O'REILLY®

Architecting HBase Applications

A GUIDEBOOK FOR SUCCESSFUL DEVELOPMENT AND DESIGN

Jean-Marc Spaggiari & Kevin O'Dell

Architecting HBase Applications

Jean-Marc Spaggiari and Kevin O'Dell

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Architecting HBase Applications
by Jean-Marc Spaggiari and Kevin O'Dell

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Marie Beaugureau
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2015: First Edition

Revision History for the First Edition
2015-08-25: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491915813* for release details.

978-1-491-91581-3

[LSI]

Table of Contents

CHAPTER 1

Underlying storage engine - Description

The first use case that will be examined is from Omneo a division of Camstar a Siemens Company. Omneo is a big data analytics platform that assimilates data from disparate sources to provide a 360-degree view of product quality data across the supply chain. Manufacturers of all sizes are confronted with massive amounts of data, and manufacturing data sets comprise the key attributes of Big Data. These data sets are high volume, rapidly generated and come in many varieties. When tracking products built through a complex, multi-tier supply chain, the challenges are exacerbated by the lack of a centralized data store and no unified data format. Omneo ingests data from all areas of the supply chain, such as manufacturing, test, assembly, repair, service, and field.

Omneo offers this system to their end customer as a Software as a Service(SaaS) model. This platform must provide users the ability to investigate product quality issues, analyze contributing factors and identify items for containment and control. The ability to offer a rapid turn-around for early detection and correction of problems can result in greatly reduced costs and significantly improved consumer brand confidence. Omneo must start by building a unified data model that links the data sources so users can explore the factors that impact product quality throughout the product lifecycle. Furthermore, Omneo has to provide a centralized data store, and applications that facilitate the analysis of all product quality data in a single, unified environment.

Omneo evaluated numerous NoSQL systems and other data platforms. Omneo's parent company Camstar has been in business for over 30 years, giving them a well established IT operations system. When Omneo was created they were given carte blanche to build their own system. Knowing the daunting task of handling all of the data at hand, they decided against building a traditional EDW. They also looked at other big data technologies such as Cassandra and MongoDB, but ended up selecting

Hadoop as the foundation for the Omneo platform. The primary reason for the decision came down to ecosystem or lack thereof from the other technologies. The fully integrated ecosystem that Hadoop offered with MapReduce, HBase, Solr, and Impala allowed Omneo to handle the data in a single platform without the need to migrate the data between disparate systems.

The solution must be able to handle numerous products and customer's data being ingested and processed on the same cluster. This can make handling data volumes and sizing quite precarious as one customer could provide eighty to ninety percent of the total records. As of writing this Omneo hosts multiple customers on the same cluster for a rough record count of +6 billion records stored in ~50 nodes. The total combined set of data in the HDFS filesystem is approximately 100TBs. This is important to note as we get into the overall architecture of the system we will note where duplicating data is mandatory and where savings can be introduced by using a unified data format.

Omneo has fully embraced the Hadoop ecosystem for their overall architecture. It would only make sense for the architecture to also takes advantage of Hadoop's Avro data serialization system. Avro is a popular file format for storing data in the Hadoop world. Avro allows for a schema to be stored with data, making it easier for different processing systems such as MapReduce, HBase, and Impala/Hive to easily access the data without serializing and deserializing the data over and over again.

The high level Omneo architecture is shown below:

- Ingest/pre-processing
- Processing/Serving
- User Experience

Ingest/pre-processing

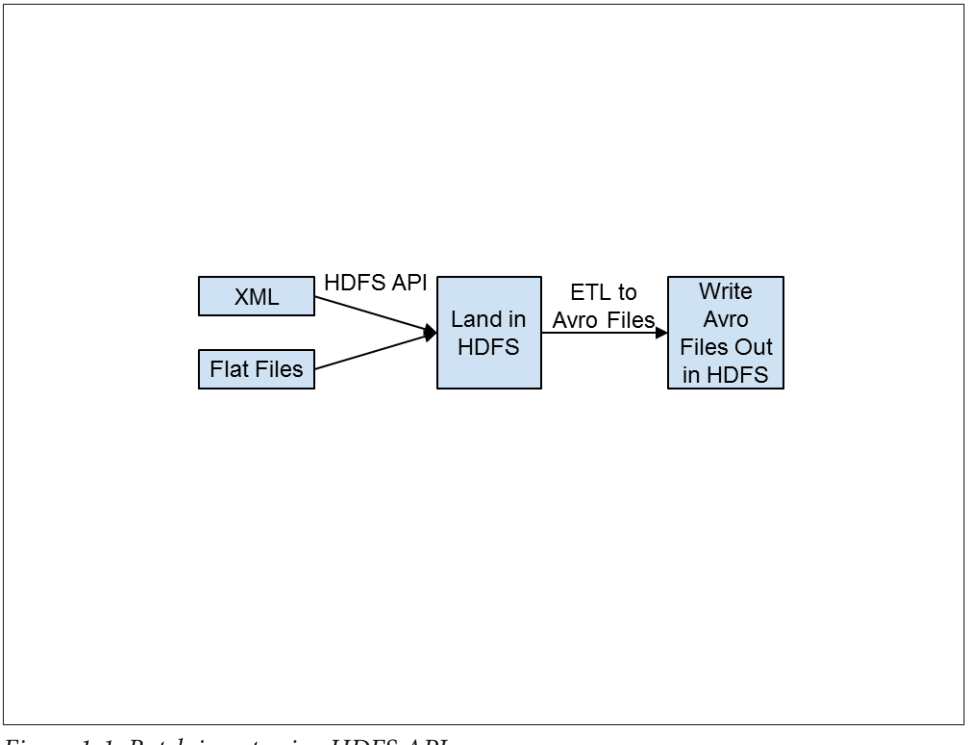

Figure 1-1. Batch ingest using HDFS API

The ingest/pre-processing phase includes acquiring the flat files, landing them in HDFS, and converting the files into Avro. As noted in the above diagram, Omneo currently receives all files in a batch manner. The files arrive in a CSV format or in a XML file format. The files are loaded into HDFS through the HDFS API. Once the files are loaded into Hadoop a series of transformations are performed to join the relevant data sets together. Most of these joins are done based on a primary key in the data. In the case of electronic manufacturing this is normally a serial number to identify the product throughout its lifecycle. These transformations are all handled through the MapReduce framework. Omneo wanted to provide a graphical interface for consultants to integrate the data rather than code custom mapReduce. To accomplish this they partnered with Pentaho to expedite time to production. Once the data has been transformed and joined together it is then serialized into the Avro format.

Processing/Serving

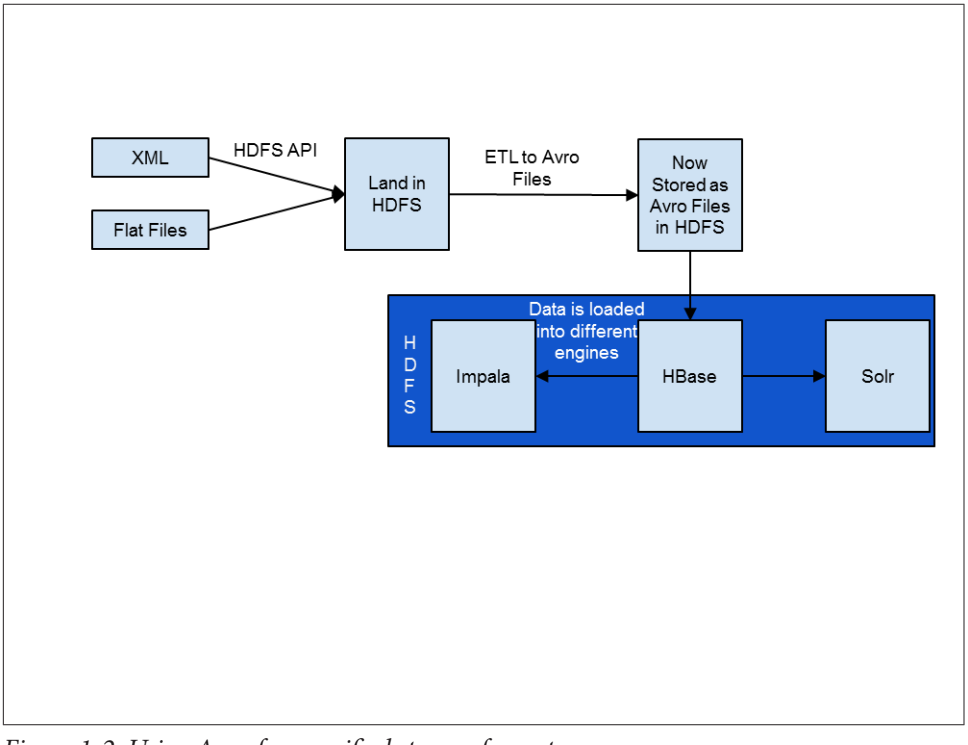

Figure 1-2. Using Avro for a unified storage format

Once the data has been converted into Avro it is loaded into HBase. Since the data is already being presented to Omneo in batch, we take advantage of this and use bulk loads. The data is loaded into a temporary HBase table using the bulk loading tool. **The previously mentioned MapReduce jobs output HFiles that are ready to be loaded into HBase.** The HFiles are loaded through the `completebulkload` tool. The `completebulkload` works by passing in a URL, which the tool uses to locate the files in HDFS. Next, the bulk load tool will load each file into the relevant region being served by each RegionServer. Occasionally a region has been split after the HFiles were created, and the bulk load tool will automatically split the new HFile according to the correct region boundaries.

