
            orderpriority, clerk, shippriority, comment));
}

// Close the batch scanner
dataScanner.close();
```

After we run the ingestion program, we can run the clerk search program to retrieve records from the orders table that were rung up by given clerks. When looking for all the orders rung up by clerks 1 and 2, we receive 2,961 rows in 2.195 seconds.

As we can see, this approach to implementing an Accumulo query application in Java is quite verbose. Additionally, should you want to issue another query, you need to re-write your application. Should you want to write a complex query containing joins, grouping, ordering, and/or additional predicates, you will need to plan significant development time to complete your application.

With the Presto connector, querying data in Accumulo is vastly simplified, and the connector provides a framework for advanced ad-hoc queries using ANSI SQL. We'll now take a look at a Presto approach to our application.

Clerk Search - A Presto Application

After installing and starting Presto, we open a connection using the Presto CLI tool and create a table to store our data. We include a table property called `index_columns` to tell the Accumulo connector that we want to index data in the `clerk` column.

```
$ presto --server localhost:8080 --catalog accumulo --schema default
presto:default> CREATE TABLE orders (
                    orderkey BIGINT,
                    custkey BIGINT,
                    orderstatus VARCHAR,
                    totalprice DOUBLE,
                    orderdate DATE,
                    orderpriority VARCHAR,
                    clerk VARCHAR,
                    shippriority BIGINT,
                    comment VARCHAR
                ) WITH (
                    index_columns = 'clerk'
                );
```

This command will create the orders table in Accumulo as well as the table used to store the index. After the table has been created, we have two choices for data ingestion. We could issue a series of INSERT statements to have the connector write the code, but this is not a high-throughput operation. Instead, we will use the `PrestoBatchWriter` that is provided with the connector.

The `PrestoBatchWriter` class is intended to be used programmatically, but can also be executed as a command-line program to ingest delimited lines of data into Accumulo. It is a wrapper for the regular Accumulo `BatchWriter`, with the added benefit of using Presto metadata to automatically index columns of data. Additionally, the `PrestoBatchWriter` tracks metrics about the index table to allow the connector to make decisions on whether or not to use the index (via configurable heuristics).

We'll compile and invoke this tool against the orders data set to ingest the data into Accumulo, telling the tool we want to ingest `orders.tbl` into the `default.orders` table, and that it is pipe-delimited data. This process of ingestion is really no different from our previous Java application -- it is the querying interface that drastically changes.

```
$ java -jar target/presto-accumulo-tools-0.142-ANY.jar batchwriter \
      -s default -t orders -d \| -f orders.tbl
... log messages ...
28360 [main] INFO  tools.PrestoBatchWriter: Wrote 1500000 mutations to table
```

Now that we have data in our table, we can issue SQL statements using the Presto CLI to interact with our data.

```
presto:default> SELECT * FROM orders
                WHERE clerk IN ('Clerk#000000001', 'Clerk#000000002');
```

Using practically the same process as programmed before, the connector knows the 'clerk' column is indexed by the metadata associated with the `orders` table. The connector scans the index column for all the order IDs processed by clerks one or two, then it queries the data table in parallel to scan the main data table. This query executes in 1.8 seconds – 15 percent faster than our previous Java application.

Certainly a much easier interface -- we didn’t need to deal with writing any Java code to query the tables. Additionally, we now have a very powerful SQL interface to work with our orders table -- but why stop there when we can quickly create the remaining TPC-H tables, ingest the data, and then run the TPC-H benchmark test suite! In the next section, we’ll take a look at some performance numbers running the TPC-H benchmarks.

