BIG DATA

Data storage on the batch layer

INTRODUCTION

- The master dataset is typically too large to exist on a single server
  - we must choose how to distribute data across multiple machines
- In this chapter we'll do the following
  - Determine the requirements for storing the master dataset
  - See why distributed filesystems are a natural fit for storing a master dataset
  - See how the batch layer storage for the SuperWebAnalytics.com project maps to distributed filesystems

STORAGE REQUIREMENTS FOR THE MASTER DATASET

- Data is immutable
  - Each piece of your data will be written once and only once
  - The only write operation will be to add a new data
  - The storage solution must be optimized to handle a large, constantly growing set of data
- The batch layer is also responsible for computing functions on the dataset
  - It needs to be good at reading lots of data at once
  - Random access to individual pieces of data is not required

STORAGE REQUIREMENTS FOR THE MASTER DATASET

- With this "write once, bulk read many times" paradigm in mind, requirements for the data storage are:

Operation	Requisite	Discussion
Write	Efficient appends of new data	The only write operation is to add new pieces of data, so it must be easy and efficient to append a new set of data objects to the master dataset.
	Scalable storage	The batch layer stores the complete dataset—potentially terabytes or petabytes of data. It must therefore be easy to scale the storage as your dataset grows.
Read	Support for parallel processing	Constructing the batch views requires computing functions on the entire master dataset. The batch storage must consequently support parallel processing to handle large amounts of data in a scalable manner.
Both	Tunable storage and processing costs	Storage costs money. You may choose to compress your data to help minimize your expenses, but decompressing your data during computations can affect performance. The batch layer should give you the flexibility to decide how to store and compress your data to suit your specific needs.
	Enforceable immutability	It's critical that you're able to enforce the immutability property on your master dataset. Of course, computers by their very nature are mutable, so there will always be a way to mutate the data you're storing. The best you can do is put checks in place to disallow mutable operations. These checks should prevent bugs or other random errors from trampling over existing data.

CHOOSING A STORAGE SOLUTION FOR THE BATCH LAYER

- Using a key/value store for the master dataset
  - the most common type of distributed database
  - giant persistent hashmaps that are distributed among many machines
  - What should a key be?
  - Need fine-grained access to key/value pairs to do random reads and writes
    - can't compress multiple key/value pairs together
  - Meant to be used as mutable stores
    - can't disable the ability to modify existing key/value pairs
  - Has a lot of things you don't need: random reads, random writes, and all the components making those work
    - enormously complex for your requirements

CHOOSING A STORAGE SOLUTION FOR THE BATCH LAYER

- Filesystems: perfect fit for batch layer storage
  - Files are sequences of bytes
    - They're stored sequentially on disk
    - The most efficient way to consume files is by scanning through them
    - You have full control over the bytes of a file
    - You have the full freedom to compress them however you want
  - A filesystem gives you exactly what you need and no more,
  - Also not limiting your ability to tune storage cost versus processing cost.
  - Filesystems implement fine-grained permissions systems
    - Perfect for enforcing immutability
  - The main problem: they exist on just a single machine
    - Limited scalability