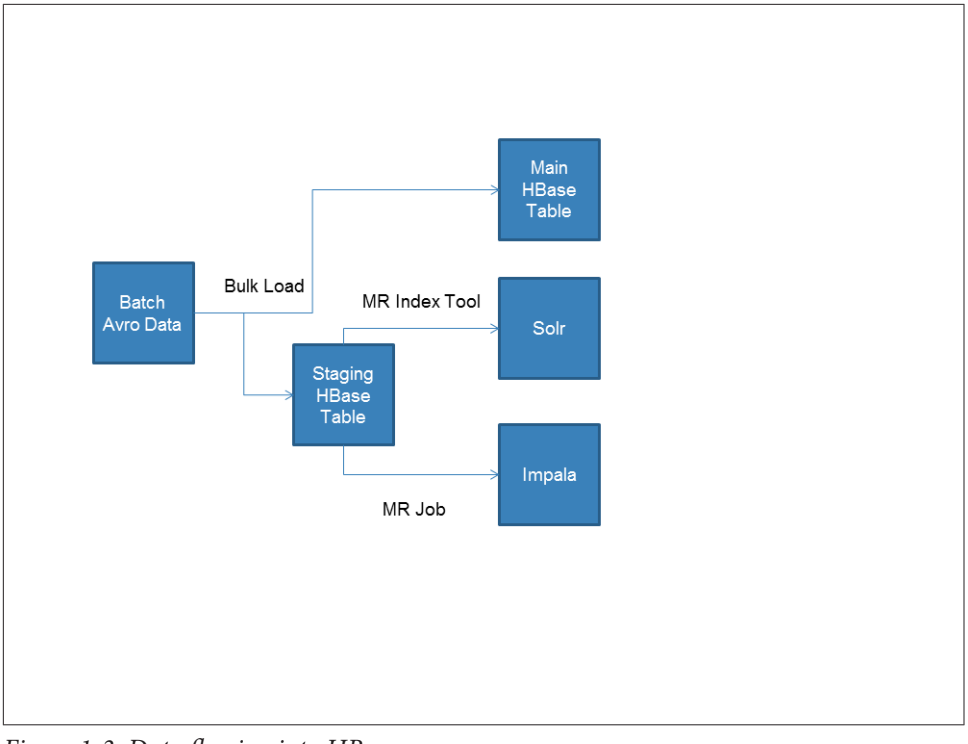

Figure 1-3. Data flowing into HBase

Once the data is loaded into the staging table it is then pushed out into two ofthe main serving engines Solr and Impala. Omneo is using a staging table to limit the amount of data read from HBase to feed the other processing engines. The reason behind using a staging table lies in the HBase key design. One of the cool things about this HBase use case is the simplicity of the schema design. Normally many hours will be spent figuring out the best way to build a composite key that will allow for the most efficient access patterns, and we will discuss composite keys in the later chapters.

However, in this use case the row key is a simple MD5 hash of the product serial number. Each column stores an Avro record. The column name contains the unique ID of the Avro record it stores. The Avro record is a de-normalized data set containing all attributes for the record.

After the data is loaded into the staging HBase table, it is then propagated into two other serving engines. The first serving engine is Cloudera Search(Solr) and the second is Impala. Here is a diagram showcasing the overall load of data into Solr:

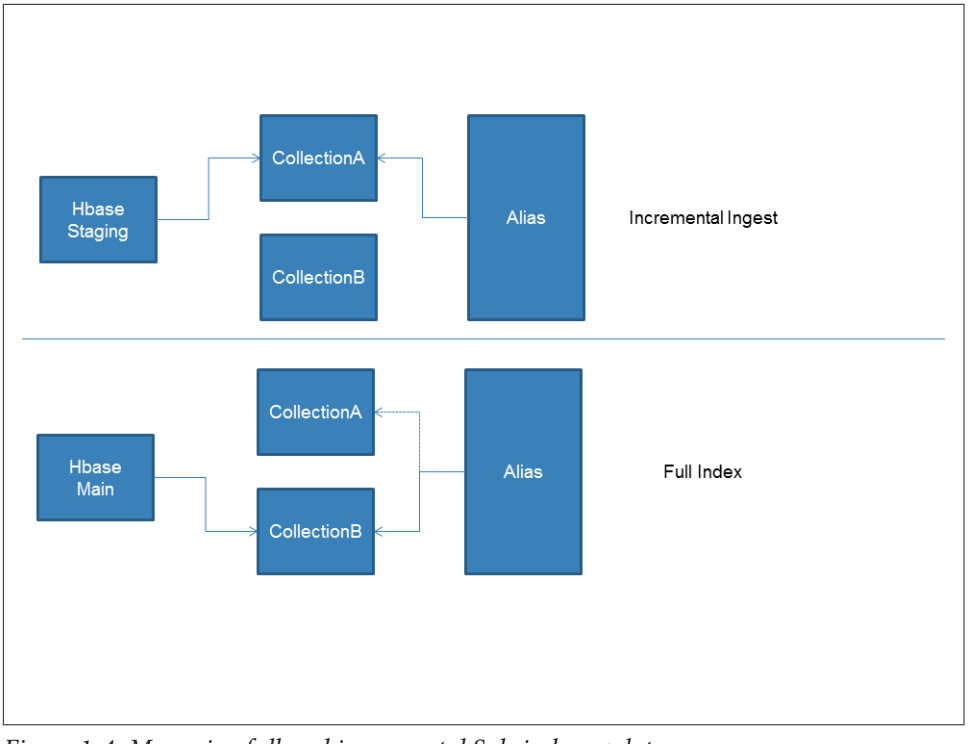

Figure 1-4. Managing full and incremental Solr index updates

The data is loaded into Search through the use of a custom `MapReduceIndexerTool`. The default nature of the `MapReduceIndexerTool` is to work on flat files being read from HDFS. Given the batch aspect of the Omneo use case, they modified the indexer tool to read from HBase and write directly into the Solr Collection through the use of MapReduce. The above diagram illustrates the two flows of the data from HBase into the Solr Collections. There are two collections in play for each customer, in this case there is CollectionA(active), CollectionB(backup), and an alias that links to the "active" Collection. During the incremental index only the current Collection is updated from the staging HBase table through the use of the `MapReduceIndexerTool`. In the above diagram the HBase staging table is loading into CollectionA and the alias is pointing to the active Collection (CollectionA). The dual collections with an alias approach offers the ability to drop all of the documents in a single collection and reprocess the data without suffering an outage. This gives Omneo the ability to alter the schema and push it out to production without taking more downtime.

Part two of the above diagram illustrates this action; the `MapReduceIndexerTool` is re-indexing the main HBase table into CollectionB while the alias is still pointing to CollectionA. Once the indexing step complete, the alias will be swapped to point at

CollectionB and incremental indexing will be pointed at CollectionB until the dataset needs to be re-indexed again.

