Big HDF FAQs

Everything that HDF Users have Always Wanted to Know about Hadoop . . . But Were Ashamed to Ask

Gerd Heber, The HDF Group <gheber@hdfgroup.org>
Mike Folk, The HDF Group <mfolk@hdfgroup.org>
Quincey A. Koziol, The HDF Group <koziol@hdfgroup.org>

Revision History
Revision 1.0 24 March 2014

Abstract

The purpose of this document is to offer some guidance on the use of HDF5 with Hadoop. If you are puzzled by the relationship between HDF and Hadoop, welcome to the club! We have condensed the acme of confusion into a few questions for which one might expect straightforward answers. In the discussion, we present our answers at three levels of intimidation: a one phrase answer (in the subtitle), a multi-paragraph elaboration, and a set of appendices. If you are not satisfied after the third level, please, send us your answer! And don't hesitate to send us your questions either.

We would like to thank all advocates, developers, maintainers, and supporters for their contributions to open-source software including HDF5 and Apache Hadoop.

Introduction

Mo' data, mo' of the same. Mo' data, mo' problems! (Jay-Zed)

Before the term 'data' took on the meaning of 'symbols in a (storage) medium', it was a rare and unlikely topic of conversation. [Gitelman2013] The clever decision by marketing departments to add the attribute 'Big' has turned it into a hybrid of a spectre and a circus. To be sure, this was triggered by real and exciting new developments, but putting too much emphasis on a single or a handful of attributes of the data concerned is a distorted view and in stark contrast with the diversity of the technologies that have emerged and continue to appear on the scene almost weekly. Making technology choices and having understood the implications beforehand(!) becomes even harder in such a period of upheaval. [McCallum2012]

Existing technologies have to address real and *perceived* challenges. Ironically, the perceived challenges can be more threatening than the actual ones. It's important to separate the two, to expose the perception as a mis-perception, and to avoid falling into an apples vs. oranges comparison.

In this discussion, we would like to highlight opportunities with HDF5 on the de-facto standard Big Data platform, Apache Hadoop. If the question in the back of your mind is, *"Should I use Hadoop or HDF5?"*, then we have good and bad news for you. Let's start with the bad news: You may not have realized that, if 'or' is meant in the exclusive sense, you are asking an ill-posed question such as, *"Should my diet include minerals or vitamins (but not both)?"* Hadoop and HDF5 are very different technologies for very different purposes, and it is difficult to imagine a situation when this question would be meaningful. Think of Hadoop as the single cup coffee maker and HDF5 as one form factor of coffee packs, — which brings us to the good news: Together they make a flavorful cup of coffee.

Rather than making this a technical monologue, we have chosen the more engaging FAQ format. This is our list of questions:

1. What is HDF5?

2. What is Hadoop?
3. HDF5? HDFS? What is the difference?
4. How can I get HDF5 data in-n-out of Hadoop?
5. Can I run MapReduce jobs against HDF5 files?

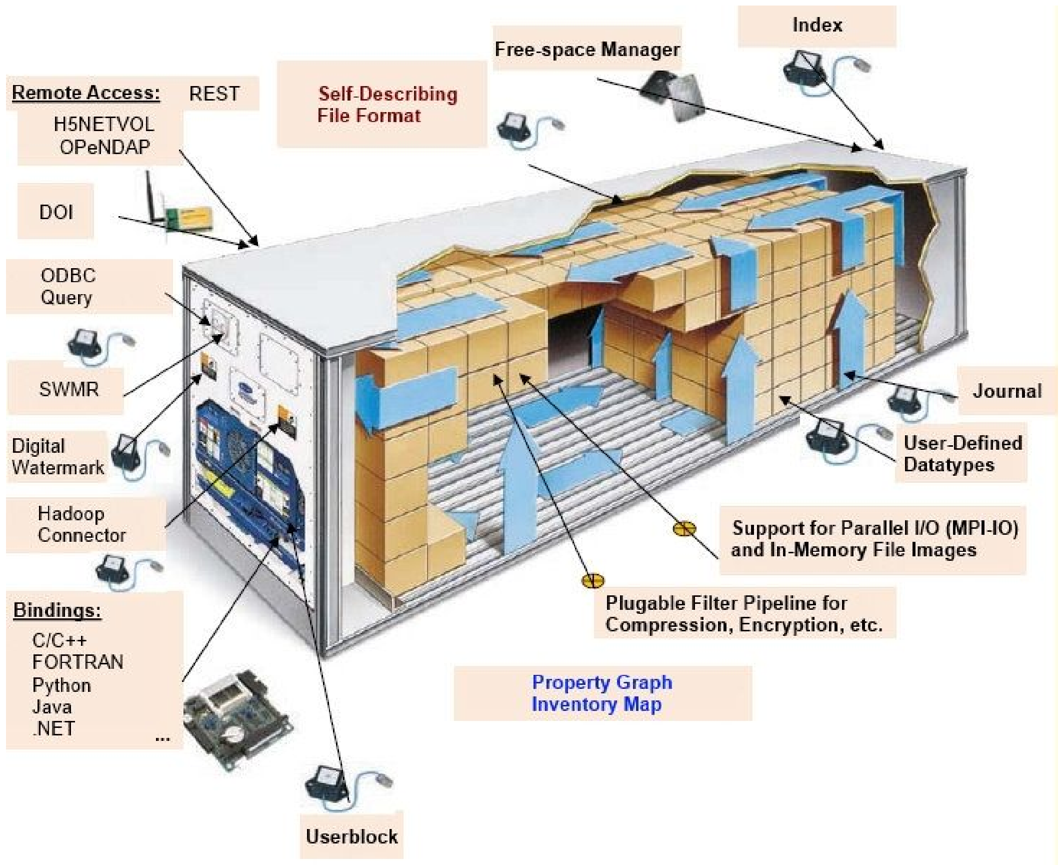
We have tried to strike a balance between making the answers self-contained and not to be overly repetitive. Jump right in, but be prepared to back-track and refer to the other answers.

Finally, this document is *not* a product announcement or commitment on our part. The ideas and concepts presented are at different levels of maturity. Some have been explored or prototyped at The HDF Group or by others, but none of them has seen the rigorous development and testing required to deserve that title.

What is HDF5?

HDF5 is a smart data container.

Figure 1. HDF5 Container Cut-Away View[1]



If Hadoop is an elephant, so is HDF5, — at least metaphorically speaking. The variety of descriptions uncovered by a simple Google search resembles the story from the blind men and the elephant. (No disrespect to our users!) HDF5 is more than just a file format, a parser/API library, or a data model. It's all of the above and more. If we had to draw a picture of an "HDF5 thing" it might look something like Figure 1, “HDF5 Container Cut-Away View”. Not quite an elephant, but close! We like to refer to it as an

[1]Image source: http://www.mvac-ohio.org/educators/PowerPoints/Smart%20Container.jpg

HDF5 Smart Data Container, a standardized, highly-customizable data receptacle designed for portability and performance. Please refer to [H5FAQ] for the details on some of the smarts called out in the figure.

As any container, the HDF5 data container achieves portability through *encapsulation* or separation of its cargo from the environment. At the lowest level, this is implemented through an open, *self-describing* file format [H5FF] and a free and open-source parser library and API. [H5RM] What a user sees through an API or connector are high-level abstractions, and very little can be inferred about the internals of the container, for example, how (dis-)similar the internal representation of a table is to a flat record stream.

An HDF5 data container combines user data and metadata in the form of a collection of array variables[2] in a property graph[3]-like arrangement. HDF5 data containers come in all sizes and one can find them used in applications running on mobile devices, desktops, servers, and supercomputers [Byna2013], and, given enough space and logistics, the containers can be moved freely between the different environments.

The containerized nature of HDF5 is not exactly a deal breaker for its use with Hadoop, but it makes the engagement less straightforward than we would like. Speaking generally, at its heart the story about the relationship between HDF5 and Hadoop is a story about the tension between encapsulation and visibility, between opacity and transparency.

What is Hadoop?

Hadoop = HDFS + MapReduce

The short answer is about as accurate as its better known ancestor:

> Communism is Soviet power plus the electrification of the whole country.
>
> —Vladimir Ilyich Lenin

According to the Apache Hadoop home page [Hadoop] the project modules include:

- **Hadoop Common:** The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™):** A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN:** A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce:** A YARN-based system for parallel processing of large data sets.

There are plenty more Hadoop-related projects at Apache, some of which we'll revisit later. For this discussion we content ourselves with HDFS and MapReduce.

If all the acronyms don't mean much to you (yet), try to remember that Hadoop is a platform that *takes the computation to the data*. This means that *data location* is a major factor and it has implications for how well an algorithm lends itself to be implemented in MapReduce.

HDFS

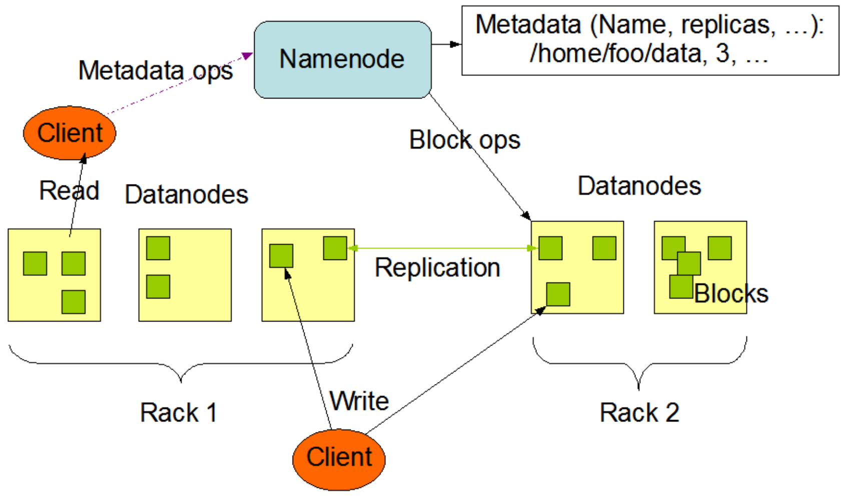
HDFS [Borthakur2013], [Shvachko2010a], [Shvachko2010b] is a distributed file system designed to store very large data sets reliably, to stream those data sets at high bandwidth to user applications such as MapReduce, and to run on commodity hardware.