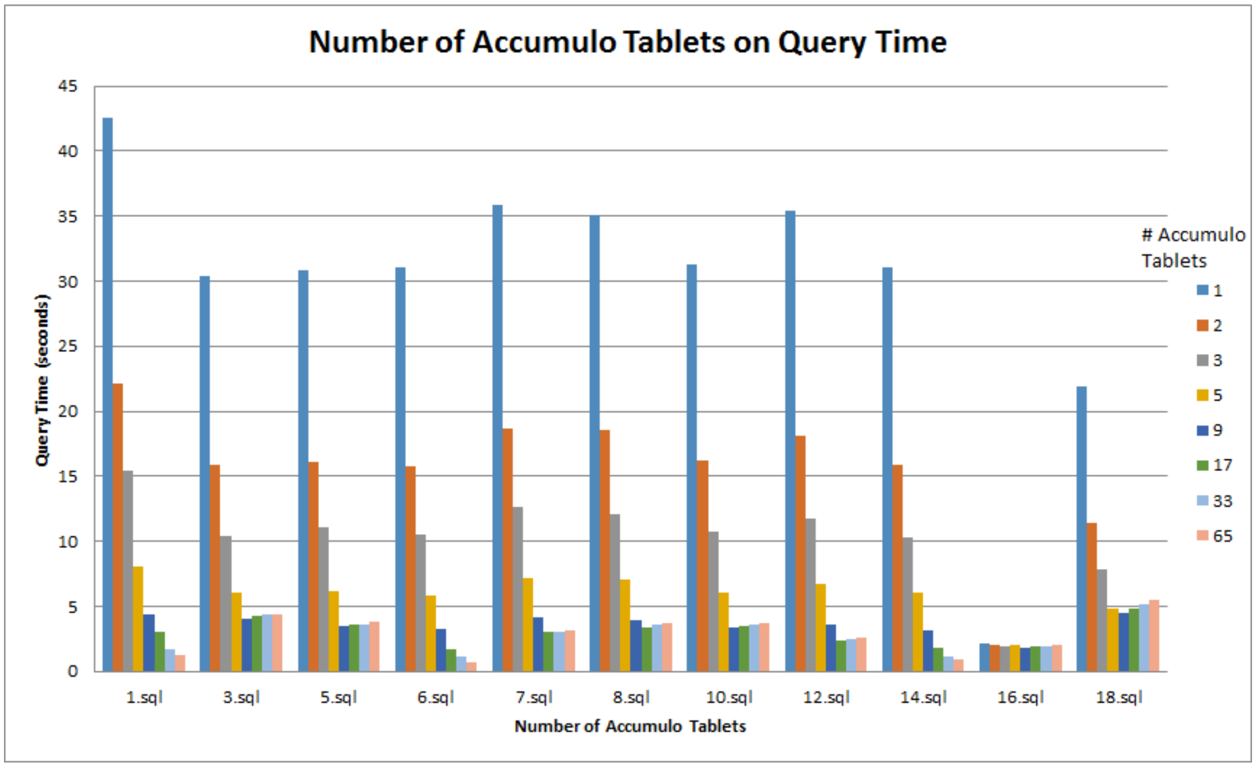
TPC-H Benchmark

The TPC-H query set contains 22 queries, often joining multiple tables together, grouping data, ordering data, and calculating aggregates. Of the 22 queries, Presto can execute 13 of them. The remaining nine require functionality that Presto does not yet support.

Effect of Tablet Splits on Query Time

The purpose of this test is to see if the query time is improved as the number of Accumulo tablets in each table increases. Here, the TPC-H benchmark suite is executed via Presto. Each query is run three times and the numbers shown below are the average for the three runs. The Accumulo tables were then split into two tablets, and the suite was run again. This process continues exponentially up to 64 split points (65 tablets).

This graph displays the query runtime of each of the above mentioned queries. As the number of Accumulo tablets increases, you can see that the execution time changes linearly as is expected. The blue line is the baseline query with only one tablet (no splits). The orange line is the query time with one Accumulo split, which created two tablets and cut the query time nearly in half.



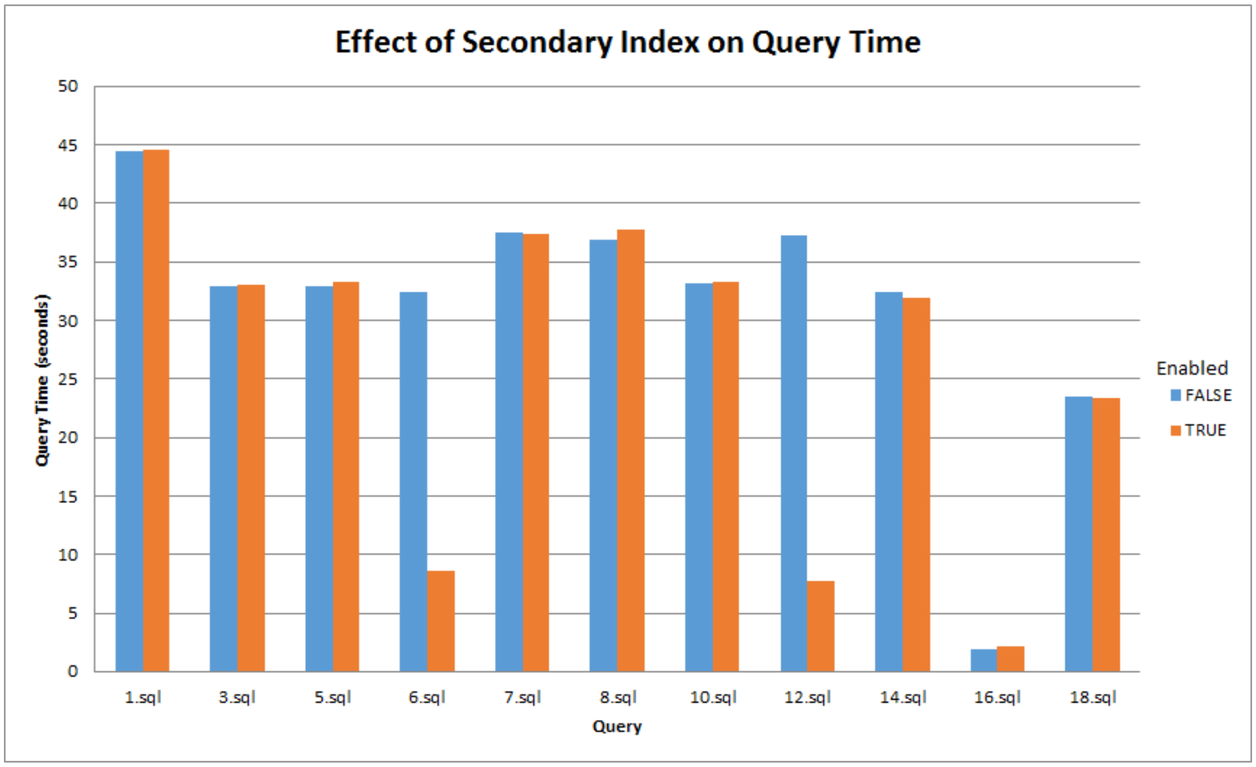
As the number of tablets is increased, the query time continues to drop, until around 17 tablets (16 split points). However, increasing the number of tablets further, does not reduce the query time. This is either because the Tablet Servers are saturated and unable to return data fast enough, or the improvement in the scan time is negligible and most of the runtime is spent planning the query, returning the results, and executing any non-TableScan nodes in the query plan.

In short, creating tablets allows Presto workers to scan the Accumulo tables in parallel. The connector creates one Presto split per Accumulo tablet; with only one tablet a single Presto worker is responsible for scanning the entire table. As we split the table, the connector can create multiple scan tasks, improving the scan time and therefore the overall query time. An Accumulo table should always be split appropriately to improve the scan time of a Presto query that would scan the entire table.

Effect of Secondary Indexing on Query Time

This test leverages the indexing scheme built into the Presto connector. The intention of this test is to demonstrate how beneficial the secondary indexing can be for queries which contain predicates that will limit the amount of data retrieved from an individual table.