This is where the use case really gets interesting. HBase serves two main functions in the overall architecture. The first one is to handle the MDM(master data management) since it allows updates. In this case, HBase is the system of record that Impala and Solr use. If there is an issue with the Impala or Solr datasets, they will rebuild them against the HBase dataset. In HBase attempting to redefine the row key typically results in having to rebuild the entire dataset. Omneo first attempted to tackle faster lookups for secondary and tertiary fields by leveraging composite keys. It turns out the end user likes to change the primary lookups based on the metrics they are looking at. This is one of the reasons Omneo avoided leveraging composite keys, and used Solr to add extra indexes to HBase. The second and most important piece is HBase actually stores all of the records being served to the end user. Lets look at a couple sample fields from Omneo's Solr schema.xml:

```
<schema name="example" version="1.5">
<fields>
   <field name="id" type="string" indexed="true" stored="true" required="true"
                                                          multiValued="false" />
   <field name="rowkey"  type="binary" indexed="false" stored="true"
                                          omitNorms="true" required="true"/>
   <field name="eventid" type="string" indexed="true"  stored="false"
                                          omitNorms="true" required="true"/>
   <field name="docType" type="string" indexed="true" stored="false"
                                                          omitNorms="true"/>
   <field name="partName" type="lowercase" indexed="true" stored="false"
                                                          omitNorms="true"/>
   <field name="partNumber" type="lowercase" indexed="true" stored="false"
                                                          omitNorms="true"/>
…
   <field name="_version_" type="long" indexed="true" stored="true"/>
</fields>
```

Looking at some of the fields in the schema.xml shown above we can see that Omneo is only flagging the HBase rowkey and the required Solr fields(id and *version*) as stored which will directly write these results to HDFS. The other fields are flagged as indexed; which will store the data in a special index directory. The index field makes a field searchable, sortable, and facetable, it is also stored in memory. The stored fields are fields retrievable through search and persisted to HDFS file system. The typical records that Omneo ingests can have many columns presents in the data ranging from 100s to 1000s of columns depending on the product being ingested. For the purposes of faceting and natural language searching typically only a small subset of those fields are necessary. The amount of fields indexed will vary per customer and use cases. This is a very common pattern as the actual data results displayed to the customer are being served from the application calling scans and multigets from HBase based on the stored data. Just indexing the fields serves two purposes:

- All of the data and facets are served out of memory offering tighter and more predictable SLAs.
- The current state of Solr Cloud on HDFS writes the data to HDFS per shard and replica. If HDFS replication is set to the default factor of 3, then a shard with two replicas will have 9 copies of the data on HDFS. This will not normally affect a Search deployment as memory or CPU is normally the bottleneck before storage, but it will use more storage.
- Indexing the fields offers lighting fast counts to the overall counts for the indexed fields. This feature can help to avoid costly SQL or pre-HBase MapReduce based aggregations

The data is also loaded from HBase into Impala tables from the Avro schemas and converted into the Parquet file format. Impala is used as Omneo's data-warehouse for the end users. The data is populated in the same manner as the Solr data with incremental updates being loaded from the HBase staging table and full rebuilds being pulled from the main HBase table. As the data is pulled from the HBase tables it is denormalized into a handful tables to allow for an access pattern conducive to Impala. The model used is another portion of the secret sauce of Omneo's business model. The model is shown below:

Figure 1-5. Nah, we can't share that. Get your own!

User Experience

Normally we do not spend a ton of time looking at the end application as they tend to be quite different per application, but in this case it is important to discuss how everything comes together. Combining the different engines together in a streamlined user experience is the big data dream. This is how companies move from playing around to truly delivering a monumental product.

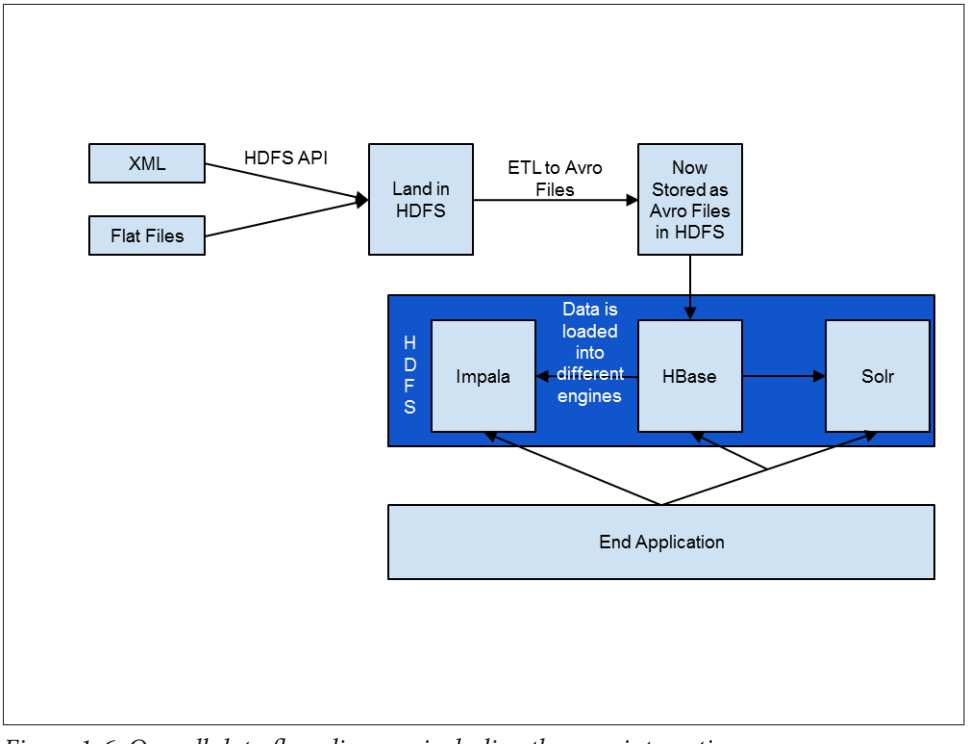

Figure 1-6. Overall data flow diagram including the user interaction

The application makes use of all three of the serving engines in a single interface. This is important for a couple of key reasons. The first is, increased productivity from the analyst. The analyst no longer has to switch between different UIs, or CLIs. Second, the analyst is able to use the right tool for the right job. One of the major issues we see in the field is customers attempting to use one tool to answer all of the questions. By allowing Solr to serve facets and handle natural language searches, HBase to serve the full fidelity records, and Impala to handle the aggregations and SQL questions Omneo is able to offer the analyst a 360 degree view of the data.

Figure 1-7. Check out this Solr Goodness

Let's start by looking at the Solr/HBase side of the house. These are the two most intertwined services of the Omneo application. As mentioned before, Solr stores the actual HBase Row Key and indexes the vast majority of other fields that the users like to search and facet against. In this case as the user drills down or adds new facets the raw records are not served back from Solr, but rather pulled from HBase using a multiget of the top 50 records. This allows the analyst to see the full fidelity record being produced by the facets and searches. The same thing holds true if the analyst wishes to export the records to a flat file, the application will call a scan of the HBase table and write out the results for end user.

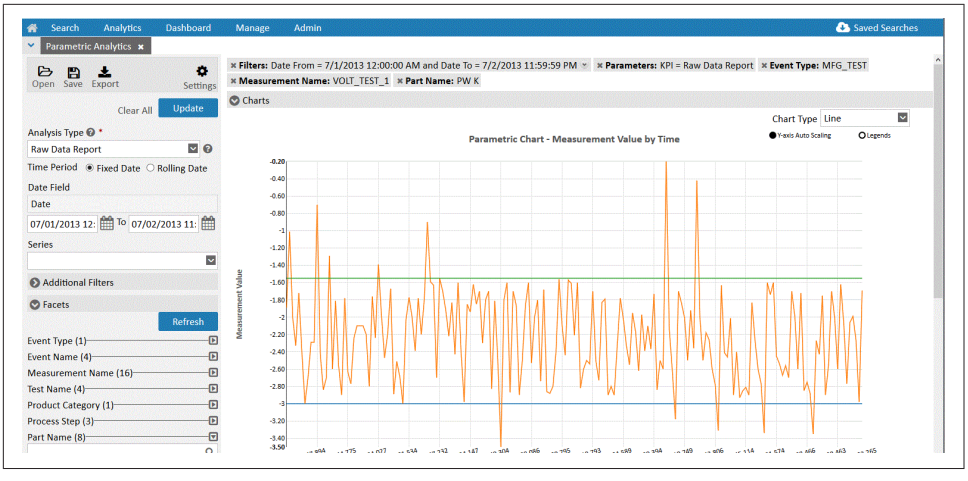

Figure 1-8. Leveraging Impala for custom analytics

On the Impala side of the house, also known as Performance Analytics, models are built and managed to handle SQL like workloads. These are workloads that would normally be forced into HBase or Solr. Performance Analytics was designed to run a set of pre-packed application queries that can be run against the data to calculate Key Performance Indicators(KPIs). The solution does not allow for random free-form SQL queries to be utilized as long running rogue queries can cause performance degradation in a multi-tenant application. In the end the users can select the KPIs they want to look at, and add extra functions to the queries (sums, avg, max, etc).

CHAPTER 2

Underlying storage engine - Implementation

In the previous chapter, we described how Omneo uses the different Hadoop technologies to implement their use-case. In this chapter we will look in detail at all the different parts involving HBase. Implementation will not go into each and every detail but will give you all the required tools and examples to understand what is important in this phase.

As usual when implementing an HBase project, the first thing we consider is the table schema which is the most important part of every HBase project. This can sometimes be easy, like for the current use-case, but can also sometimes require a lot of time. It is a good practice to always start with this task, keeping in mind how data is received from your application (write path) and how you will need to retrieve it (read path). Read and write access patterns will dictate most of the table design.