An HDFS file system consists of a namenode (metadata server) and several datanodes. A file is represented as a sequence of equi-sized blocks (except maybe the last one), which are replicated among the datanodes.

[2] An array variable is a variable whose value is a multi-dimensional, array of elements of the same type.
[3]A property graph is a multi-graph whose nodes and edges may have zero or more (named) properties. [RobinsonWebberEifrem2013]

The block size and replication factor can be specified on a per file basis. Typical default values are a 64 MB block size and a replication factor of 3. The namenode tracks the block distribution and ensures proper replication to guard against datanode failure. Unlike many other file systems, HDFS exposes the block *location* to clients and there is a concept of proximity or distance of a block replica to the client, which typically will try to access the closest replica. This has several ramifications for HDF5 containers, see the section called “HDF5? HDFS? What is the difference?” and the section called “Can I run MapReduce jobs against HDF5 files?”.

Figure 2. HDFS Architecture [Borthakur2013]

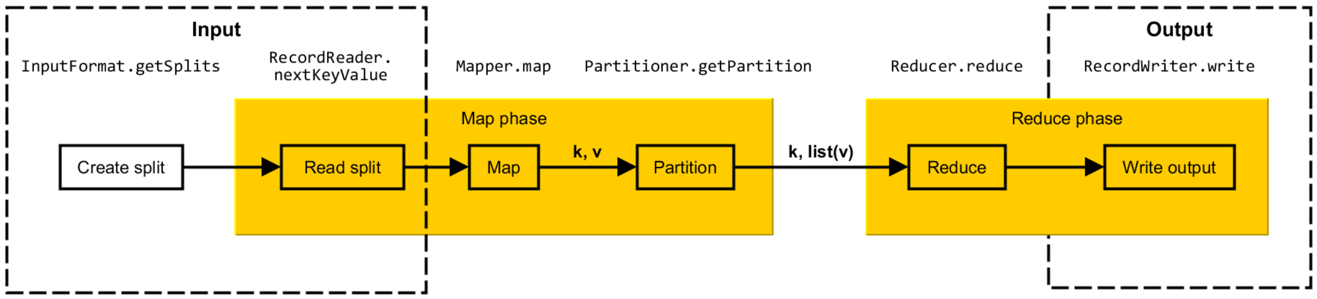


Currently, HDFS is an append-only file system and, because of different design goals, not fully POSIX compliant. There are several efforts to use MapReduce & Co. with other file systems, but this is a topic beyond the scope of this discussion. Keep in mind that, unlike these other file systems, HDFS was designed for Hadoop, and because of its focused design there is a much better chance to understand why something does or doesn't perform.

MapReduce

The MapReduce framework implements a data processing paradigm which is based on a two phase (`map` and `reduce`) processing model popular in many functional languages. In a MapReduce job, a stream of data elements in the form of key-value pairs is partitioned into *splits*, which are processed in parallel by independent *map tasks*, which produce a stream of key-value pairs. This intermediate stream is then re-partitioned and assigned to a set of independent *reduce tasks* that combine the intermediate results into a final key-value pair output stream. [Holmes2012], [MinerShook2012]

Figure 3. MapReduce Overview. [Holmes2012]



The specifics of the source data stream are isolated and neutralized in a class called `InputFormat<K,V>`, which provides a partitioning of the input stream (`List<InputSplit>`) and a record reader (`RecordReader<K,V>`). A similar arrangement takes care of the output stream (`OutputFormat<K,V>`, `RecordWriter<K,V>`). For example, there are different specializations of `InputFormat` and `OutputFormat` for common file formats and several SQL and NoSQL data stores.

There's a good chance that your next question is covered in the section called "Can I run MapReduce jobs against HDF5 files?".

HDF5? HDFS? What is the difference?

"The pen is in the box and the box is in the pen." [Bar-Hillel1960]

Ok, we are just joking. Since we've already covered HDF5 and HDFS separately, it would have been a little dull to reiterate that HDF5 is a smart data container and HDFS is a file system. The word play is for all who harbor sympathy for the idea of HDF5 as "a file system in a file". HDF5 is *not* a file system. Why?

- HDF5 containers are portable; file systems are not portable.
- File systems store uninterpreted streams of bytes (aka. files); HDF5 containers store (among other things) HDF5 datasets, which are well defined HDF5 objects.
- HDF5 containers have a portable user metadata layer, file systems don't.

In order to get a better idea of how different HDFS is from some of the file systems you might be familiar with, let's remind ourselves what happens when one copies a file (e.g., an HDF5 file) from a local file system into HDFS! With the Hadoop shell, copying a file is as simple as:

```
hadoop fs -put localfile.h5 hdfs://nn.example.com/hadoop/hadoopfile
```

Optionally, we could prescribe the block size for the file, e.g., 128 MB:

```
hadoop fs -D dfs.blocksize=134217728 -put localfile.h5 hdfs://nn.example.com/hadoop/hadoopfile
```

Let's assume that the source file size is 128 MB even. With a default HDFS block size of 64 MB, the file would be split into two blocks and sent to the appropriate datanodes. (There would be only one block, if we had specified 128 MB as the block size for this file.) There's plenty of activity behind the scenes: The namenode would create a metadata record for the file and decide on a distribution of the blocks among datanodes, and two replica of each block would be created. The distribution is done in a pipelined fashion where datanodes receive and send block data simultaneously, etc. [Shvachko2010a] HDFS is an ingenious piece of engineering!

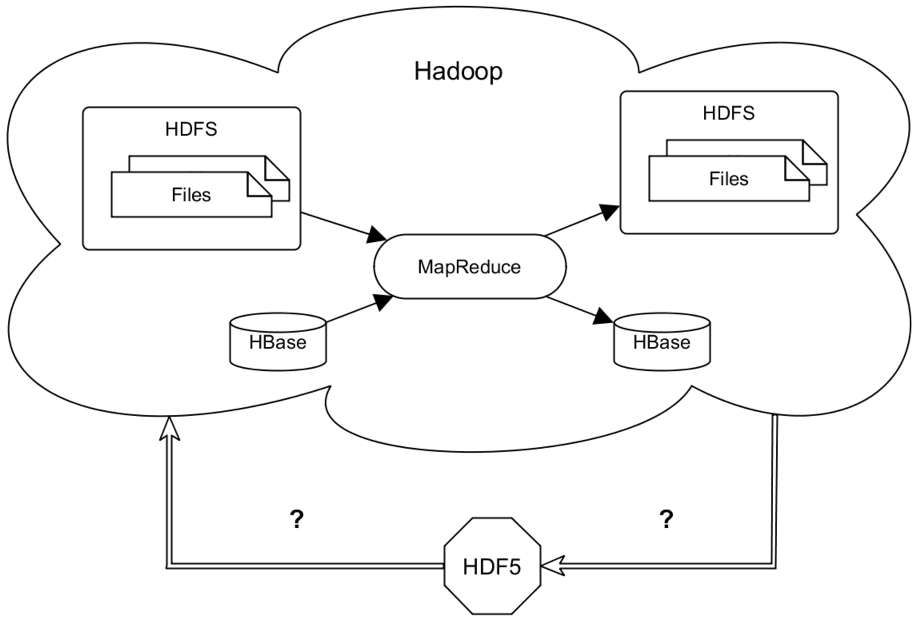
Unfortunately, our beautifully crafted HDF5 container (see Figure 1, "HDF5 Container Cut-Away View") doesn't take that kind of treatment very well: Breaking the container into blocks is a bit like taking an axe and chopping it to pieces, severing blindly the content and the smart wiring in the process. The result is a mess, because there's no alignment or correlation between HDFS block boundaries and the internal HDF5 cargo layout or container support structure. (Think of the block boundaries as a set of parallel, equidistant planes cutting through the container shown in Figure 1, "HDF5 Container Cut-Away View".) We could repair the damage via `InputFormat<K,V>` and "put the pieces back together", if we were able to easily determine the physical location of a dataset element in an HDF5 container and, hence, which HDFS block it falls into. Currently, we cannot do this in a general and efficient manner. This is where location transparency (HDFS) and location opacity (HDF5), where visibility and encapsulation *collide*. Call it quits? Not so fast! Read on in the section called "Can I run MapReduce jobs against HDF5 files?"!

How can I get HDF5 data in-n-out of Hadoop?

Your imagination is the limit.

The situation is depicted in Figure 4, “Hadoop In-n-out”. We have HDF5 containers sitting in a file system outside Hadoop and would like to bring the data (not necessarily the whole container) into Hadoop for analysis, or we have data products sitting in Hadoop that we would like to package as HDF5 data containers and ship them elsewhere.

Figure 4. Hadoop In-n-out



Unfortunately, there is no single, simple, and general answer. Several factors need to be taken into account, but here we'll focus on object size. There are known solutions of this problem for small, medium, and large objects [Holmes2012], and sometimes a creative combination is what gives the best results.

Small HDF5 Containers

How small? By 'small' we mean "small compared to the size of an HDFS block." By this definition a 2 or 10 MB file is a small object. A less precise definition might be, "I'm having performance problems with this system that allegedly was designed to handle massive data streams, but it can't even handle my small ones." The same way that nature abhors a vacuum (Aristotle), HDFS abhors (large numbers of) small objects. [Shvachko2010b] They put a lot of unnecessary pressure on the namenode (see Figure 2, “HDFS Architecture [Borthakur2013]”), which one should avoid.