CHOOSING A STORAGE SOLUTION FOR THE BATCH LAYER

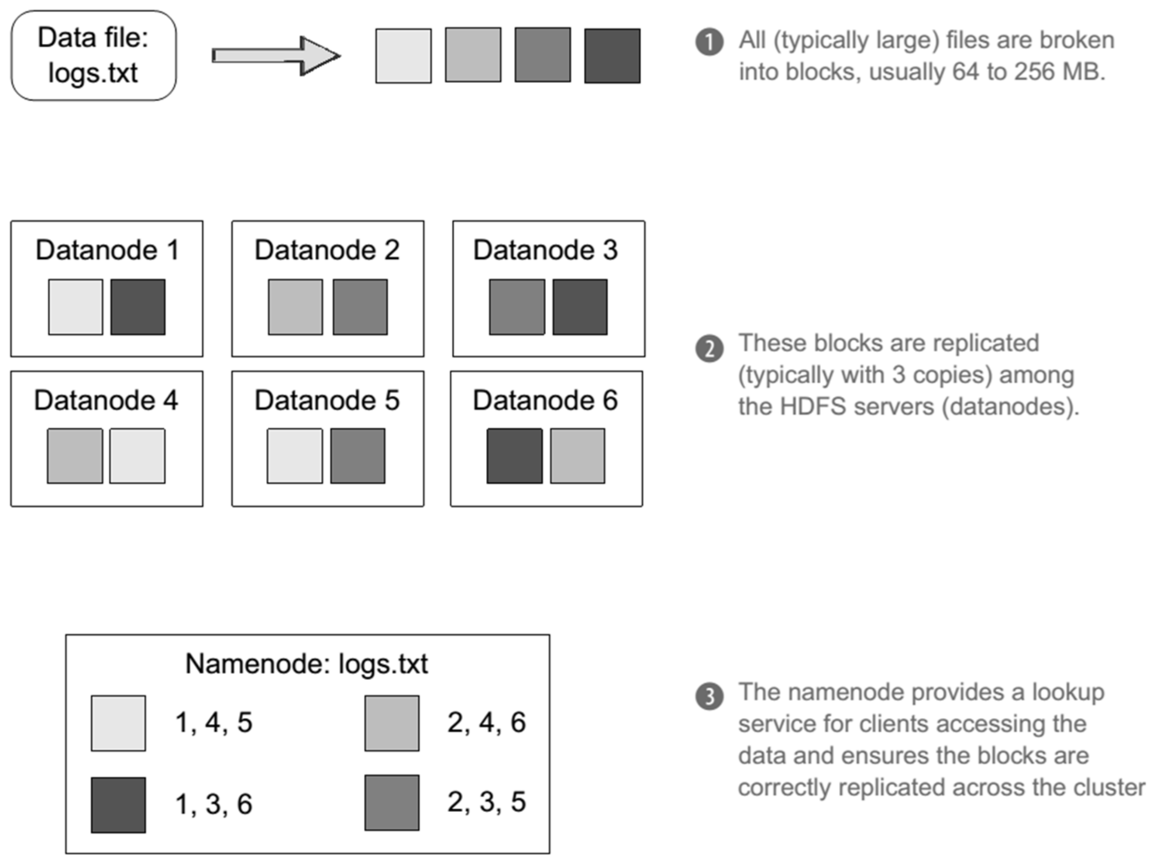
- Distributed filesystems
  - Similar to regular filesystems, except
    - Spread their storage across a cluster of computers
    - Scale by adding more machines to the cluster
    - Designed for tolerating faults
    - Their operations are more limited
      - Not able to write to the middle of a file
      - Not able to modify a file at all after creation
      - Having small files are inefficient

CHOOSING A STORAGE SOLUTION FOR THE BATCH LAYER

- Hadoop Distributed File System (HDFS)
  - HDFS and Hadoop MapReduce are the two prongs of the Hadoop project
  - Hadoop is deployed across multiple servers called a cluster
  - HDFS manages how data is stored across the cluster
  - In an HDFS cluster
    - Single namenode
    - Multiple datanode
    - Files are first chunked into blocks of a fixed size (typically between 64 MB and 256 MB)
    - Each block is then replicated across random chosen datanodes (typically three)
    - The namenode keeps track of the file-to-block mapping and where each block is located