Table design

Table schema

Table design for the Omneo use-case is pretty easy, but let's work through the steps so you can apply a similar approach to your own table schema design. We want both read and write paths to be efficient. In Omneo's case, data is received from external systems in bulk. Therefore, unlike other ingestion patterns where data is inserted one single value at a time, here it can be processed directly in bulk format and doesn't require single random writes or updates based on the key. On the read side, the user needs to be able to retrieve all the information for a specific sensor very quickly by searching on any combination of sensor id, event id, date and event type. There is no way we can design a key to allow all those retrieval criteria to be efficient. We will

have to rely on an external index which, given all of our criteria, will give us back a key we will use to query HBase. Given that the key will be retrieved from this external index and that we don't have to look-up or scan for it, we can simply use a hash of the sensor ID, with the column qualifier being the event ID. You can refer to "Generate Test Data" on page 22 to have a preview of the data format.

Sensors can have very similar IDs, such as 42, 43, 44. However, sensor IDs can also have a wide range (e.g. 40000-49000). If we use the original sensor ID as the key, we might encounter hot-spots on specific regions due to the keys' sequential nature. You can read more about hot-spotting in ???.

Hashing keys

One option to deal with hotspotting is to simply pre-split the table based on those different known IDs to make sure they are correctly distributed accross the cluster. However, what if in the future, distribution of those IDs changes? Then splits might not be correct anymore, and we might end-up again by hot-spotting some regions. If today all IDs are between 40xxxx and 49xxxx, regions will be split from the beginning to 41, 41 to 42, 42 to 43, etc. But if tomorrow a new group of sensors is added with IDs from 40xxx to 39xxx, they will end up in the first region. Since it is not possible to forecast what the future IDs will be, we need to find a solution to ensure a good distribution whatever the IDs will be. When hashing data, even 2 initially close keys will produce a very different result. From our example above, 42 will produce `50a2fabfdd276f573ff97ace8b11c5f4` as its md5 hash, while 43 will produce `f0287f33eba7192e2a9c6a14f829aa1a`. As you can see, unlike the original sensor IDs 42 and 43, sorting those two md5 hashes puts them far from one another. And even if new IDs are coming, since they are now translated into a hexadecimal value, they will always be distributed between 0 and F. Using such a hashing approach will ensure a good distribution of the data across all the regions while given a specific sensor ID, we still have direct access to its data.

The hash approach can not be used when you need to scan your data keeping the initial order of the key, as the md5 version of the key perturbs the original ordering, distributing the rows throughout the table.

Column qualifier

Regarding the column qualifier, the event ID will be used. The event ID is a hash value received from the downstream system, unique for the given event for this specific sensor. Each event has a specific type such as "alert", "warning", or "RMA" (Return Merchandise Authorization). At first, we considered using the event type as a column qualifier. However, a sensor can encounter a single event type multiple times. Each "warning" a sensor encountered would overwrite the previous "warning", unless

we used HBase's "versions" feature. Using the unique event ID as the column qualifier allows us to have multiple events with the same type for the same sensor being stored without having to code extra logic to use HBase's "versions" feature to retrieve all of a sensor's events.

Table parameters

Most HBase table parameters should be considered to improve performance. However, only the parameters that apply to this specific use-case are listed in this section. The list of all the existing parameters for table creation are available in chapter ???.

Compression

The first parameter we examine is the compression algorithm used when writing table data to disk. HBase writes the data into HFiles in a block format. Each block is 64 KB by default, and is not compressed. Blocks store the data belonging to one region and column family. A table's data contains related information and usually has common pattern. Compressing those blocks can almost always give good results. As an example, it will be good to compress column families containing logs and customer information. HBase supports multiple compression algorithms: LZO, GZ (for GZip), SNAPPY and LZ4. Each compression algorithm will have its own pros and cons. For each algorithm, consider the performance impact of compressing and decompressing the data versus the compression ratio, i.e. was the data sufficiently compressed to warrant running the compression algorithm.

Snappy will be very fast in all operations but will have a very low compression ratio, while GZ will be more resource intensive but will compress better. The algorithm you will choose depends on your use-case. It is recommended to test a few of them on a sample dataset to validate compression rate and performance. As an example, a 1.6GB CSV file generates 2.2GB of uncompressed HFiles while from the exact same dataset it uses only 1.5GB with LZ4. Snappy compressed HFiles for the same dataset take 1.5GB too. Since read and write latency are important for us, we will use Snappy for our table. Be aware of the availability of the various compression libraries on various Linux distributions. For example, Debian does not include Snapply libraries by default. Due to licensing, LZO and LZ4 libraries are usually not bundled with common Apache Hadoop distributions, and must be installed separately.

Keep in mind that compression ratio might vary based on the data type. Indeed, if you try to compress a text file, it will compress much better than a PNG image. For example, a 143,976 byte PNG file will only compress to 143,812 bytes (a 2.3% space saving) whereas a 143,509 byte XML file can compress as small as 6,284 bytes (a 95.7% space saving!) It is recommended that you test the different algorithms on your dataset before selecting one. If the compression ratio is not significant, avoid using compression and save processor overhead.

Data block encoding

Data block encoding is an HBase feature where keys are encoded and compressed based on the previous key. One of the encoding options (FAST_DIFF) is to ask HBase to store only the difference between the current key and the previous one. HBase stores each cell individually, with its key and value. When a row has many cells, much space can be consumed by writing the same key for each cell. Therefore activating the data block encoding can allow important space saving. It is almost always helpful to activate data block encoding, so if you are not sure, activate the FAST_DIFF encoding. Since for the current use-case, for a given row, we can have thousands of columns, we will benefit from this encoding. The current use-case will benefit from this encoding since a given row can have thousands of columns.

You can refer to ??? for more information.

Bloom filter

Bloom filters are useful in reducing unnecessary IO by skipping input files from HBase regions. A Bloom filter will tell HBase if a given key **might be** or **is not** in a given file. But it doesn't mean the key is definitively included in the file.

However, there are certain situations where Bloom filters are not required. For the current use-case, files are loaded once a day then a major compaction is run on the table. As a result, there will almost always be only a single file per region. Also, queries to the HBase table will be based on results returned by SOLR. This means read requests will always succeed and return a value. Because of that, the Bloom filter will always return true and HBase will always open the file. As a result, for this specific use-case, the Bloom filter will be an overhead and is not required.

Bloom filers will be covered with more details in ???.

Since Bloom filters are activated by default, in this case we will need to explicitly disable them.

Pre splitting

Pre-splits are not really table parameters. Pre-splits information is not stored within the table meta information and is used only at the table creation time. However, this is so important that we want to talk about it here before you even move to the implementation. Pre-splitting a table means asking HBase to split the table into multiple regions when it is created. HBase comes with different pre-split algorithms described in ???. The goal of pre-splitting a table is to make sure the initial load will be correctly distributed across all the regions and will not hotspot a single region. Granted, data would be distributed over time as region splits occur automatically, but pre-splitting provides the distribution from the onset.

Implementation

Now that we have decided which parameters we want to set for our table, it's time to create it. We will keep all the default parameters except the ones we just talked about. Run the following command in the HBase shell to create a table called "sensors" with a single column family and the parameters we discussed above, pre-split into 15 regions. NUMREGIONS and SPLITALGO are the two parameters used to instruct HBase to pre-split the table.

```
hbase(main):001:0> create 'sensors', {NUMREGIONS => 15,\
                                      SPLITALGO => 'HexStringSplit'}, \
                                     {NAME => 'v', COMPRESSION => 'SNAPPY',\
                                      BLOOMFILTER => 'NONE',\
                                      DATA_BLOCK_ENCODING => 'FAST_DIFF'}
```

Please refer to ??? for details of available parameters when creating a table using the shell and the Java API.

When your table is created, you can see its details using the HBase WebUI interface or the following shell command:

```
hbase(main):002:0> describe 'sensors'
Table sensors is ENABLED
sensors
COLUMN FAMILIES DESCRIPTION
{NAME => 'v', DATA_BLOCK_ENCODING => 'FAST_DIFF', BLOOMFILTER => 'NONE',
REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'SNAPPY',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
1 row(s) in 0.1410 seconds
```

NUMREGIONS and SPLIALGO parameters are used for the table creation but are not stored within the meta data of the table. If is not possible to retrieve this information after the table has been created.

As you can see, the parameters we specified are listed in the output, along with the default table parameters. Default parameters might vary based on the HBase version you are using. What is important here is to note that `BLOOMFILTER`, `DATA_BLOCK_ENCODING` and `COMPRESSION` are configured as we asked.

Now that we have our table ready, we can move forward with the data preparation.

Data conversion

Generate Test Data

The next goal is to generate a set of representative test data to run through our process and verify the results. The first thing we will create is some data files with test values. The goal is to have a dataset to allow you to run the different commands and programs.