There are at least two options to deal with small HDF5 containers:

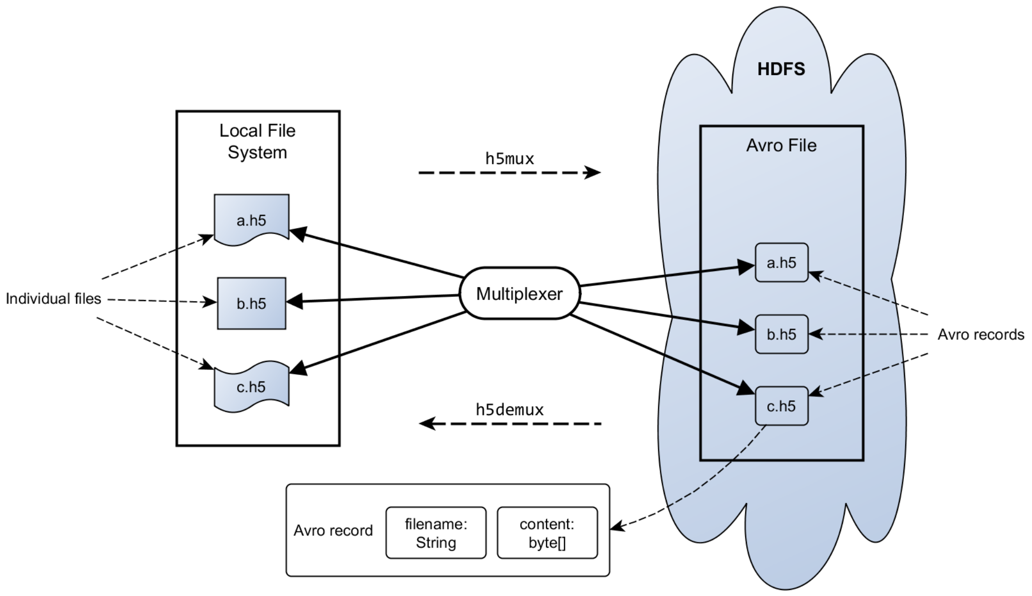
1. After the fashion of `tar` archives, pack many small HDF5 containers into one or more large Avro files. (See Appendix B, *Apache Avro*.)

2. Save small HDF5 containers as cell values in HBase tables. (See Appendix C, *Apache HBase*.)

The idea of an HDFS/Avro Multiplexer [Holmes2012] is illustrated in Figure 5, “HDFS/Avro Multiplexer”.

Both Avro files and HBase tables are excellent MapReduce data sources/targets (see the section called “Can I run MapReduce jobs against HDF5 files?”) and the small HDF5 containers can be processed locally (or in memory) via the Java native interface for HDF5 [JHI5].

Figure 5. HDFS/Avro Multiplexer



Medium-sized HDF5 Containers

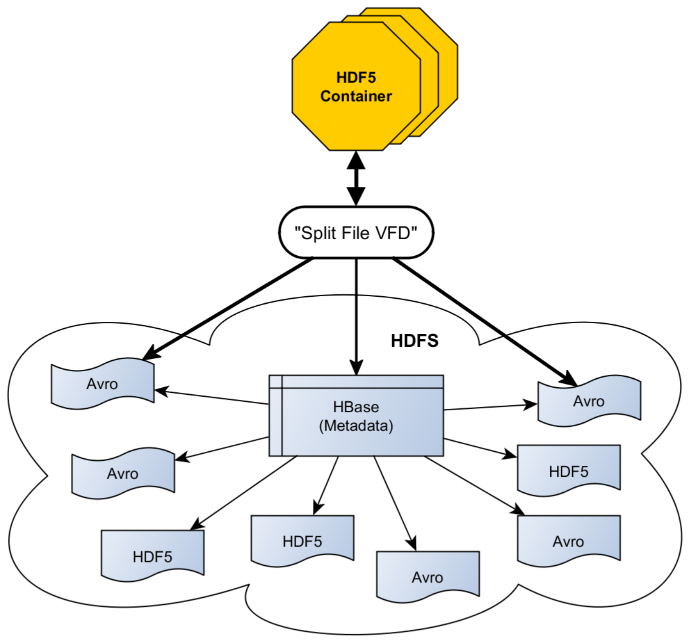
A medium-sized HDF5 container spans several HDFS blocks and `hadoop fs -put` (copy) is the way to go. In the section called “Can I run MapReduce jobs against HDF5 files?”, we will talk about how to access them from MapReduce. We treat medium-sized HDF5 containers as *non-splittable* input and process them in memory or off the local file system. (See Appendix F, *An HDF5 Container Input Format*.)

Large HDF5 Containers

In the section called “Can I run MapReduce jobs against HDF5 files?”, we will discuss the issues around in situ MapReduce processing of large HDF5 containers. In this section, we describe an ETL-like approach which requires a certain amount of up-front processing. It emulates the idea of an HDF5 split file VFD [H5VFL] on a Hadoop scale. The basic idea behind the HDF5 split file VFD is to separate data and metadata, and to store them in separate physical files.

The idea of a Big HDF split file VFD is depicted in Figure 6, “Big HDF Split File VFD”. It "shreds" the content of an HDF5 container into of a combination of HBase records (metadata) and Avro files for the heavy-weight data. Unlike the blunt HDFS blocking approach, this process makes the data context visible (in HBase) and simplifies exporting and packaging data products.

Figure 6. Big HDF Split File VFD



Can I run MapReduce jobs against HDF5 files?

Yes, but proceed with caution.

The question should probably be formulated more precisely like this: *Can I run MapReduce jobs against HDF5 files and get good performance?* We hope the material in this document gave you some of the necessary background to make that determination. Let's summarize the most important points!

Is MapReduce the right tool for the job?

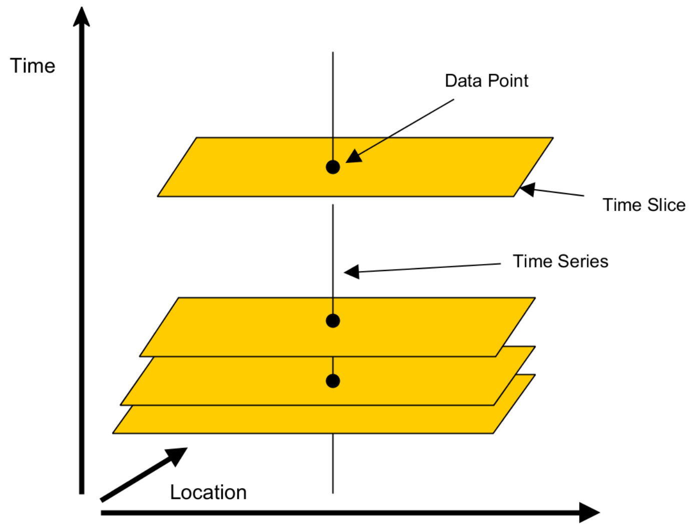
That is a question you will have to answer. You might be better off creating your own customized version of MapReduce. [Juric2011], [Wang2013] MapReduce is good for many tasks that include summarizing, filtering, sampling, reorganizing, and joining large streams of data. [MinerShook2012] But it is not a general solution to all problems around large datasets. Despite its high-level of abstraction (streams of key/value pairs), MapReduce is a low-level component compared to Hive [CaprioloWamplerRutherglen2012] and Pig [Gates2011], or more problem-oriented environments such as Mahout [OwenEtAl2011] and R [R].

In Figure 7, “Big HDF Use Case”, a task is shown for which Hadoop might be a good solution. We have a large number of time slices or frames (stored in one or more HDF5 containers), which we would like to convert into a bundle of location- or pixel-based time series to be further analyzed. The implementation details will vary depending on the frame count and number; depending on the analysis, there might be several mappers, or even multiple MapReduce jobs. Be that as it may, Hadoop is a good candidate for this kind of use case.

Size matters

Our discussion in the section called “How can I get HDF5 data in-n-out of Hadoop?” suggests that there's no one-size-fits-all solution. MapReduce works best when it can parallelize (split) the reading of input data. Treating medium-sized HDF5 containers as non-splittable is a band aid and, unless there are many containers to be processed, a bottleneck. Avro file are splittable, which helps the Big HDF split file VFD, and the multiplexer trick takes care of small HDF5 files.

Figure 7. Big HDF Use Case



What about in situ processing?

Currently, HDF5 containers cannot be processed in situ by MapReduce.[4] The best technical account is Joe Buck's SC'11 paper [Buck2011]. The paper is about SciHadoop, an implementation of array-based querying processing in Hadoop. Although SciHadoop only supports NetCDF-3 (a distant cousin of HDF5), the line of argument would be nearly identical for HDF5. SciHadoop is implemented essentially by extending the `FileInputFormat<K,V>` class. At the heart of the implementation is a function that translates the logical coordinates of a data element into a physical file offset. Unfortunately, the greater complexity of the file format (and the library) and the maintenance cost for such a "backdoor" make this a much less attractive proposition for HDF5. Ultimately, this goes back to the tension between encapsulation and visibility. An HDF5 file is a container, which effectively eliminates the visibility of its cargo.

Does that mean in situ processing of large HDF5 containers won't happen? No, but it requires a few good use cases and prudent engineering to produce a maintainable solution.[5] Please send us your use case, if you'd like to be kept in the loop!

Besides "native" HDF5 containers, there are also other options on the horizon, for example an HDFS VOL plug-in. (See Appendix E, *The HDF5 Virtual Object Layer (VOL)*.)