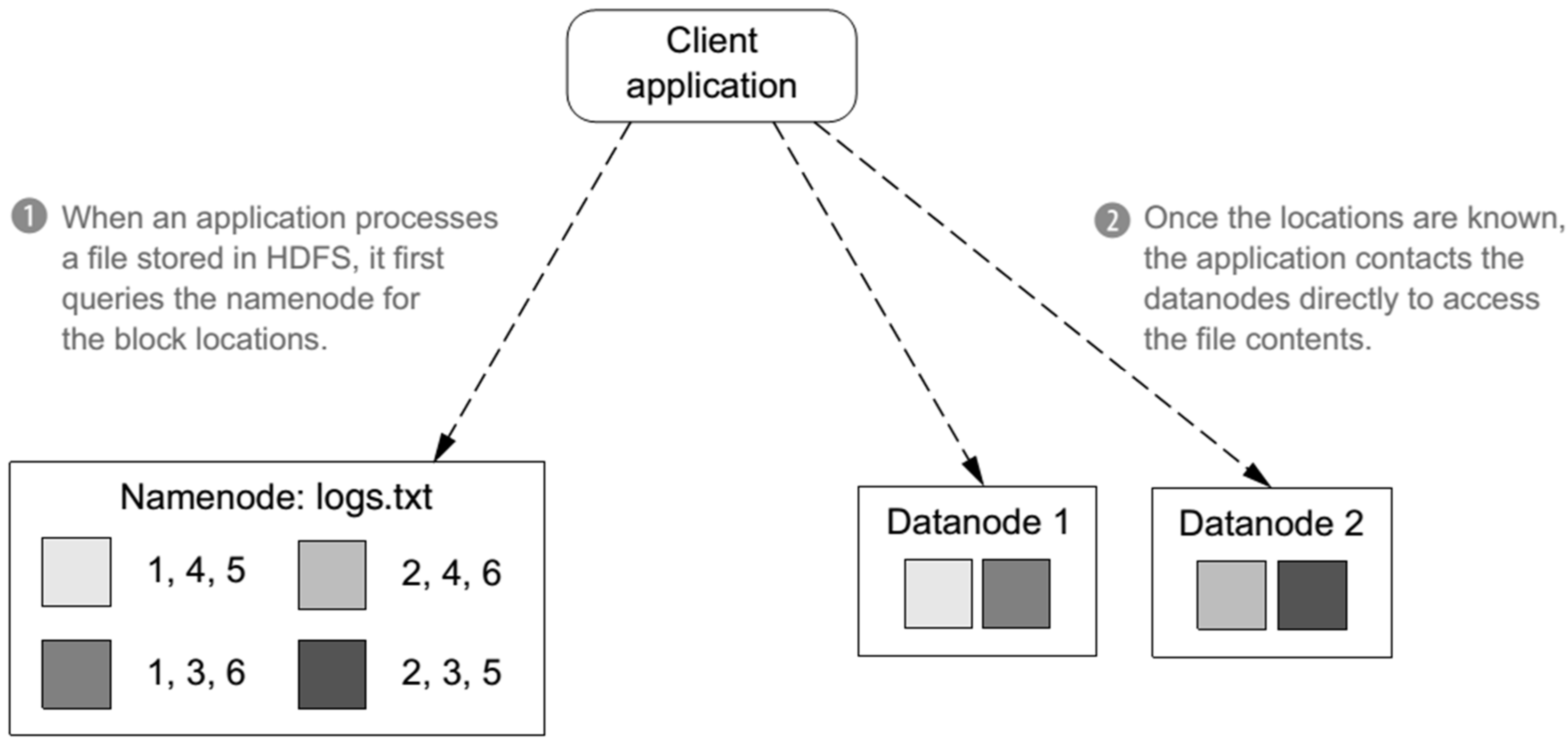
CHOOSING A STORAGE SOLUTION FOR THE BATCH LAYER

- Hadoop Distributed File System (HDFS)