In the examples, you will find a class called CSVGenerator which creates data resembling the example below:

```
1b87,58f67b33-5264-456e-938a-9d2e9c5f4db8,ALERT,NE-565,0-0000-000,1,ECEGYFFLQIOV ...
3244,350cee9e-55fc-409d-b389-6780a8af9e76,RETURNED,NE-382,0-0000-000,1,OOQTYQSDT ...
727d,b97df483-f0bd-4f24-8ff3-6988d8eff88c,ALERT,NE-858,0-0000-000,1,MSWOCQXMHWPO ...
53d4,d8c39bf8-6f5f-4311-8ee5-9d3bce3e18d7,RETURNED,NE-881,0-0000-000,1,PMKMWSVPB ...
1fa8,4a0bf5b3-680d-4b87-8d9e-e55f06614ae4,ALERT,NE-523,0-0000-000,1,IYIZSHKAXNRY ...
```

Each line contains a random sensor id comprised of 4 characters (0 to 65535, represented in hexadecimal), then a random event id, document type, part name, part number, version and a payload formed of random letters (64 to 128 characters long). To generate a different workload, you can re-run the `CSVGenerator` code anytime you want. Subsequent parts of the example code will read this file from the ~/ahae/resources/ch09 folder. This class will create files relative to where it's run, therefore we need to run the class from the ~/ahae folder. If you want to increase or reduce the size of the data set, simply update the following line:

```
for (int index = 0; index < 1000000; index++) {
```

You can run this data generator directly from Eclipse without any parameter or from the shell into the ahae folder using the following command:

```
hbase -classpath ~/ahae/target/ahae.jar com.architecting.ch09.CSVGenerator
```

This will create a file under called ~/ahae/resources/ch09/omneo.csv.

Create avro schema

Now that we have some data to start with, we need to define an Avro schema that will reflect the format of the data generated. Based on the search schema provided in the previous chapter, we will need the following Avro schema:

```
{"namespace": "com.architecting.ch09",
 "type": "record",
 "name": "Event",
 "fields": [
     {"name": "id", "type": "string"},
     {"name": "eventid",  "type": "string"},
     {"name": "docType",  "type": "string"},
     {"name": "partName",  "type": "string"},
     {"name": "partNumber",  "type": "string"},
     {"name": "version",  "type": "long"},
     {"name": "payload",  "type": "string"}
 ]
}
```

You can find the schema in the resources/ch09 directory of the examples under the name omneo.avsc. Since it has already been compiled and imported into the project, it is not required to compile it. However, if you want to modify it, you can recompile it using the following command:

```
java -jar ~/ahae/lib/avro-tools-1.7.7.jar compile schema omneo.avsc ~/ahae/src/
```

This creates the file ~/ahae/src/com/architecting/ch09/Event.java containing the Event object that will be used to store the Event Avro object into HBase.

Implement MapReduce transformation

The first steps of the production process is to parse the received CSV file to generate HBase HFiles, which will be the input to the next step. They will map the format of the previously created table.

Our production data will be large files, so we will implement this transformation using MapReduce to benefit from parallelism. Input of this MR job will be the text file and the output will be the HFiles. This dictates the way you should configure your MR job.

Example 2-1. Convert to HFiles example

```
Table table = connection.getTable(tableName);

Job job = Job.getInstance(conf, "ConvertToHFiles: Convert CSV to HFiles");

HFileOutputFormat2.configureIncrementalLoad(job, table,
                                  connection.getRegionLocator(tableName)); ❶
job.setInputFormatClass(TextInputFormat.class); ❷

job.setJarByClass(ConvertToHFiles.class); ❸
job.setJar("/home/cloudera/ahae/target/ahae.jar"); ❸

job.setMapperClass(ConvertToHFilesMapper.class); ❹
job.setMapOutputKeyClass(ImmutableBytesWritable.class); ❺
```

```
job.setMapOutputValueClass(KeyValue.class); ❻

FileInputFormat.setInputPaths(job, inputPath);
HFileOutputFormat2.setOutputPath(job, new Path(outputPath));
```

❶ HBase provides a helper class which will do most of the configuration for you. This is the first thing to call when you want to configure your MR job to provide HFiles as the output.

❷ Here we want to read a text file with CSV data, so we will use the TextInputFormat.

❸ When running from the command line, all the required classes are bundled into a client jar which is referenced by the `setJarByClass` method. However, when running from Eclipse, it is necessary to manually provide the jar path because the class that we are running is from the eclipse environment which MapReduce is not aware of. Because of that, we need to give to MapReduce the path of an external file where the given class is also available.

❹ Defines the mapper you want to use to parse your CSV content and create the Avro output.

❺ We need to define ImmutableBytesWritable as the mapper output key class. It is the format we will use to write the key.

❻ We need to define KeyValue as the mapper output value class. This will represent the data we want to store into our HFiles.

The reducer used to create the HFiles needs to load into memory the columns of a single row and then sort all before being able to write them all. If you have **many** columns in your dataset, it might not fit into memory. This should be fixed in a future release when HBASE-13897 will be implemented.

The operations on the mapper side are simple. The goal is just to split the line into different fields, assign them to an Avro object and provide this Avro object to the HBase framework to be stored into HFiles ready to be loaded.

The first thing we do is to define a set of variables that we will re-use for each and every iteration of the mapper. This is done to reduce the number of objects created.

Example 2-2. Convert to HFiles mapper

```
public static final EncoderFactory encoderFactory = EncoderFactory.get();
public static final ByteArrayOutputStream out = new ByteArrayOutputStream();
public static final DatumWriter<Event> writer = new SpecificDatumWriter<Event>
                                                    (Event.getClassSchema());
public static final BinaryEncoder encoder = encoderFactory.binaryEncoder(out, null);
public static final Event event = new Event();
public static final ImmutableBytesWritable rowKey = new ImmutableBytesWritable();
```

Those objects are all re-used on the map method below.

Example 2-3. Convert to HFiles mapper

```
// Extract the different fields from the received line.
String[] line = value.toString().split(","); ❶

event.setId(line[0]);
event.setEventId(line[1]);
event.setDocType(line[2]);
event.setPartName(line[3]);
event.setPartNumber(line[4]);
event.setVersion(Long.parseLong(line[5]));
event.setPayload(line[6]); ❷

// Serialize the AVRO object into a ByteArray
out.reset(); ❸
writer.write(event, encoder); ❹
encoder.flush();

byte[] rowKeyBytes = DigestUtils.md5(line[0]);
rowKey.set(rowKeyBytes); ❺
context.getCounter("Convert", line[2]).increment(1);

KeyValue kv = new KeyValue(rowKeyBytes, CF, Bytes.toBytes(line[1]), out.toByteArray()); ❻
context.write (rowKey, kv); ❼
```

❶ The first thing we do is to split the line into fields to have individual direct access to each of them.

❷ Instead of creating a new Avro object at each iteration, we re-use the same object for all the map calls and simply assign it the new received values.

❸ This is another example of object reuse. The less objects you create in your mapper code, the less garbage collection you will have to do and the faster your code will execute. The `map` method is called for each and every line of your input file. Creating a single `ByteArrayOutputStream` and re-using it and its internal buffer for each `map` iteration saves millions of object creations.

❹ Serialize the Avro object into an array of bytes to store them into HBase re-using existing objects as much as possible.

❺ Construct our HBase key from the sensor ID.

❻ Construct our HBase KeyValue object from our key, our column family, our eventid as the column qualifier and our Avro object as the value.

❼ Emit our KeyValue object so the reducers can regroup them and write the required HFiles. The rowKey will only be used for partitioning the data. When data will be written into the underlying files, only the KeyValue data will be used for both the key and the value.

When implementing a MapReduce job, avoid creating objects when not required. If you need to access a small subset of fields in a String, it is not recommended to use the String split() method to extract the fields. Using split() on 10 million Strings having 50 fields each will create 500 million objects that will be garbage collected. Instead, parse the String to find the few fields' locations and use the substring() method. Also consider using the com.google.common.base.Splitter object from Guava libraries.

Again, the example can be run directly from Eclipse or from the command line. In both cases, you will need to specify the input file, the output folder and the table name as the parameters. The table name is required for HBase to find the region's boundaries to create the required splits in the output data, but also to lookup the column family parameters like the compression and the encoding. The MapReduce job will produce HFiles in the output folder based on the table regions and the column family parameters.

The following command line will create the HFiles on HDFS. If because you are running on the standalone version you need the files to be generated on local disk, simply update the destination folder.

```
hbase -classpath ~/ahae/target/ahae.jar:`hbase classpath` \
com.architecting.ch09.ConvertToHFiles \ ❶
file:///home/cloudera/ahae/resources/ch09/omneo.csv \ ❷
hdfs://localhost/user/cloudera/ch09/hfiles/ sensors ❸
```

❶ the class called for the conversion.

❷ our input file

❸ output folder and table name.

If you start the class from Eclipse, make sure to add the parameters in Run … Run Configurations / Arguments.

Since this will start a MapReduce job, the output will be verbose and will give you lots of information. Pay attention to the following lines:

```
Map-Reduce Framework
        Map input records=1000000
        Map output records=1000000
        Reduce input groups=65536
```

The `Map input records` value represents the number of lines in your CSV file. Since for each line we emit one and only one Avro object, it matches the value of the `Map output records` counter. The `Reduce input groups` represents the number of unique keys. So here we can see that there were one million lines for 65,536 different rows, which gives us an average of 15 columns per row.