The below graph plots each of the selected TPC-H queries with the secondary indexing disabled (in blue) and enabled (in orange).



The only queries here that benefit from the secondary indexing at all are queries 6 and 12. In query 6, 1.3% of the largest TPC-H table, lineitem, is actually scanned when the predicates are taken into consideration. Query 12 scans 4.4% of the lineitem table. When we enable the secondary indexing, only these small percentages of the tables are scanned.

For the remaining queries, 100% of the tables in the queries are scanned. When the secondary index is used, each distinct row ID is packed into a Presto split, 10,000 row IDs per split, and then sent off to a Presto worker to be pulled from Accumulo using a `BatchScanner`. We are basically executing a full table scan, but requesting every single row ID instead of just scanning the entire table using fewer Accumulo Range objects.

It is not surprising that the queries which scan 100% of the table do not derive any benefit from the secondary indexing. Only queries which scan a small number of rows from a table can leverage the secondary indexing. This optimization is tunable using session parameters which can be set for each distinct query.

Conclusion

By using the Presto Connector for Apache Accumulo, users are able to execute efficient ANSI SQL queries against relational data sets for rapid exploration and production analytics. This drastically reduces the development time to extract data from Accumulo. The methodologies used by the Presto Connector follow common design patterns used by Accumulo application developers today.

Features of the Presto-Accumulo Connector

- Creating/dropping tables backed by Accumulo, including specifying locality groups and indexed columns
- Creating/dropping views
- Inserting data using SQL
- High-throughput ingestion of data via `PrestoBatchWriter`
- Predicate pushdowns on row ID column and indexed columns
- External table support for existing data sets
- Store values as strings or using Accumulo's lexicoder API
- Scan-time authorizations via a table property
- Scan-time user impersonation via a session property
- Additional session properties to enable/disable optimizations and tune the indexing heuristics
- Pluggable metadata storage if not using Zookeeper

If you find this interesting, we'd love to hear from you – and we are hiring!