Document Store on HBase for Better Query Processing

Jie Huang, Shengsheng Huang, Jason Dai
Intel Corporation

1. Problem Statement

High level Data warehousing systems built on top of Hadoop (e.g., Hive) significantly lowers the barrier to MapReduce, by allowing users to analyze their data using a high level query language based on the relational model. In the last couple of years, increasingly these systems have transitioned to a (semi) realtime analytics system built around HBase; one such example is Chukwa, which begin to store data directly in HBase for analysis. This makes several new use cases possible:

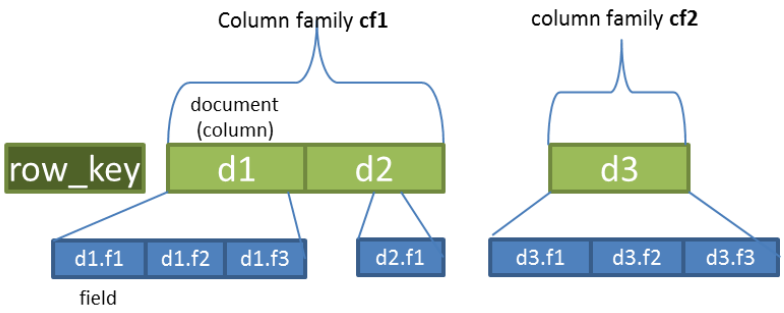
- *Stream* new data into HBase in near realtime for processing (versus loading new data into the HDFS cluster in large batch, e.g., every few hours).
- Support *high-update rate* workloads (by directly updating existing data in HBase) to keep the warehouse always up to date.
- Allow every low latency, online data serving (e.g., to power an online web service).

On the other hand, while HBase already has very effective MapReduce integration with its good scanning performance, query processing using MapReduce on HBase still has significant gaps compared to query processing on HDFS.

- *Space overheads.* To provide flexible schema support, physically HBase stores its table as a multi-dimensional map, where each cell (except the row key) is stored on disk as a key-value pair: *(row_id, family:column, timestamp)* → *cell*. On the other hand, a Hive table has a fixed relational model, and consequently HBase can introduce large space overheads (as large as 3x) compared to storing the same table in HDFS.
- *Query performance*. Query processing on HBase can be much (sometimes 3~5x) slower than that on HDFS due to various reasons. One of the reason is related to how HBase handles data accesses – HBase provides very good support for high concurrent read/write accesses; consequently, one needs to pay some amount of overheads (e.g., concurrency control) for each column read. On the other hand, data accesses in analytical query processing are predominantly read (with some append), and should preferably avoid the column read overheads.

2. Document Store on HBase

To address these issues, we propose to implement *a document store on HBase*, which can greatly improve query processing on HBase (by leveraging the relational model and read-mostly access patterns). The figure below illustrates the data model of the document store.



In the document store, a table can be declared as a *document-oriented table* (DOT) at the table creation time. Each row in DOT contains, in addition to the *row key*, a collection of *documents* (doc), and each document contains a collection of *fields*; in query processing, each column in a relational table is mapped to a field in some document. Physically, each document is encoded using a serialization framework (such as Avro or Protocol Buffers), and its schema is stored separately (just once); consequently, the storage overheads can be greatly reduced. In addition, each document is mapped to an HBase column and is the unit for update; consequently the associated read overheads can be amortized across different fields in a document. According to our experiments, DOT can reduce storage space by up-to ~3x and speedup query processing by up-to ~2x.

3. Co-processor Based Implementation

The document store can be implemented using HBase co-processors as follows.