At the end of this process, your folder content should look like the following:

```
[cloudera@quickstart ~]$ hadoop fs -ls -R ch09/
drwxr-xr-x         0 2015-05-08 19:23 ch09/hfiles
-rw-r--r--         0 2015-05-08 19:23 ch09/hfiles/_SUCCESS
drwxr-xr-x         0 2015-05-08 19:23 ch09/hfiles/v
-rw-r--r-- 104861200 2015-05-18 19:57 ch09/hfiles/v/345c5c462c6e4ff6875c3185ec84c48e
-rw-r--r-- 104784750 2015-05-18 19:56 ch09/hfiles/v/46d20246053042bb86163cbd3f9cd5fe
-rw-r--r-- 104920812 2015-05-18 19:56 ch09/hfiles/v/6419434351d24624ae9a49c51860c80a
-rw-r--r-- 104753687 2015-05-18 19:57 ch09/hfiles/v/680f817240c94f9c83f6e9f720e503e1
-rw-r--r-- 104750096 2015-05-18 19:58 ch09/hfiles/v/69f6de3c5aa24872943a7907dcabba8f
-rw-r--r-- 105088024 2015-05-18 19:56 ch09/hfiles/v/75a255632b44420a8462773624c30f45
-rw-r--r-- 104557019 2015-05-18 19:56 ch09/hfiles/v/7c4125bfa37740ab911ce37069517a36
-rw-r--r-- 104982419 2015-05-18 19:57 ch09/hfiles/v/9accdf87a00d4fd68b30ebf9d7fa3827
-rw-r--r-- 105013848 2015-05-18 19:58 ch09/hfiles/v/9ee5c28cf8e1460c8872f9048577dace
-rw-r--r-- 104935349 2015-05-18 19:57 ch09/hfiles/v/c0adc6cfceef49f9b1401d5d03226c12
-rw-r--r-- 104899602 2015-05-18 19:57 ch09/hfiles/v/c0c9e4483988476ab23b991496d8c0d5
-rw-r--r-- 104828814 2015-05-18 19:58 ch09/hfiles/v/ccb61f16feb24b4c9502b9523f1b02fe
-rw-r--r-- 105049861 2015-05-18 19:56 ch09/hfiles/v/d39aeea4377c4d76a43369eb15a22bff
-rw-r--r-- 104805384 2015-05-18 19:57 ch09/hfiles/v/d3b4efbec7f140d1b2dc20a589f7a507
-rw-r--r-- 104794836 2015-05-18 19:56 ch09/hfiles/v/ed40f94ee09b434ea1c55538e0632837
```

Owner and group information were removed to fit the page. All the files belong to the user who has started the MapReduce job.

As you can see in the file system, the MapReduce job created as many HFiles as we have regions in the table.

When generating the input files, be careful to provide the correct column family. Indeed, it a common mistake to not provide the right column family name to the MapReduce job which will create the directory structure based on its name. This will make the Bulk-Load phase to fail.

The folder within which the files are stored is named based on the column family name we have specified in our code, "v" in the given example.

HFile validation

Throughout the process all the information we get in the console is related to the MapReduce framework and tasks. However, even if they succeed, the content they have generated might not be good. Maybe we used the wrong column family, maybe we forgot to configure the compression when we have created our table, etc.

HBase comes with a tool to read HFiles and extract the meta information. This tool is called the `HFilePrettyPrinter` and can be called by using the following command line:

```
hbase hfile -printmeta -f ch09/hfiles/v/345c5c462c6e4ff6875c3185ec84c48e
```

The only parameter this tool takes is the HFile location in HDFS.

Below is part of the output of the previous command. We removed sections or parts of them which are not relevant for this chapter.

```
Block index size as per heapsize: 161264
reader=ch09/hfiles/v/345c5c462c6e4ff6875c3185ec84c48e,
    compression=snappy, ❶
    cacheConf=CacheConfig:disabled,
    firstKey=7778/v:03afef80-7918-4a46-a903-f6e35b629926/1432004229936/Put, ❷
    lastKey=8888/v:fc69a89f-4a78-4e2d-ae0a-b22dc93c962c/1432004229936/Put, ❸
    avgKeyLen=53, ❹
    avgValueLen=171, ❺
    entries=666591, ❻
    length=104861200 ❼
```

Now here is what is important to look at in this output.

❶ This shows you the compression format used for your file. It should reflect what you have configured when you created the table. We initially chose to use snappy, but if you have configured a different one, you should see it here.

❷ Key of the first cell of this HFile, as well as column family name.

❸ The last key contained in the HFile. Only keys between 7778 and 8888 are present in the file. It is used by HBase to skip entire files when the key you are looking for is not between the first and last key.

❹ Average size of the keys.

❺ Average size of the values.

❻ Number of cells present in the HFile.

❼ Total size of the HFile.

Using the output of this command, you can validate there is data in the files you have generated and the format of the data is according to your expectations (compression, bloom filters, average key size, etc.)

Bulk loading

Bulk loading inserts multiple pre-generated HFiles into HBase instead of performing puts one-by-one using the HBase API. Bulk loads are the most efficient way to insert a large quantity of values into the system. To get more details about bulk loading, please refer to later chapter ???. Here we will show you how to perform a bulk load.

Here is the HDFS content of your table.

```
  0 2015-05-18 19:46 .../s/.tabledesc
287 2015-05-18 19:46 .../s/.tabledesc/.tableinfo.0000000001
  0 2015-05-18 19:46 .../s/.tmp
  0 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd
 58 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd/.regioninfo
  0 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd/recovered.edits
  0 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd/recovered.edits/2.seqid
  0 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd/v
```

To fit the page width, file permissions, owner were removed, and /hbase/data/default/sensors was abbreviated to …/s.

If your table is empty, you will still have all the region folders because we have pre-split the table. HFiles might be present in the regions' folders if data already existed prior to loading. We show only one region's directory in the above extract and you can see that this region's column family `v` is empty since it doesn't contain any HFiles.

Our HFiles have been generated by the MapReduce job, and we now need to tell HBase to place the HFiles into the given table. This is done using the following command:

```
hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles ch09/hfiles sensors
```

In this command, we provide HBase the location of the HFiles we have generated (ch09/hfiles) and the table into which we want to insert those files (sensors). If the target table splits or merges some regions before the files are bulk loaded, splits and merges of the input HFiles will be handled on the client side at that time by the application. Indeed, the application used to push the HFiles into the HBase table will validate that each and every HFile still belongs to a single region. If a region got split before we pushed the file, the load tool will split the input files the same way before

pushing them into the table. On the other side, if 2 regions are merged, the belonging input HFiles are simply going to be pushed into the same region.

When it runs, it will produce this output in the console:

```
$ hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles ch09/hfiles sensors
2015-05-18 20:09:29,701 WARN  [main] mapreduce.LoadIncrementalHFiles: Skipping
    non-directory hdfs://quickstart.cloudera:8020/user/cloudera/ch09/hfiles/_SUCCESS
2015-05-18 20:09:29,768 INFO  [main] Configuration.deprecation: hadoop.native.lib is
    deprecated. Instead, use io.native.lib.available
2015-05-18 20:09:30,476 INFO  [LoadIncrementalHFiles-0] compress.CodecPool: Got
    brand-new decompressor [.snappy]
```

After completion of the bulk load, you should find your files in HDFS under the table and the regions they belong too. Looking again at HDFS should show you something like this:

```
         0 2015-05-18 19:46 .../s/0cc...
        58 2015-05-18 19:46 .../s/0cc.../.regioninfo
         0 2015-05-18 19:46 .../s/0cc.../recovered.edits
         0 2015-05-18 19:46 .../s/0cc.../recovered.edits/2.seqid
         0 2015-05-18 20:09 .../s/0cc.../v
104794836 2015-05-18 19:56 .../s/0cc.../v/c0ab6873aa184cbb89c6f9d02db69e4b_SeqId_4_ (1)
```

Again, to fit the page width, file permissions, owner were removed, and /hbase/data/default/sensors was abbreviated to …/s. We have also truncated the region encoded name.

(1) You can see that we now have a file in our previously empty region. This is one of the HFiles we have initially created. By looking at the size of this file and by comparing it to the initial HFiles created by the MapReduce job, we can match it to ch09/hfiles/v/ed40f94ee09b434ea1c55538e0632837. You can also look at the other regions and map them to the other input HFiles.

Data validation

Now that data is in the table we need to verify that it is expected. The first thing we will do is to make sure we have as many rows as expected. Then we will verify the records contain what we expect.