Summary

Since we have confined most of the technical noise to the appendices, we hope that the main message of this document comes through loud and clear: *Hadoop is an interesting platform for processing and*

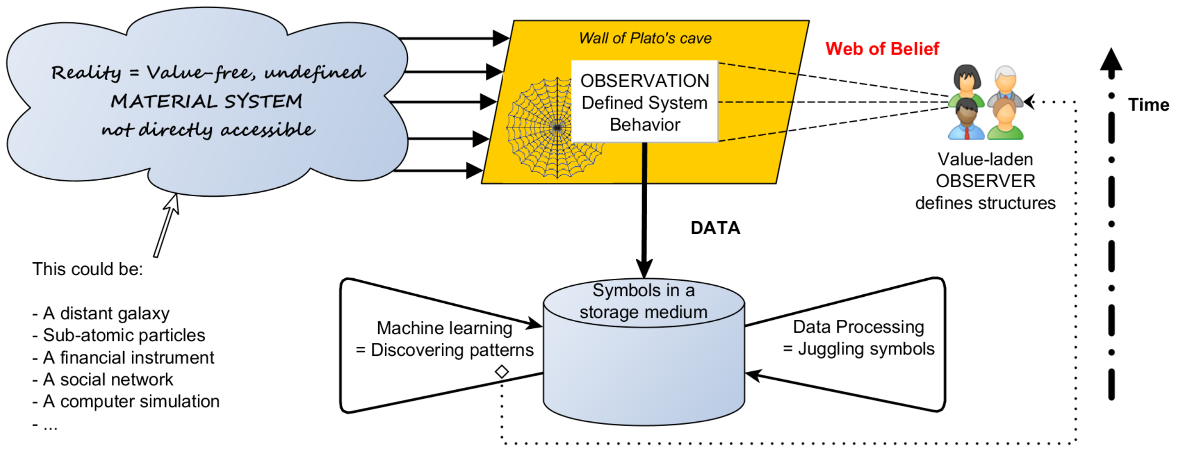
[4] With the exception of the small and medium objects as described earlier, or datasets using external storage.
[5] If you are desperate and maintainability is not a concern, you can hack your own splittable version of `InputFormat<K,V>`. The case of an HDF5 dataset with contiguous layout is trivial: Get the object address in the file (`H5Oget_info`), parse the header messages ([H5FF]), and get the address of the first data element. Together with the knowledge of the element type, the dataset extent, and `hdfsGetHosts` it's easy to create `InputSplits`. For chunked datasets, one has to work harder to get the chunk addresses, but the main complication comes from compression or other filters that may have been applied to the chunks. We are not saying, *"Don't try this at home!"*, but we want you to understand that you should consider applying for a job at The HDF Group, if you've come this far.

analyzing data stored in HDF5 containers. Technical challenges remain, but plenty can be done with both technologies as they are today, and we have included several suggestions on how you might get started. Help us to help the community overcome those challenges by joining the discussion and by sharing experiences, ideas, and use cases!

A. Data

Only superficially does the generation, transmission, storage, and consumption of data resemble a resource such as electric power. Some data might be treated as a commodity, but *data is not fungible*. (An ounce of gold is equivalent to another ounce of gold. A megabyte of data is not equivalent to another megabyte of data.) Not enough that data is not fungible, it entails an interpretative base and is bound to be viewed differently by simultaneous observers and at different times. The notion of 'raw data' sounds compelling (crude oil!), but is an oxymoron. [Gitelman2013] A more realistic scenario is depicted in Figure A.1, “Data — Fragments of a Theory of the Real World”. [Mealy1967], [AhlAllen1996]

Figure A.1. Data — Fragments of a Theory of the Real World



And the difficulties don't stop there. For example, the representation of the data in a medium can be the source of additional complexity:

> When data is locked up in the wrong representation, however, the value it holds can be hidden away by accidental complexity...
>
> A big part of our jobs as data scientists and engineers is to manage complexity. Having multiple data models to work with will allow you to use the right tool for the job at hand.
>
> —(Bobby Norton in [McCallum2012])

See Appendix C, *Apache HBase* and Appendix D, *HDF5 and Graph Databases* for alternative representations of HDF5.

B. Apache Avro

Avro is a hidden gem in the Hadoop stack. It's not widely covered in the literature on Hadoop, but the work proves the craftsman (Doug Cutting). "Apache Avro is a data serialization system. Avro provides:

- Rich data structures.
- A compact, fast, binary data format.
- A container file to store persistent data.
- Remote procedure call (RPC).

- Simple integration with dynamic languages." [Avro]

In Avro, data are serialized with a so called *Avro schema*, which is stored with the data and makes the Avro format *self-describing*. Avro schemata are represented as JSON objects. For example, the following would be an Avro schema for a node record in 3D Cartesian space:

```
{
  "type": "record",
  "name": "Node",
  "namespace": "org.hdfgroup.www",
  "fields": [
    { "name": "node_id", "type": "long" },
    { "name": "x",       "type": "double" },
    { "name": "y",       "type": "double" },
    { "name": "z",       "type": "double" }
  ]
}
```

This is a rather trivial example: The Avro type system includes more sophisticated data structures such as arrays (1D), unions, and maps. Nevertheless, the Avro format is *record-oriented* and the Avro framework lets one write those records compactly and efficiently. Among several Big Data serialization frameworks, Avro is the front runner and works natively with MapReduce (Cf. Table 3.1, p. 100 in [Holmes2012] for a nice comparison.), Pig [Gates2011], and Hive [CaprioloWamplerRutherglen2012].

The remainder of this section consists in going down the list of HDF5 datatype classes and making sure all can be mapped to appropriate Avro record types. *Remember that in the process certain metadata will fall by the wayside unless they are captured outside Avro.*

Avro Schema and HDF5 Datatypes

We have to convince ourselves that all HDF5 datatypes have a home in the Avro type system. Let's go down the list!

HDF5 Array

The values of an HDF5 array datatype are dense, rectilinear, multi-dimensional (up to rank 32) arrays of elements of a given HDF5 datatype. Avro has an array type with the following schema:

```
{ "type": "array", "items": <item type> }
```

Avro types can be nested the same way that HDF5 datatypes can be nested and to emulate multi-dimensional arrays we have at least two options:

1. Flatten and represent them as one-dimensional arrays

2. Represent them as nested one-dimensional arrays

In both cases, one needs to keep track of the metadata, i.e., the rank (option no. 2 only) and extent in the different dimensions.

Lets look at an example! Take the following HDF5 compound datatype:

```
DATATYPE "type1" H5T_COMPOUND {
  H5T_ARRAY { [4] H5T_STD_I32BE } "a";
  H5T_ARRAY { [5][6] H5T_IEEE_F32BE } "b";
}
```

It has two fields, `a` and `b`, where `a` is a four vector of 32-bit integers and `b` is a five by six matrix of single precision floating-point numbers. An Avro rendering would be the following record type:

```
{ "type": "record",
  "name": "type1",
  "fields": [
    { "name": "a",
      "type": { "type": "array", "items": "int" }
    },
    { "name": "b",
      "type": {
        "type": "array",
        "items": {
          "type": "array", "items": "float" }
        }
      }
    }
  ]
}
```

The flattened version would be just:

```
{ "type": "record",
  "name": "type1",
  "fields": [
    { "name": "a",
      "type": { "type": "array", "items": "int" }
    },
    { "name": "b",
      "type": { "type": "array", "items": "float" }
    }
  ]
}
```

HDF5 Bitfield

Avro does not have a direct counterpart to HDF5 bitfields, which can be thought of as fixed size sequences of flags. The most intuitive, but not the most efficient representation is an Avro array of Booleans.

```
{ "type": "array", "items": "bool" }
```

Avro represents Booleans as single bytes. This is not a problem for short bitfields. For larger bitfields (uncommon?), a representation as Avro `bytes` is more efficient. The HDF5 bitfield metadata, such as size, must be stored elsewhere and this should also be a place to store the specifics of the mapping (`bool` or `bytes`).

HDF5 Compound

The Avro record type is the perfect match for an HDF5 compound datatype.

```
{ "type": "record",
  "name": <name>,
  "namespace": <namespace>,
  "fields": [
    { "name": <field name>, "type": <field type> },
    { "name": <field name>, "type": <field type> },
    ...
  ]
}
```

HDF5 Enumeration

Avro has an enumeration type which is only marginally less flexible than HDF5 enumeration datatypes.

```
{ "type": "enum",
  "name": <name>,
  "namespace": <namespace>,
  "symbols": [ <symbol>, <symbol>, ... ]
}
```

No duplicates among the symbols are permitted. In an HDF5 enumeration datatype, an arbitrary value of an HDF5 integer datatype can be assigned to a symbol, which cannot be achieved directly with Avro `enum`.

HDF5 Floating-Point Number

Avro supports single and double precision IEEE floating point numbers, `float` and `double`. The HDF5 datatype system supports user-defined floating point datatypes of arbitrary length and precision. However, this is a rarely used feature.

HDF5 Integer

The Avro type system supports 4-byte and 8-byte signed integers (`int` and `long`). The HDF5 datatype system supports user-defined integer datatypes, signed and unsigned, of arbitrary length and precision. Apart from the standard 8-, 16-, 32-, 64-bit integer types, this feature is used occasionally. Use care when converting your integers!

HDF5 Opaque

The values of an HDF5 opaque datatype are fixed-length byte sequences. There are two pieces of metadata associated with such a type: its length and an optional tag (string). The Avro `bytes` type is a good match for an HDF5 opaque datatype. Remember to keep the length and tag handy someplace else!

HDF5 Reference

HDF5 reference datatypes come in two flavors, HDF5 object references and HDF5 dataset region references.

HDF5 Object Reference

Object references are “pointers” to HDF5 objects. Unlike links, which have a name and are anchored in HDF5 groups, object references are (anonymous) values that happen to be *addresses* of HDF5 objects (groups, datatsets, datatypes). We haven't spoken about how to maintain data context including the HDF5 'web of belief'. In a moment (see Appendix C, *Apache HBase*), we'll talk about how to achieve that using HBase. In this representation, the natural choice of address is the name of the appropriate HBase table and the object’s row key in that table.

```
{ "type": "record",
  "name": "H5T_STD_REF_OBJ",
  "namespace": "org.hdfgroup.hdf5",
  "fields": [
    { "name": "key", "type": "bytes" },
    { "name": "table",
      "type": {
        "type": "enum",
        "name": "HBaseObjectTableNames",
        "namespace": "org.hdfgroup.hdf5",
        "symbols":
          [ "groups", "datasets", "datatypes" ]
      }
    }
  ]
}
```

The `key` field of this record type holds the 128-bit row key of the object and the `table` field stores the table name.