3.1 Data definition operations

When creating a DOT, the user is required to specify the schema and serializer (e.g., Avro) for each document in the table; this information can then be store in the table metadata (e.g., table descriptor or column family descriptor) by the *preCreateTable* co-processor. For instance, for the DOT shown in the previous section, the code to create the table is illustrated below:

```
HTableDescriptor desc = new HTableDescriptor("t1");
desc.setValue("hbase.dot.enable","true");     //Specify a dot table
desc.setValue("hbase.dot.type","ANALYTICAL");//The only type supported at
                                              //this moment is ANALYTICAL

HColumnDescriptor cf1Desc = new HColumnDescriptor(Bytes.toBytes("cf1"));
cf1Desc.setValue("hbase.dot.columnfamily.doc.element",
                 "d1,d2");      //Specify the documents in the column family
String doc1Schema = " {    \n" + " \"name\": \"d1\", \n"
       + " \"type\": \"record\",\n" + " \"fields\": [\n"
       + "   {\"name\": \"f1\", \"type\": \"bytes\"},\n"
       + "   {\"name\": \"f2\", \"type\": \"bytes\"},\n"
       + "   {\"name\": \"f3\", \"type\": \"bytes\"} ]\n" + "}";
cf1Desc.setValue("hbase.dot.columnfamily.doc.schema.d1",
                 doc1Schema);       //specify the schema for d1
String doc2Schema = " {    \n" + " \"name\": \"d2\", \n"
       + " \"type\": \"record\",\n" + " \"fields\": [\n"
       + "   {\"name\": \"f1\", \"type\": \"bytes\"} ]\n" + "}";
cf1Desc.setValue("hbase.dot.columnfamily.doc.schema.d2",
                 doc2Schema);       //specify the schema for d2
desc.addFamily(cf1Desc);

HColumnDescriptor cf2Desc = new HColumnDescriptor(Bytes.toBytes("cf2"));
cf2Desc.setValue("hbase.dot.columnfamily.doc.element",
                 "d3");     //Specify the documents in the column family
String doc3Schema = " {    \n" + " \"name\": \"d3\", \n"
       + " \"type\": \"record\",\n" + " \"fields\": [\n"
       + "   {\"name\": \"f1\", \"type\": \"bytes\"},\n"
       + "   {\"name\": \"f2\", \"type\": \"bytes\"},\n"
       + "   {\"name\": \"f3\", \"type\": \"bytes\"} ]\n" + "}";
cf2Desc.setValue("hbase.dot.columnfamily.doc.schema.d3",
                 doc3Schema);       //specify the schema for d3
desc.addFamily(cf2Desc);

admin.createTable(desc);
```

A consequence of this is that the documents contained in each DOT and the associate schema of each document are required to be fixed and predetermined at table creation time. This turns out to be a reasonable requirement for (Hive) query processing, as tables in data warehouse always have a fixed relation model.

3.2 Data access operations (get/scan/put/delete)

The users of DOT (e.g., a Hive query) can access individual fields in the document-oriented table in the same way they access individual columns in a conventional HBase table – just specifying "doc.field" in place of "column qualifier" in Get/Scan/Put/Delete, as illustrated below:

```
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, "t1");
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("d1.f1")).
addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("d3.f1")).
setStartRow(Bytes.toBytes("row-10")).
setStopRow(Bytes.toBytes("row-20"));
ResultScanner scanner = table.getScanner(scan);
for (Result res : scanner) {
  System.out.println(res);
}
scanner.close();
```

On the other hand, each document is stored as a column in DOT. Therefore, before these data access operations can be performed by the region server, the associated co-processers (*preGet*, *preScannerOpen*, *prePut* and *PreDelete*) need to dynamically rewrite the Get/Scan/Put/Delete objects, mapping the fields to associated documents (i.e., column qualifiers) – e.g., for the above example, the *preScannerOpen* co-processor rewrites the Scan object to contain two columns: "d1" and "d3".

In addition, each document is mapped to an HBase column and is required to be the update unit; therefore the *prePut* and *PreDelete* also need to check if all the fields in a document are provided in the Put/Delete object as follows:

- If a field is missing in the Put object, a warning is given and a NULL value is assigned to the missing field.
- If a field is missing in the Delete object, an error is given.

The data access operations can then be performed by the region server using the transformed Get/Scan/Put/Delete objects. After the read (Get/Scan) operations are completed, the associated co-processors (*postGet* and *postScannerClose*) need to extracted related fields from the document read from the region, and then return the field list to the user.

3.3 HBase Filters

The users can continue to use HBase filters on DOT in the same way they use filter on a conventional HBase table – specifying "doc.field" in place of "column qualifier" in the supplied filter:

```
Scan scan = new Scan();
SingleColumnValueFilter filter = new SingleColumnValueFilter(
       Bytes.toBytes("cf1"), Bytes.toBytes("d1.f1"),
       CompareFilter.CompareOp.EQUAL, new SubstringComparator("row1_fd1"));
scan.setFilter(filter);
HTable table = new HTable(conf, "t1");
ResultScanner scanner = table.getScanner(scan);
for (Result result : scanner) {
    …
}
```

The co-processor framework needs to be extended to provide observers for the filter operations, similar to the observers of the data access operations. For DOT support, these filter observers need to first extracted individual fields from the document read from the region, and pass the fields to the related filter operations; after the filter operations, the filter observers need to record the decisions returned by the filter operations, which will be later used in the *postGet* and *postScannerClose* co-processors to extract related fields to return.