Table size

Looking into an HFile using the `HFilePrettyPrinter` gives us the number of cells within a single HFile, but how many unique rows does it really represent? Since an HFile only represents a subset of rows, we need to count rows at the table level. HBase provides two different mechanisms to count the rows.

Counting from the shell

Counting the rows from the shell is pretty straightforward, simple, and efficient for small examples. It will simply do a full table scan and count the rows one by one. It works well for small tables, however it can take a lot of time for big tables, so we will use this method only when we are sure our tables are small.

Here is the command to count our rows and we will look at the parameters after:

```
hbase(main):003:0> count 'sensors', INTERVAL => 40000, CACHE => 40000
Current count: 40000, row: 9c3f
65536 row(s) in 1.1870 seconds
```

The count command takes one to three parameters. The first parameter is mandatory, it is the name of the table whose rows you want to count. The second parameter is optional, it tells the shell to display a progress status only every 40,000 rows. The final parameter is optional too, it is the size of the cache we want to use to do our full table scan. This last value is used to setup the `setCaching` value of the underlying `scan` object.

Counting from MapReduce

The second way to count the number of rows in an HBase table is to use the RowCounter MapReduce tool. The big benefit of using MapReduce to count your rows is HBase will create one mapper per region in your table. For a very big table this will distribute the work on multiple nodes to perform the count operation in parallel instead of scanning regions sequentially, which is what the shell's count command does.

This tool is called from the command line by passing the table name only:

```
hbase org.apache.hadoop.hbase.mapreduce.RowCounter sensors
```

Here is the most important part of the output and we will detail below the important fields to look at. Some sections of the output have been removed in order to focus attention on key information and to reduce the size of the extract.

```
2015-05-18 20:21:02,493 INFO  [main] mapreduce.Job: Counters: 31
        Map-Reduce Framework
                Map input records=65536 ❶
                Map output records=0
                Input split bytes=1304
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=2446
                CPU time spent (ms)=48640
                Physical memory (bytes) snapshot=3187818496
                Virtual memory (bytes) snapshot=24042749952
                Total committed heap usage (bytes)=3864526848
```

```
        org.apache.hadoop.hbase.mapreduce.RowCounter$RowCounterMapper$Counters
                ROWS=65536 ❷
```

❶ Since the input records of this job are the HBase rows, we will have as many input records as we have rows.

❷ The number of rows we have in the table, which will match the number of input records. Indeed, this MapReduce job simply increments a ROWS counter for each input record.

HBase also provides a MapReduce tool called CellCounter to count not just the number of rows in a table, but also the number of columns and the number of versions for each of them. However, this tool needs to create a Hadoop counter for each and every unique row key found in the table. Hadoop has a default limit of 120 counters. It is possible to increase this limit, but increasing it to the number of rows we have in the table might create some issues. If you are working on a small dataset, this might be useful to test your application and debug it. This tool generally cannot be run on a big table.

File content

We have our table with the number of lines we expected and the format we asked. But what does the data in the table really look like? Are we able to read what we wrote? Let's see two ways to have a look at our data.

Using the shell

The easiest and quickest way to read data from HBase is to use the HBase shell. Using the shell, you can issue commands to retrieve the data you want. The first command is `get` which will give you a single row. If you specify a column family, it will return only the colums for this family. If you specify both a column family and a column qualifier (separated with a colon), it will return only the specific value if it exists. The second option is to use `scan` which will return a certain number of rows that we can limit using the `LIMIT` parameter or the STARTROW and STOPROW parameters. Both commands below will return all the columns for the row with rowkey value 000a:

```
get 'sensors', '000a', {COLUMN => 'v'}
scan 'sensors', {COLUMNS => ['v'], STARTROW => '000a', LIMIT => 1 }
```

Now as you will see in the output, there might be many columns for each row. If you want to limit the output to a specific column qualifier, you need to specify it in both commands the following way:

```
get 'sensors', '000a', {COLUMN => 'v:f92acb5b-079a-42bc-913a-657f270a3dc1'}
scan 'sensors', { COLUMNS => ['v:f92acb5b-079a-42bc-913a-657f270a3dc1'], \
                  STARTROW => '000a', STOPROW => '000a' }
```

The output of the get should then look like this:

```
COLUMN       CELL
 v:f9acb...  timestamp=1432088038576, value=\x08000aHf92acb5b-079a-42bc-913a-657...
1 row(s) in 0.0180 seconds
```

Since the value is an Avro object, it contains some non-printable characters which are displayed as `\x08`, but most of it should still be readable. This shows us that our table containts some data, with a key which is what we have expected and data which looks like what we are looking for.

Using Java

Using the shell we have been able to validate that our table contains some data, looking like `Avro` data, but to make sure it is exactly what we are expecting, we will need to implement a piece of Java code to retrieve the value, convert it into an `Avro` object and retrieve the fields from it.

Example 2-4. Read Avro object from HBase example

```
try (Connection connection = ConnectionFactory.createConnection(config);
     Table sensorsTable = connection.getTable(sensorsTableName)) {  ❶
  Scan scan = new Scan ();
  scan.setCaching(1); ❷

  ResultScanner scanner = sensorsTable.getScanner(scan);
  Result result = scanner.next(); ❸
  if (result != null && !result.isEmpty()) { ❹
    Event event = new Util().cellToEvent(result.listCells().get(0), null); ❺
    LOG.info("Retrived AVRO content: " + event.toString());
  } else {
    LOG.error("Impossible to find requested cell");
  }
}
```

❶ Retrieves the table from the HBase connection.

❷ Make sure we return from the scan after we get the first row. Since we don't want to print more than that, there is no need to wait for HBase to send us back more data.

❸ Executes the scan against the table and get the result.

❹ Validates if we have a result or if the response is empty.

❺ Transforms the cell we received as the value into an Avro object.

Once again, you can run this example from Eclipse or from the command line. The output you will get will look like:

```
2015-05-20 18:30:24,214 INFO  [main] ch09.ReadFromHBase: Retrieved Avro object
    with ID 000a
2015-05-20 18:30:24,215 INFO  [main] ch09.ReadFromHBase: Avro content: {"id":
    "000a", "eventid": "f92acb5b-079a-42bc-913a-657f270a3dc1", "docType": "FAILURE",
    "partName": "NE-858", "partNumber": "0-0000-000", "version": 1, "payload":
    "SXOAXTPSIUFPPNUCIEVQGCIZHCEJBKGWINHKIHFRHWHNATAHAHQBFRAYLOAMQEGKLNZIFM 000a"}
```

With this very small piece of code we have been able to perform the last step of the validation process and retrieved, de-serialized and printed an Avro object from the table. To summarize, we have validated the size of the HFiles, their format, the numbers of entries in the HFiles and in the table, and the table content itself. We can now confirm that our data has been correctly and fully loaded into the table.

Data indexing

The next and last step of the implementation consists of indexing the table we have just loaded, to be able to quickly search for any of the records using SOLR. Indexation is an incremental process. Indeed, Omneo receive new files daily. As seen in the previous chapter, data from those files is loaded into a main table which contains data from the previous days, and an indexation table. The goal is to add the indexation result into the SOLR index build from previous days indexations. At the end, the index will reference all what has been uploaded in the main table. To implement this last example you will need to have a SOLR instance running on your environment. If you are comfortable with it, you can install it and run it locally, however HBase needs to run in pseudo-distributed mode since the SOLR indexer can not work with the local jobrunner. Alternatively, you can execute this example in a VM where SOLR is already installed.

Most of the MapReduce indexing code has been built from the SOLR examples and has been modified and simplified to index an HBase table.

Once you have confirmed you have a working local SOLR environment, running the following command will create our SOLR collection with a single shard and the provided schema. In a production environment, to scale your application, you might want to consider using more shards.

The most important file to define your index is it's schema.xml file. This file is available in the book online resources and contains many tags. The most important section of the schema is the following:

```
<field name="id" type="string" indexed="true" stored="true" required="true"
                                                          multiValued="false" />
```

```
<field name="rowkey" type="binary" indexed="false" stored="true" omitNorms="true"
                                                                  required="true"/>
<field name="eventId" type="string" indexed="true" stored="false"
                                                 omitNorms="true" required="true"/>
<field name="docType" type="string" indexed="true" stored="false"
                                                                  omitNorms="true"/>
<field name="partName" type="lowercase" indexed="true" stored="false"
                                                                  omitNorms="true"/>
<field name="partNumber" type="lowercase" indexed="true" stored="false"
                                                                  omitNorms="true"/>
<field name="version" type="long" indexed="true" stored="false" required="true"
                                                                 multiValued="false" />
<field name="payload" type="string" indexed="true" stored="false" required="true"
                                                                 multiValued="false" />
<field name="_version_" type="long" indexed="true" stored="true"/>
```

Because the scope of this book is focused on HBase, we can not go into all the details of this file and all its fields and the invite you to look at the SOLR online documentation.[1]

The following commands will create the required index for the examples:

```
export PROJECT_HOME=~/ahae/resources/ch09/search
rm -rf $PROJECT_HOME
solrctl instancedir --generate $PROJECT_HOME
mv $PROJECT_HOME/conf/schema.xml $PROJECT_HOME/conf/schema.old
cp $PROJECT_HOME/../schema.xml $PROJECT_HOME/conf/
solrctl instancedir --create Ch09-Collection $PROJECT_HOME
solrctl collection --create Ch09-Collection -s 1
```

If for any reason you want to delete your collection, you can use the following commands:

```
solrctl collection --delete Ch09-Collection
solrctl instancedir --delete Ch09-Collection
solrctl instancedir --delete search
```

The steps to get the table indexed are pretty straight forward. The first thing we need to do is to scan the entire HBase table using MapReduce to create SOLR index files. The second step is to bulkload those files into SOLR similar to how we bulkloaded our HFiles into HBase. The entire code will not be shown here due to size, however there are few pieces we want to show you here.