HDF5 Dataset Region Reference

HDF5 dataset region references are masks, selections, or bookmarks on subsets of elements of HDF5 datasets. Logically, they consist of an object reference to the dataset in question and a description of the selection. The selections come in two flavors, point selections or hyperslab selections. A point selection is a simple list of data element positions (pixels, voxels, etc.). A hyperslab is a regular multidimensional pattern described by a start, stride, count, and block. In addition to individual patterns, certain set-theoretical combinations (union, intersection, etc.) are supported in HDF5 hyperslab selections. We represent dataset region references as Avro records with a dataset field holding an object reference to the dataset in question, and a selection field for the selection.

```
{ "type": "record",
  "name": "H5T_STD_DSET_REG",
  "namespace": "org.hdfgroup.hdf5",
  "fields": [
    { "name": "dataset",
      "type": "H5T_STD_REF_OBJ"
    },
    { "name": "selection",
      "type": [ "array", "string" ]
    }
  ]
}
```

Point selections are conveniently modeled as an Avro `array` of `long` tuples. Hyperslab selections can be conveniently expressed in JSON. Since Avro supports a `union` data type, it is easy to support both types of selections in the same field.

HDF5 Strings

Avro's `string` type can be used to store HDF5 string datatype values. The HDF5 datatype system distinguishes between fixed-length and variable-length string datatypes. The metadata such as length, padding, character encoding (ASCII, UTF-8), etc. need to be stored elsewhere.

HDF5 Variable-Length Sequence

An HDF5 variable-length sequence is pretty much the same as an Avro `array`.

```
{ "type": "array", "items": <item type> }
```

That's it. We have successfully dealt with all HDF5 datatypes known to mankind. Time to solve a real problem!

Tools

Dealing with large numbers of small (compared to the default HDFS block size) files is a familiar problem in HDFS. A standard trick [Holmes2012] is to package collections of small files as individual records in Avro "archives". In a MapReduce job, blocks of these records are parceled out to `map` tasks, which then process the underlying files in memory or off the local (non-HDFS) file system.

This idea, of course, can be applied to HDF5 files and is a convenient way to process small HDF5 files in bulk with MapReduce. We have implemented a small utility called `hmux`, which helps to prepare the HDF5 files by wrapping them into an Avro file.

We have developed another tool, `h5vax`, that let's one extract datasets (array variables) from an HDF5 file and dump them into an Avro file, which can be processed directly by MapReduce.

`h5mux` — HDF5/Avro Multiplexer

`h5mux` [--append] [-b,--avro-block-size *`size`*] [-h,--help] [-m,--max-file-size *`size`*] [-r,--recurse] [--skip-md5] [-z,--compression *`[deflate, none, snappy]`*] {*`source directory`*} {*`destination file`*}

The `h5mux` tool generates Avro files with the following schema:

```
{ "type": "record",
  "name": "h5mux",
  "namespace": "org.hdfgroup.hdf5",
  "fields": [
    { "name": "filename",
      "type": "string"
    },
    { "name": "content",
      "type": "bytes"
    }
  ]
}
```

The detailed meaning of the command line options is as follows:

--append	Use this optional parameter to append records to an existing Avro file with the same Avro schema. Without this flag `h5mux` generates an error if the destination file exists.
-b, --avro-block-size *`size`*	Use this optional parameter to set the Avro data block size. (The default Avro value is 16 KB.) If not specified, `h5mux` will attempt to determine a suitable value.
-h, --help	Displays a help message.
-m, --max-file-size *`size`*	Use this optional parameter to exclude files exceeding a maximum file size. `h5mux` will skip such files in the source directory. The default maximum file size is 16 MB.

-r,--recurse	Use this option to recursively visit subdirectories.
--skip-md5	Use this option to skip the generation of MD5 checksums.
-z, --compression *[deflate, none, snappy]*	Use this optional parameter to specify record compression.
source directory	Use this required parameter to specify the source directory containing HDF5 files. Non-HDF5 files will be skipped by `h5mux`.
destination file	Use this required parameter to specify the path to the destination Avro file. `h5mux` will generate an error if the destination file exists and `--append` was not specified or the Avro schema of the existing file is different from the default `h5mux` schema.

`h5demux` — HDF5/Avro Demultiplexer

`h5demux` restores the (HDF5) files from an Avro archive created by `h5mux`.

`h5demux` [-h,--help] {*source file*} {*destination directory*}

`h5vax` — HDF5 Variable Extractor

`h5vax` [--append] [-b,--avro-block-size *size*] [-d,--dataset *HDF5 path name*] [-h,--help] [-k,--key-value] [-z,--compression *[deflate, none, snappy]*] {*source file*} {*destination file*}

The `h5vax` tool extracts a dataset from an HDF5 file and writes it to an Avro file (in HDFS). The Avro schema is inferred from the dataset's HDF5 datatype. The detailed meaning of the command line options is as follows:

--append	Use this optional parameter to append records to an existing Avro file with the same Avro schema. Without this flag `h5vax` generates an error if the destination file exists.
-b, --avro-block-size *size*	Use this optional parameter to set the Avro data block size. (The default Avro value is 16 KB.) If not specified, `h5vax` will attempt to determine a suitable value.
-d, --dataset *HDF5 path name*	Use this optional parameter to set the dataset's HDF5 path name.
-h, --help	Displays a help message.
-k, --key-value	Use this optional parameter to force a key/value representation.
-z, --compression *[deflate, none, snappy]*	Use this optional parameter to specify record compression.
source file	Use this required parameter to specify the source HDF5 file.
destination file	Use this required parameter to specify the path to the destination Avro file. `h5vax` will generate an error if the destination file exists and `--append` was not specified or the Avro schema of the existing file is different from the inferred `h5vax` schema.

Without the `-k` option, we forget the array structure of the original dataset and turn it into a linear stream, and the `h5vax` tool generates Avro files with the following schema:

```
{ "type": "array", "type": <inferred value type> }
```

With the `-k` option the `h5vax` tool generates Avro files with the following schema:

```
{ "type": "record",
  "name": "h5vax",
  "namespace": "org.hdfgroup.hdf5",
  "fields": [
    { "name": "key",
      "type": "long"
    },
    { "name": "value",
      "type": <inferred value type>
    }
  ]
}
```

The `key` field is the *logical* position of the data element in the C-order flattened multi-dimensional array. For example, the logical position of the data element at position [4,7] in a square array of length 256 is 1031 (= 4*256+7).

The `h5vax` tool works with HDF5 datasets of arbitrary size. However, it strips the data of its context in the HDF5 'web of belief'. In order to maintain that context we need to look elsewhere on the Hadoop platform. We don't have to look far and wide.

C. Apache HBase

In the previous section, we discussed how to deal with small HDF5 files in Hadoop and how to use Avro to serialize arbitrary HDF5 datasets. The data context remains intact for small HDF5 files, but is potentially lost when we extract individual array variables. In this section, we explain how to use Apache HBase [HBase] to address this problem. This is by no means the only solution, but one that leverages another key element of the Hadoop platform.

The "storing/searching the internet" problem stood at the cradle of Google’s Bigtable design [Chang2006], which inspired HBase. [George2011], [DimidukKhurana2012] The problem discussed here is not dissimilar: If one thinks of HDF5 data containers as micro-webs or property graphs, then the digital asset management of large inventories of such containers is largely a problem of "storing/searching the HDF5 container logistical network". (The difference is that we are not dealing with clusters of HTML pages, but networks of array variables.)

Six concepts form the foundation of HBase. Data is organized into named *tables*, which are identified by strings. A table is a collection of *rows*, each of which is uniquely identified by its row key (a byte array). Row attributes are grouped into named *column families*. Attributes within a column family are addressed via *column qualifiers* (byte arrays). A *cell*, identified by a combination of row key, column group, and column qualifiers, is the logical placeholder of a value (byte array). Finally, a cell’s values are *versioned* by their (integer) timestamp. The combination of row key, column family, column qualifier, and version is also referred to as a (cell) value’s *coordinates* in an HBase table.

```
<HBase Table>
{
  <row key 0>: {
    "<column family 1>": {
      "<qualifier A>": {
          "<version 0>": <cell value>,
          "<version 1>": <cell value>,
          "<version 2>": <cell value>,
      }
      "<qualifier B>": <cell value>,
      ...
    } ,
    "<column family 2>": {
      "<qualifier X>": {
          "<version 0>": <cell value>,
          "<version 1>": <cell value>,
          ...
      }
      "<qualifier Y>": <cell value>,
      ...
    } ,
    ...
  }
  <row key 1>: {
    "<column family 2>": {
      "<qualifier Y>": {
          "<version 0>": <cell value>,
          "<version 1>": <cell value>,
          "<version 2>": <example>
      }
      ...
    } ,
  }
}

// coordinate example
// (<row key 1>, <column family 2>, <qualifier Y>, <version 2>) -> <example>
```

An HBase table's row key is akin to the internal `row_id` in relational tables. (A candidate or primary key on a relational table is composed of one or more relational columns, and is a very different kind of key.) A relational table always has only one column family and the column names are the counterparts of HBase column qualifiers. Typically, cells in relational tables are not versioned. Unlike in a relational schema, HBase column qualifiers need not to be specified in advance, they do not have to be the same between rows, and there is no concept of a `NULL` value. The column families of an HBase table must be specified at creation time, but there is no obligation that a row must store data in all of them. HBase is an example of a key-value store, with the coordinates (row key, column family and qualifier, version) as the key and the cell value as its value.

HBase offers five primitive operations, `Get`, `Put`, `Delete`, `Scan`, and `Increment`. For this paper, only `Get`, `Put`, and `Scan` are relevant. `Get` is used to retrieve a row or subset of cells by row key. `Put` is used to insert or update a row. `Scan` is used to retrieve a collection of rows whose row key falls into a given key range. HBase has no concept of a relational `SELECT` on attributes, or of secondary indexes. Selecting a good row key is maybe the most important design decision. *"It's about the questions, not the relationships.* When designing for HBase, it's about efficiently looking up the answer to a question, not purity of the entity model." ([DimidukKhurana2012], p. 124)

It is common to use a JSON notation to describe the structure of HBase tables:

```
{
  <row key>: {
    "<column family>": {
      "<qualifier>": <cell value type>,
      "<qualifier>": <cell value type>,
      ...
    } ,
    "<column family>": {
      "<qualifier>": <cell value type>,
      "<qualifier>": <cell value type>,
      …
    } ,
    ...
  }
}
```

Note that versions are not explicitly represented in that structure.