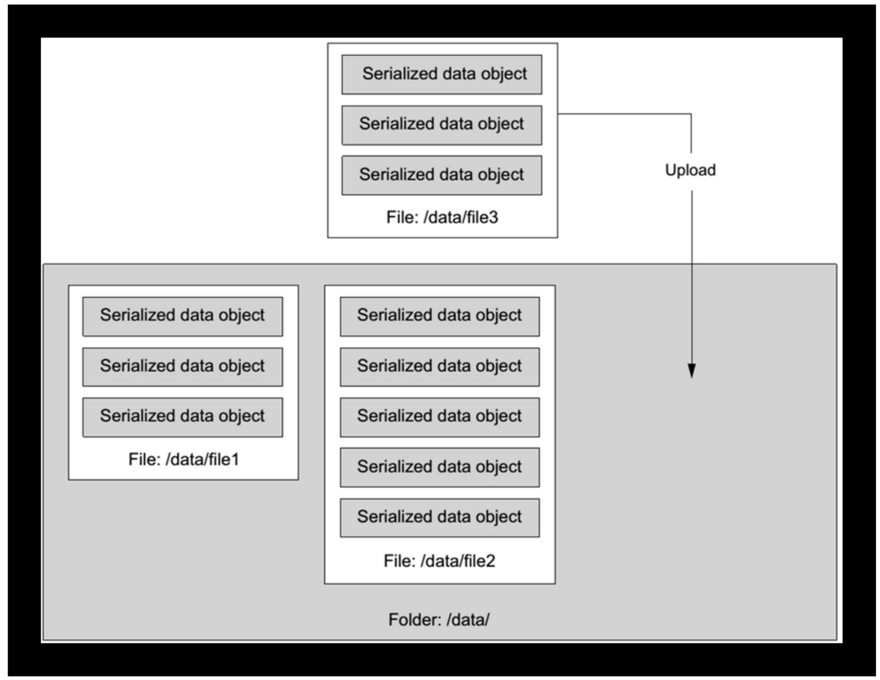
CHOOSING A STORAGE SOLUTION FOR THE BATCH LAYER

- Hadoop Distributed File System (HDFS)



STORING A MASTER DATASET WITH A DISTRIBUTED FILESYSTEM

- Distributed filesystems vary in the kinds of operations they permit.
  - Some let you modify existing files, and others don't.
  - Some allow to append to existing files, and some don't.
- How you can store a master dataset where a file can't be modified at all?
  - spread the master dataset among many files