First, here is how we need to configure our MapReduce job in the driver class.

1 *http://lucene.apache.org/solr/4_10_3/*

Example 2-5. Index HBase Avro table to SOLR using MapReduce driver

```
scan.setCaching(500);          ❶
scan.setCacheBlocks(false);    ❷

TableMapReduceUtil.initTableMapperJob( ❸
  options.inputTable,              // Input HBase table name
  scan,                            // Scan instance to control what to index
  HBaseAvroToSOLRMapper.class,     // Mapper to parse cells content.
  Text.class,                      // Mapper output key
  SolrInputDocumentWritable.class, // Mapper output value
  job);

FileOutputFormat.setOutputPath(job, outputReduceDir);

job.setJobName(getClass().getName() + "/" + Utils.getShortClassName(HBaseAvroToSOLRMapper.class));
job.setReducerClass(SolrReducer.class); ❹
job.setPartitionerClass(SolrCloudPartitioner.class); ❺
job.getConfiguration().set(SolrCloudPartitioner.ZKHOST, options.zkHost);
job.getConfiguration().set(SolrCloudPartitioner.COLLECTION, options.collection);
job.getConfiguration().setInt(SolrCloudPartitioner.SHARDS, options.shards);

job.setOutputFormatClass(SolrOutputFormat.class);
SolrOutputFormat.setupSolrHomeCache(options.solrHomeDir, job);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(SolrInputDocumentWritable.class);
job.setSpeculativeExecution(false);
```

❶ By default, scans cache only one row at a time. To reduce RPC calls and improve throughput we want to increase the size of the cache.

❷ Since we are going to scan the entire table once and only once, caching the blocks is not required and will just put pressure on the RegionServers' blockcache. It is always recommended to disable the blockcache when running a MapReduce job over a table.

❸ Again we are using HBase utility classes to configure required MapReduce input formats and output formats as well as the required mapper.

❹ Use the default Apache SOLR reducer class.

❺ Also use the default apache SOLR partitioner class.

Everything on the class should be pretty straight forward to understand.

Now let's have a look at the mapper. Goal of the mapper is to read the content from HBase and translate it for SOLR. We have already done a class to create an Avro

object from an HBase cell. We are going to reuse the same code here as this is exactly what we want to achieve. We want to read each and every cell, convert it back to an Avro object and provide to SOLR the data we want to index. The code for that is the following:

Example 2-6. Index HBase Avro table to SOLR using MapReduce mapper

```
event = util.cellToEvent(cell, event); ❶

inputDocument.clear(); ❷
inputDocument.addField("id", UUID.randomUUID().toString()); ❸
inputDocument.addField("rowkey", row.get());
inputDocument.addField("eventId", event.getEventId().toString());
inputDocument.addField("docType", event.getDocType().toString());
inputDocument.addField("partName", event.getPartName().toString());
inputDocument.addField("partNumber", event.getPartNumber().toString());
inputDocument.addField("version", event.getVersion());
inputDocument.addField("payload", event.getPayload().toString());

context.write(new Text(cell.getRowArray()),
                  new SolrInputDocumentWritable(inputDocument)); ❹
```

❶ Transform the received cell into an Avro object re-using the event instance to avoid creation of new objects.

❷ Here again we want to re-use existing objects as much as possible and therefore will simply re-initialize and re-use the SOLR input document.

❸ Assign to the SOLR input document all the fields we want to index or store from the Avro event object.

❹ Write the SOLR document to the context for indexing.

If you want to run the indexing from the command line, you will have to use the following command:

```
hbase -classpath ~/ahae/target/ahae.jar:`hbase classpath` \
com.architecting.ch09.MapReduceIndexerTool
```

You can also execute it from Eclipse without any specific parameter.

Data retrieval

At this point, we have generated test data, transformed it into Avro format stored into HFiles, loaded it into a table and indexed it into SOLR. The only remaining piece is to make sure we can query SOLR to find what we are looking for and then retrieve the

related information from HBase. The HBase retrieval part is the same as what we have already seen above. For SOLR, you can query SOLR using the following code:

Example 2-7. Retrieve Avro data from HBase based on SOLR.

```
CloudSolrServer solr = new CloudSolrServer("localhost:2181/solr"); ❶
solr.setDefaultCollection("Ch09-Collection"); ❷
solr.connect();

ModifiableSolrParams params = new ModifiableSolrParams();
params.set("qt", "/select");
params.set("q", "docType:ALERT AND partName:NE-555"); ❸

QueryResponse response = solr.query(params); ❹
SolrDocumentList docs = response.getResults();

LOG.info("Found " + docs.getNumFound() + " matching documents.");
if (docs.getNumFound() == 0) return;
byte[] firstRowKey = (byte[]) docs.get(0).getFieldValue("rowkey");
LOG.info("First document rowkey is " + Bytes.toStringBinary(firstRowKey));

// Retrieve and print the first 10 columns of the first returned document
Configuration config = HBaseConfiguration.create();
try (Connection connection = ConnectionFactory.createConnection(config);
    Admin admin = connection.getAdmin();
    Table sensorsTable = connection.getTable(sensorsTableName)) {
  Get get = new Get(firstRowKey); ❺

  Result result = sensorsTable.get(get);
  Event event = null;
  if (result != null && !result.isEmpty()) { ❻
    for (int index = 0; index < 10; index++) { // Print first 10 columns
      if (!result.advance())
        break; // The is no more column and we have not reached 10.
      event = new Util().cellToEvent(result.current(), event);
      LOG.info("Retrieved AVRO content: " + event.toString());
    }
  } else {
    LOG.error("Impossible to find requested cell");
  }
}
```

❶ Connect to your SOLR cluster. Adjust this if you are not running SOLR on the same cluster as HBase.

❷ Define the SOLR connection you want to use.

❸ Configure the request you want SOLR to execute. Here we ask it all the ALERT documents for the NE-555 part.

4. Execute the SOLR request and retrieve the response from the server.
5. Call HBase given the row key of the first document sent back by SOLR.
6. Iterate over the columns for the given key and display the first ten Avro objects retrieved from those columns.

Going further

If you want to go further on the examples from this chapter, here are some things you can try based on what we have discussed in this chapter.

Bigger input file

To make sure examples run pretty fast, the dataset we worked with was pretty small. What about trying with a bigger one? Depending on the available disk space you have and the performance of your environment, try to create a significantly bigger input file and verify it's processed the exact same way.

One region table

Since it's a good practice to avoid hotspotting, we have created a table with multiple regions split based on the key we used. Therefore that the different MapReduce jobs have generated multiple files, one per region. What if we create a table with a single region instead? Try to modify the create table statement to have a single region and load more than 10GB of data into it. You should see the region splitting after the data is inserted, however, since we are using bulkload, you should still not see any hotspotting on this region. You can validate your tables splits and the content of each region by looking in HDFS has seen in "Bulk loading" on page 29

Impact on table parameters

We have created our table using the parameters which are good for our current use-case. We recommend modifying the various parameters and re-running the process to measure the impact.

Compression

Try to use different types of compression and compare. If you used Snappy, which is fast, try to configure LZ4, which is slower but compress better, and compare the overall time it takes to process everything vs. the size of your files.

Block encoding

Because of format of the key we store into this table, we configured it to use the FAST_DIFF data block encoding. Refer to later chapter ??? and try to modify your table to use different data block encodings. Here again, look at the performance and the overall data size at the end.

Bloom Filter

When doing reads, bloom filters are useful to skip HBase store files where we can affirm the key we are looking for is not present. However, here we knew that the data we are looking for will always be present in the file, so we disabled the bloom filters. Create a list of tens of keys and columns that you know are present in the table and mesure how long it takes to read them all. Now activate the bloom filter on your table, major compact it to get them written and test again. You should see that for this specific use-case, Bloom filters are not improving the performances.

It is almost always good to have bloom filters activated. We disabled them here because this use case is very specific. If you are not sure, just keep them on.