An "HBase Schema" for HDF5

With the exception of the table and the column family names, everything else is just a `byte[]` for HBase, and occasional indications of type are mere documentation and carry no meaning in HBase. All HDF5 HBase tables use a single column family called `hdf5`. There might be additional application domain-specific column families, but one column family covers all HDF5 essentials. Also, only one version of each cell value is required.

There are seven tables in this schema for HDF5 domains, groups, links, datasets, attributes, datatypes, and HDF5 path names. The first six are essential; the last is more of a convenience.

HDF5 Domains

We use the MD5 hash of a domain's Digital Object Identifier (DOI) name [14] as its row key. DOI names are variable length strings. By hashing, we produce a fixed-length row key. Since the hash is one-way, we need to store the actual DOI name in a separate column. The root column holds the UUID (row key) of the HDF5 root group. The two other columns hold the created and last modified ISO 8601 UTC date/time.

```
<HDF5 DOMAINS Table>
{
  MD5(<doi>): {
    "hdf5": {
      "DOI":          <doi>,
      "root":         <uuid>,
      "created":      <datetime>,
      "lastModified": <datetime>
    }
  }
}
```

This is a minimalistic domain representation: Good candidates for additional columns would be a JSON digest or a reference to one or more HDF5 profiles.

HDF5 Groups

We use UUIDs (128-bit) as row keys for groups. Other than a back-reference to the hosting domain, HDF5 path name, and date/time stamps, there is not much to keep track of at this level. Really? Where are the group members or links? Read on!

```
<HDF5 GROUPS Table>
```

```
{
  <uuid>: {
    "hdf5": {
      "domain_id":    MD5(<doi>),
      "aPath":        <string>,
      "created":      <datetime>,
      "lastModified": <datetime>
    }
  }
}
```

HDF5 Links

HDF5 links refer to other HDF5 objects, which might be called group members. In order to find a group’s members, we need know their row keys. Since links have names, we could concatenate the group’s UUID and the link name and that would fit the bill. In order to keep to avoid the variable-length link names in the key, we use their MD5 hash instead.[DimidukKhurana2012]

If we don't know the link name, we'll have to rely on `Scan`. It takes start and stop row keys as arguments. Assuming that the group’s UUID is `212d...866f`, the scan range is `212d...866f00...00` to `212d...8670ff...ff`. HBase will return all rows with keys in that range, which correspond to our links / group members.

The group UUID is stored for reverse lookup. The link’s referent can be the row key (`idref`) of another object (the object type is then stored in `H5O_TYPE`), an HDF5 path name (`h5path`), or an external link (`href`).

```
<HDF5 LINKS Table>
{
  <uuid>+MD5(name): {
    "hdf5": {
      "group_id":     <uuid>,
      "name":         <string>,
      "idref":        <uuid>,
      "H5O_TYPE":     <string>,
      "h5path":       <string>,
      "href":         <uri>,
      "created":      <datetime>,
      "lastModified": <datetime>
    }
  }
}
```

HDF5 Array Variables

HDF5 array variables come in a *named* and a *linked* flavor. As named variables or HDF5 attributes, they decorate other HDF5 objects and exist in the context of the attributee. As linked array variables or HDF5 datasets, they are linked to one or more HDF5 groups. An array variable is characterized by the type of its array elements, the shape of the array (up to 32 dimensions), and the aggregate value. The representation and storage of the values of array variable is a separate topic and covered later in this section.

HDF5 Datasets

HDF5 datasets are linked array variables. We maintain a reference to the hosting domain, and the `type` and `shape` columns hold JSON representations. The JSON representation of the type is pretty much its

Avro schema. The dataset value is either inlined or stored in an external Avro file, in which case a JSON descriptor is used to locate the actual value.

```
<HDF5 DATASETS Table>
{
  <uuid>: {
    "hdf5": {
      "domain_id":    MD5(<doi>),
      "type":         <json>,
      "shape":        <json>,
      "value":        <avro>,
      "extern":       <json>,
      "aPath":        <string>,
      "created":      <datetime>,
      "lastModified": <datetime>
    }
  }
}
```

HDF5 Attributes

For HDF5 attributes, we use a composite key similar to the row keys in the links table: An attribute row key is the concatenation of the attributee's UUID with the MD5 hash of the attribute name. That way we can easily scan this table for an object's attributes or directly access an individual attribute.

```
<HDF5 ATTRIBUTES Table>
{
  <uuid>+MD5(name): {
    "hdf5": {
      "name":         <string>,
      "attributee":   <uuid>,
      "type":         <json>,
      "shape":        <json>,
      "value":        <avro>,
      "extern":       <json>,
      "created":      <datetime>,
      "lastModified": <datetime>
    }
  }
}
```

In the next section, we discuss how to deal with non-inlined values of HDF5 datasets and attributes.

HDF5 Array Variable Values

It is not recommended to store arbitrarily large byte arrays in HBase cells. HBase tables are stored in HFiles which are organized in indexed blocks (default size: 64 KB) as the smallest unit that is read from/written to disk. We "inline" array values, i.e., store their Avro representation directly in a column, if the size does not exceed 64 KB. This is very similar to the *compact storage* of HDF5 attributes and datasets in HDF5 files. For larger values, we will use external Avro storage, an idea very similar to HDF5 *external storage*. In this case, the `extern` column will hold a JSON descriptor of the following structure:

```
{
  files: [
    {
      "name":   <HDFS path of Avro file>,
      "offset": <long>,
      "count":  <long>
    },
    ...
  ]
}
```

The files property is a JSON array of segments of records in Avro files stored in HDFS. A segment is characterized by an offset (the number of records to skip) and a count (in records). Multiple such segments may back a dataset or attribute value. Obviously, the Avro files must all have the same schema.

This kind of layout adds great flexibility to the data organization. In particular, it can help reduce the number of small files (< 64 MB) and the pressure on the HDFS name node.

What about element *order*? The data elements are stored across segments as one logical stream in C array order. Hence, the order of the segments in the JSON descriptor is vital, if the multidimensional structure or array order is relevant to a task.

HDF5 Datatype Objects

We think of HDF5 datatype objects as linked (to the group structure) Avro schemata.

```
<HDF5 DATATYPES Table>
{
  <uuid>: {
    "hdf5": {
      "domain_id":    MD5(<doi>),
      "schema":       <json>,
      "aPath":        <string>,
      "created":      <datetime>,
      "lastModified": <datetime>
    }
  }
}
```

HDF5 Path Names

For convenience, we store an "HDF5 object by path name" lookup table.

```
<HDF5 PATH NAMES Table>
{
  <uuid>+MD5(path): {
    "hdf5": {
      "id":       <uuid>,
      "H5O_TYPE": <string>
    }
  }
}
```

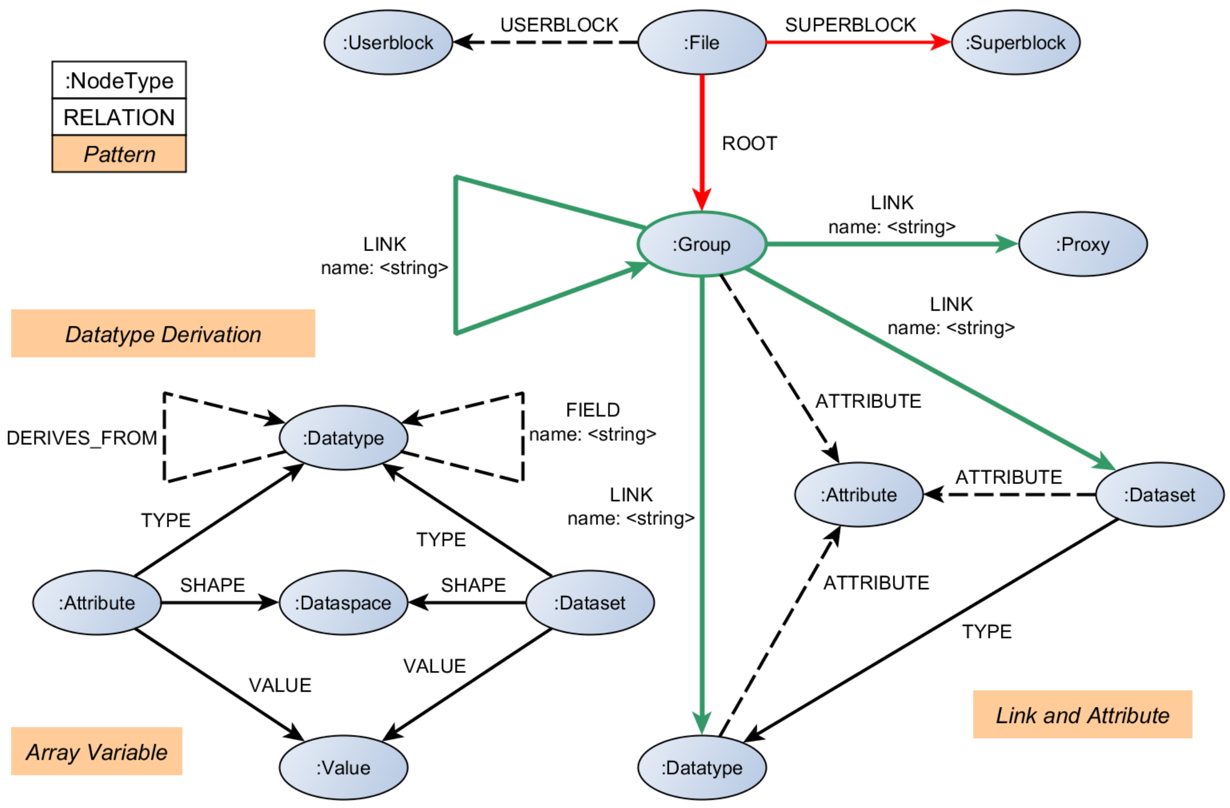
The row key is the usual suspect, a concatenation of the domain row key and the MD5 hash of the HDF5 path name. The columns are the object's row key and the object type (group, dataset, datatype). This table is both redundant and as complete or incomplete as needed. It is redundant, since it can be generated from the other tables and it might be incomplete since only some of the paths leading to a given HDF5 object might be included. (If the group structure contains cycles, then only paths to a certain length will be included.)

If the HBase representation is not your cup of tea, have a look at the next section where we introduce a graph database representation.

D. HDF5 and Graph Databases

In the section called “What is HDF5?”, we mentioned that, in HDF5 data containers, array variables are arranged in property graph-like structures. Below, we show an example of how to represent this structure in a Neo4j graph database [RobinsonWebberEifrem2013]. At the heart of such a representation are patterns or primitives from which more complex graphs are constructed. Depending on how one counts, there are four HDF5 patterns: link, attribute, array variable, and datatype derivation.

Figure D.1. HDF5 Patterns



We have taken the example from section 4 of [H5DDL] and interspersed it with the corresponding Cypher[1] code.

```
// HDF5 "example.h5" {

CREATE (f:File
           {
               id: 'e203fee7-89b4-4216-894d-7aef0e3a199d',
               created: '1985-04-12T23:20:50Z'
           }
       ),

       (ub:Userblock {size: 0}),
       (f)-[:USERBLOCK]->(ub),
```

[1] This is the name of the Neo4j query language.

```
        (sb:Superblock
            {
                superblock_version: 2,
                freelist_version: 0,
                symboltable_version: 0,
                objectheader_version: 0,
                offset_size: 8,
                length_size: 8,
                btree_rank: 16,
                btree_leaf: 4,
                file_driver: 'H5FD_SEC2',
                istore_k: 32
             }
        ),
        (f)-[:SUPERBLOCK]->(sb),

// GROUP "/" {

        (g1:Group { id: '903d1d75-e617-4767-a3bf-0cb3ee509027' }),
        (f)-[:ROOT]->(g1),

//   ATTRIBUTE "attr1" {
//      DATATYPE H5T_STRING {
//           STRSIZE 17;
//           STRPAD H5T_STR_NULLTERM;
//           CSET H5T_CSET_ASCII;
//           CTYPE H5T_C_S1;
//         }
//      DATASPACE SCALAR
//      DATA {
//         "string attribute"
//      }
//   }

        (a1:Attribute { name: 'attr1', value: 'string attribute' }),
        (a1t:Datatype { class: 'H5T_STRING', size: 17 }),
        (a1s:Dataspace { class: 'H5S_SCALAR' }),
        (a1)-[:TYPE]->(a1t),
        (a1)-[:SHAPE]->(a1s),

        (g1)-[:ATTRIBUTE]->(a1),

//   DATASET "dset1" {
//      DATATYPE H5T_STD_I32BE
//      DATASPACE SIMPLE { ( 10, 10 ) / ( 10, 10 ) }
//   }

        (d1:Dataset { id: '30292613-8d2a-4dc4-a277-b9d59d5b0d20' }),
        (d1t:Datatype { class: 'H5T_INTEGER', predefined: 'H5T_STD_I32BE' }),
        (d1s:Dataspace { class: 'H5S_SIMPLE', dims: [10,10] }),
        (d1)-[:TYPE]->(d1t),
        (d1)-[:SHAPE]->(d1s),

        (g1)-[:LINK {name: 'dset1'}]->(d1),

//   DATASET "dset2" {
//      DATATYPE H5T_COMPOUND {
//         H5T_STD_I32BE "a";
//         H5T_IEEE_F32BE "b";
//         H5T_IEEE_F64BE "c";
```

```
   //      }
   //      DATASPACE SIMPLE { ( 5 ) / ( 5 ) }
   //   }

          (d2:Dataset { id: '0a68caca-629a-44aa-9f37-311e7ffb8417' }),
          (d2t:Datatype { class: 'H5T_COMPOUND', field_count: 3 }),
          (d2tf1:Datatype { class: 'H5T_INTEGER', predefined: 'H5T_STD_I32BE' }),
          (d2t)-[:FIELD {name: 'a'}]->(d2tf1),
          (d2tf2:Datatype { class: 'H5T_FLOAT', predefined: 'H5T_IEEE_F32BE' }),
          (d2t)-[:FIELD {name: 'b'}]->(d2tf2),
          (d2tf3:Datatype { class: 'H5T_FLOAT', predefined: 'H5T_IEEE_F64BE' }),
          (d2t)-[:FIELD {name: 'c'}]->(d2tf3),
          (d2s:Dataspace { class: 'H5S_SIMPLE', dims: [5] }),
          (d2)-[:TYPE]->(d2t),
          (d2)-[:SHAPE]->(d2s),

          (g1)-[:LINK {name: 'dset2'}]->(d2),

   //   DATATYPE "type1" H5T_COMPOUND {
   //      H5T_ARRAY { [4] H5T_STD_I32BE } "a";
   //      H5T_ARRAY { [5][6] H5T_IEEE_F32BE } "b";
   //   }

          (t1:Datatype
              {
                class: 'H5T_COMPOUND',
                field_count: 2,
                id: 'a93ff089-d466-44e7-b3f0-09db34ec2ef5'
              }
          ),
          (t2:Datatype { class: 'H5T_ARRAY', dims: [4] }),
          (t3:Datatype { class: 'H5T_INTEGER', predefined: 'H5T_STD_I32BE' }),
          (t2)-[:DERIVES_FROM]->(t3),
          (t4:Datatype { class: 'H5T_ARRAY', dims: [5,6] }),
          (t5:Datatype { class: 'H5T_FLOAT', predefined: 'H5T_IEEE_F32BE' }),
          (t4)-[:DERIVES_FROM]->(t5),
          (t1)-[:FIELD {name: 'a'}]->(t2),
          (t1)-[:FIELD {name: 'b'}]->(t4),

          (g1)-[:LINK {name: 'type1'}]->(t1),

   //   GROUP "group1" {

          (g2:Group { id: 'be8dcb22-b411-4439-85e9-ea384a685ae0' }),
          (g1)-[:LINK {name: 'group1'}]->(g2),

   //      DATASET "dset3" {
   //         DATATYPE "/type1"
   //         DATASPACE SIMPLE { ( 5 ) / ( 5 ) }
   //      }
   //   }

          (d3:Dataset { id: '42f5e3a2-5e70-4faf-9893-fd216257a0d9' }),
          (d3s:Dataspace { class: 'H5S_SIMPLE', dims: [4] }),
          (d3)-[:TYPE]->(t1),
          (d3)-[:SHAPE]->(d3s),

          (g2)-[:LINK {name: 'dset3'}]->(d3),

   //   DATASET "dset3" {
```

```
//      DATATYPE H5T_VLEN { H5T_STD_I32LE }
//      DATASPACE SIMPLE { ( 4 ) / ( 4 ) }
//   }

       (d4:Dataset { id: '4b43748e-817f-44c6-a9f1-16e242fd374b' }),
       (d4t:Datatype { class: 'H5T_VLEN' }),
       (d4tb:Datatype { class: 'H5T_INTEGER', predefined: 'H5T_STD_I32BE' }),
       (d4t)-[:DERIVES_FROM]->(d4tb),
       (d4s:Dataspace { class: 'H5S_SIMPLE', dims: [5] }),
       (d4)-[:TYPE]->(d4t),
       (d4)-[:SHAPE]->(d4s),

       (g1)-[:LINK { name: 'dset3'}]->(d4),

//   GROUP "group2" {
//      HARDLINK "/group1"
//   }

       (g1)-[:LINK {name: 'group2'}]->(g2),

//   SOFTLINK "slink1" {
//      LINKTARGET "somevalue"
//   }

       (p1:Proxy { href: 'somevalue' }),
       (g1)-[:LINK {name: 'slink1'}]->(p1)

// }
// }
```

Exercise: Can you think of a Cypher query which returns all HDF5 path names up to a given length?

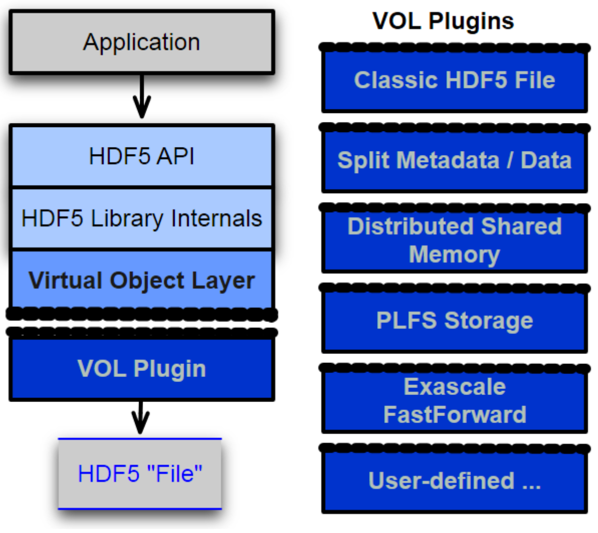
E. The HDF5 Virtual Object Layer (VOL)

In order to deal with large HDF5 containers or datasets, earlier we described an approach that would amount to writing another HDF5 file format parser. Once the data is "in the box", unfortunately, this is the only way out. In this section, we describe the HDF5 virtual object layer (VOL) [ChaarawiKoziol2011], which offers a better foundation for implementing an "HDFS back-end" for HDF5 and which maintains the familiar HDF5 API while taking advantage of many of the unique capabilities of HDFS. Sounds too good to be true? Let's take a step back and try to get our bearings!

We like to tell our users that there are layers and opportunities above (closer to the user) and below (closer to the storage) the HDF5 library, which has its own layers. One of the customization interfaces below the library is the virtual file driver (VFD) interface [H5VFL]. A VFD maps the linear HDF5 file format space onto storage. The simplest mapping goes directly to a single file, but there are VFDs, for example, for multi-file layouts and parallel file systems. For a moment we might entertain the idea of creating an HDF5 virtual file driver for HDFS. The `libhdfs` C-API [libhdfs] has some of the routines that are necessary to implement such a VFD. Alas, VFDs sit *below* the HDF5 library at which point user-level objects, such as datasets, are gone and everything has dissolved into a byte stream, — a DNA base sequence whose genes (HDF5 objects) can be recovered only by a painstaking process.

By contrast, the HDF5 VOL layer sits just below the public HDF5 API. (See Figure E.1, “HDF5 Virtual Object Layer”.) The main advantage is that at this level it's about mapping entire HDF5 objects rather than message streams or blocks of bytes. For example, HDF5 groups might be mapped to directories in a files system or tags in an XML file, and datasets might be represented as individual files. How about a combination of an HBase database and Avro files for datasets? See [MehtaEtAl2012], for an example of such a VOL *plugin* for PLFS.

Figure E.1. HDF5 Virtual Object Layer



In both cases, VFD and VOL, the goal is to maintain the HDF5 API and data model, and to take advantage of the capabilities offered by the underlying storage layer. From a control and data flow perspective, VOL is an *early* optimization and VFD is a *late* optimization.

F. An HDF5 Container Input Format

This extension of `FileInputFormat<K,V>` treats an HDF5 file as a non-splittable BLOB. A map task with this input format reads exactly one record with a `NullWritable` key and a `BytesWritable` value, which is the HDF5 file's byte sequence.

```
package org.hdfgroup.bighdf;

import java.io.IOException;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

public class HDF5ContainerInputFormat extends
    FileInputFormat<NullWritable, BytesWritable> {

 @Override
 protected boolean isSplitable(JobContext context, Path filename) {
  return false;
 }

 @Override
 public RecordReader<NullWritable, BytesWritable> createRecordReader(
   InputSplit inputSplit, TaskAttemptContext context)
        throws IOException, InterruptedException {
  HDF5ContainerRecordReader reader = new HDF5ContainerRecordReader();
  reader.initialize(inputSplit, context);
  return reader;
 }
}
```

```
package org.hdfgroup.bighdf;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.io.BytesWritable;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;

public class HDF5ContainerRecordReader extends
  RecordReader<NullWritable, BytesWritable> {

 private FileSplit split;
 private Configuration conf;

 private final BytesWritable currValue = new BytesWritable();
 private boolean fileProcessed = false;

 @Override
 public void initialize(InputSplit split, TaskAttemptContext context)
   throws IOException, InterruptedException {
  this.split = (FileSplit)split;
  this.conf = context.getConfiguration();
 }

 @Override
 public boolean nextKeyValue() throws IOException, InterruptedException {
  if ( fileProcessed ){
   return false;
  }

  int fileLength = (int)split.getLength();
  byte [] result = new byte[fileLength];

  FileSystem  fs = FileSystem.get(conf);
  FSDataInputStream in = null;
  try {
   in = fs.open( split.getPath());
   IOUtils.readFully(in, result, 0, fileLength);
   currValue.set(result, 0, fileLength);
```

```
     } finally {
      IOUtils.closeStream(in);
     }
     this.fileProcessed = true;
     return true;
    }

    @Override
    public NullWritable getCurrentKey() throws IOException,
      InterruptedException {
     return NullWritable.get();
    }

    @Override
    public BytesWritable getCurrentValue() throws IOException,
      InterruptedException {
     return currValue;
    }

    @Override
    public float getProgress() throws IOException, InterruptedException {
     return 0;
    }

    @Override
    public void close() throws IOException {
    }
  }
```

G. HDF Folklore

Figure G.1. An Anonymous Artist's Impression of Big HDF (AD 1998)

For more information on the story behind HDF see [H5HIST].

Bibliography

[AhlAllen1996] *Hierarchy Theory*. A Vision, Vocabulary, and Epistemology. Valerie Ahl and T. F. H. Allen. Columbia University Press. New York, NY. 15 October 1996.

[Avro] *Apache Avro Home [http://avro.apache.org/]*. Apache Foundation.

[Bar-Hillel1960] *Advances in Computers*. 1. 1960. “A Demonstration of the Nonfeasibility of Fully Automatic High Quality Translation”. Yehoshua Bar-Hillel. 174-179.

[BNFDDL] *DDL in BNF for HDF5 [http://www.hdfgroup.org/HDF5/doc/ddl.html]*. The HDF Group. 2010.

[Borthakur2013] *HDFS Architecture Guide [http://avro.apache.org/]*. Dhruba Borthakur. Apache Foundation. 4 August 2013.

[Buck2011] *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. New York NY USA . “SciHadoop: Array-based Query Processing in Hadoop”. Joe B. Buck et al.. 12 November 2011. DOI:10.1145/2063384.2063473.

[Byna2013] *Cray User Group Meeting*. “ Trillion Particles, 120,000 cores, and 350 TBs: Lessons Learned from a Hero I/O Run on Hopper [https://sdm.lbl.gov/exahdf5/papers/2013-CUG_byna.pdf]”. Surendra Byna et al.. 2013. DOI:10.1145/2063384.2063473.

[CaprioloWamplerRutherglen2012] *Programming Hive*. Edward Capriolo, Dean Wampler, and Jason Rutherglen. O'Reilly Media. 3 October 2012.

[ChaarawiKoziol2011] *RFC*. “Virtual Object Layer”. Mohamad Chaarawi and Quincey Koziol. 17 October 2011.

[Chang2006] *OSDI'06: Seventh Symposium on Operating System Design and Implementation*. “Bigtable: A Distributed Storage System for Structured Data”. Fay Chang et al.. November 2006.

[DimidukKhurana2012] *HBase in Action* . Nick Demiduk and Amandeep Khurana. Manning Publications. 14 November 2012.

[Gates2011] *Programming Pig*. Alan Gates. O'Reilly Media. 20 October 2011.

[Gitelman2013] *"Raw Data" is an Oxymoron*. Lisa Gitelman. The MIT Press. 25 Jabuary 2013.

[George2011] *HBase*. The Definitive Guide. Lars George. O'Reilly Media. 20 September 2011.

[H5DDL] *Data Description Language in BNF for HDF5 [http://www.hdfgroup.org/HDF5/doc/ddl.html]*. The HDF Group. 25 September 2012.

[H5FAQ] *Frequently Asked Questions about HDF5 [http://www.hdfgroup.org/HDF5-FAQ.html]*. The HDF Group. 31 January 2014.

[H5FF] *HDF5 File Format Specification Version 2.0 [http://www.hdfgroup.org/HDF5/doc/H5.format.html]*. The HDF Group. 12 January 2012.

[H5HIST] *HDF Group History [http://www.hdfgroup.org/about/history.html]*. The HDF Group. 16 March 2011.

[H5RM] *HDF5: API Specification — Reference Manual [http://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html]*. The HDF Group. 8 November 2013.

[H5VFL] *HDF5 Virtual File Layer [http://www.hdfgroup.org/HDF5/doc/TechNotes/VFL.html]*. The HDF Group. 18 November 1999.

[Hadoop] *Apache Hadoop Home [http://hadoop.apache.org/]*. Apache Foundation. 23 January 2014.

[HBase] *Apache HBase Home [http://hbase.apache.org/]*. Apache Foundation.

[Holmes2012] *Hadoop in Practice* . Alex Holmes. Manning Publications. 10 October 2012.

[JHI5] *Java Native Interface for HDF5 [http://www.hdfgroup.org/products/java/hdf-java-html/JNI/]*. The HDF Group. 2013.

[Juric2011] *Large Survey Database: A Distributed Framework for Storage and Analysis of Large Datasets [http://adsabs.harvard.edu/abs/2011AAS...21743319J]* . Mario Juric.

[libhdfs] *C API libhdfs [http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-hdfs/LibHdfs.html]* . Apache Hadoop Project .

[McCallum2012] *Bad Data Handbook*. Q. Ethan McCallum. O'Reilly Media. 21 November 2012.

[Mealy1967] *AFIPS'67*. ACM Press. New York, NY. “Another look at data”. George H. Mealy. 1967. DOI:10.1145/1465611.1465682.

[MehtaEtAl2012] *SC'12*. IEEE. New York, NY. “A Plugin for HDF5 Using PLFS for Improved I/O Performance and Semantic Analysis”. Kshitij Mehta et al.. 2012. DOI:10.1109/SC.Companion.2012.102.

[MinerShook2012] *MapReduce Design Pattern*. Donald Miner and Adam Shook. O'Reilly Media. 20 November 2012.

[MattmannZitting2012] *Tika in Action* . Chris A. Mattmann and Jukka L. Zitting. Manning Publications. 8 December 2011.

[OwenEtAl2011] *Mahout in Action* . Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. Manning Publications. 14 October 2011.

[R] *The R Project for Statistical Computing [http://www.r-project.org/]*. 2014.

[RCMES] *Model Evaluation Using the NASA Regional Climate Model Evaluation System (RCMES) [http://www.earthzine.org/2013/12/02/model-evaluation-using-the-nasa-regional-climate-model-evaluation-system-rcmes/]*. Earthzine. 2013.

[RobinsonWebberEifrem2013] *Graph Databases*. Ian Robinson, Jim Webber, and Emil Eifrem. O'Reilly Media. 17 June 2013.

[Shvachko2010a] *26th Symposium on Mass Storage Systems and Technologies (MSST), 3-7 May 2010, Incline Village, NV*. IEEE. “The Hadoop Distributed File System”. Konstantin V. Shvachko et al.. 1 - 10. 2010. 10.1109/MSST.2010.5496972.

[Shvachko2010b] *;login*. USENIX. “HDFS Scalability: The Limits to Growth”. Konstantin V. Shvachko. 32. 2. 6 - 16. April 2010.

[Wang2013] *Supporting a Light-Weight Data Management Layer over HDF5*. Yi Wang. DOI:10.1109/CCGrid.2013.9 .