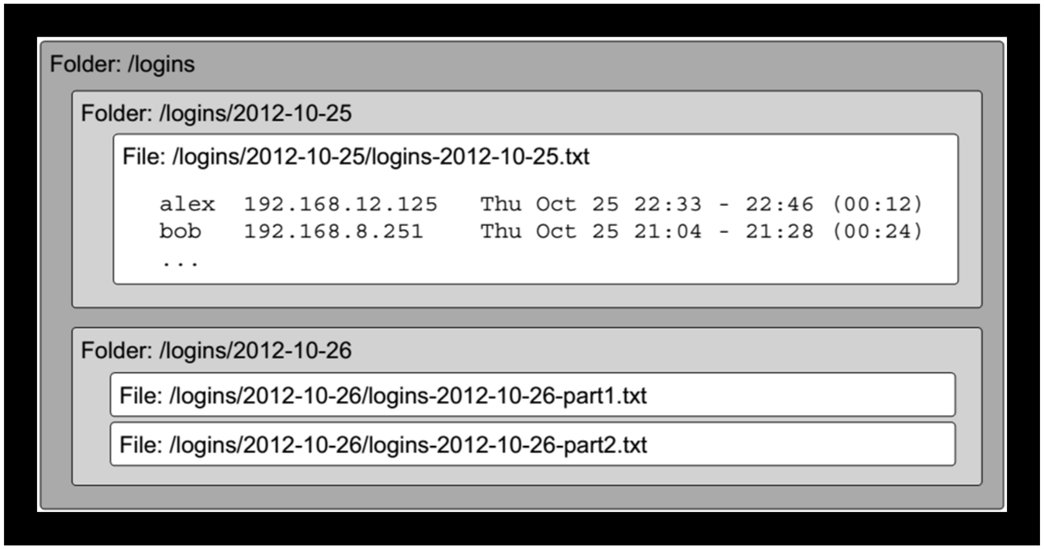


STORING A MASTER DATASET WITH A DISTRIBUTED FILESYSTEM

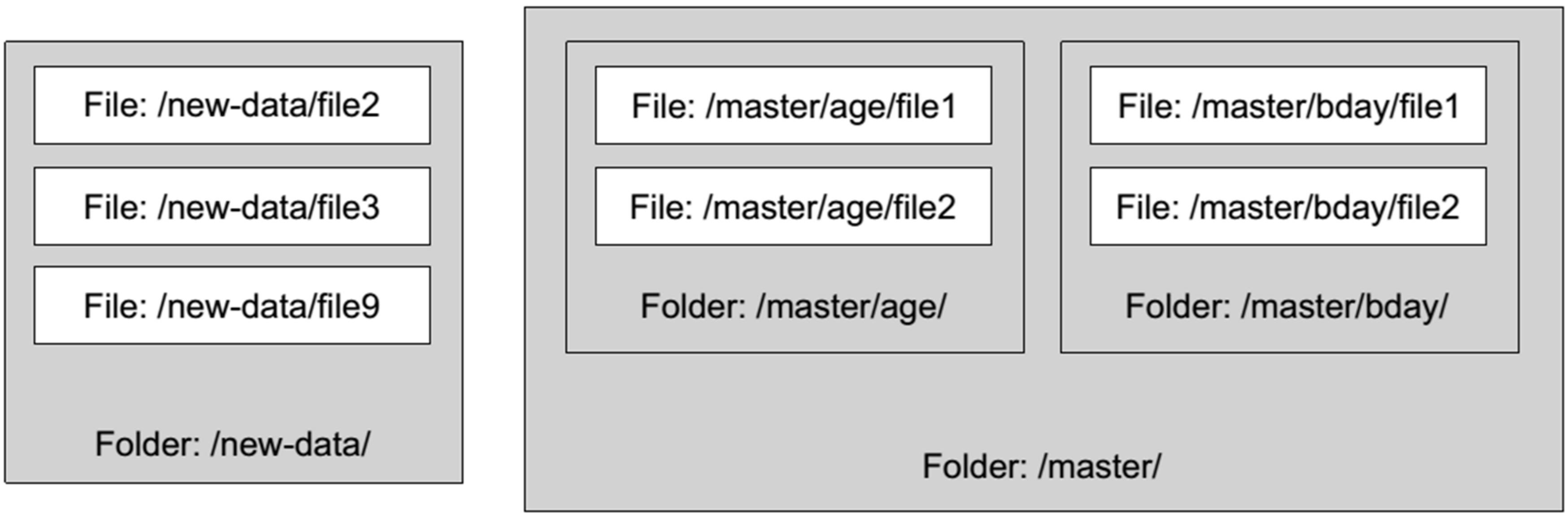
Operation	Requisite	Discussion
Write	Efficient appends of new data	Appending new data is as simple as adding a new file to the folder containing the master dataset.
	Scalable storage	Distributed filesystems evenly distribute the storage across a cluster of machines. You increase storage space and I/O throughput by adding more machines.
Read	Support for parallel processing	Distributed filesystems spread all data across many machines, making it possible to parallelize the processing across many machines. Distributed filesystems typically integrate with computation frameworks like MapReduce to make that processing easy to do (discussed in chapter 6).
Both	Tunable storage and processing costs	Just like regular filesystems, you have full control over how you store your data units within the files. You choose the file format for your data as well as the level of compression. You're free to do individual record compression, block-level compression, or neither.
	Enforceable immutability	Distributed filesystems typically have the same permissions systems you're used to using in regular filesystems. To enforce immutability, you can disable the ability to modify or delete files in the master dataset folder for the user with which your application runs. This redundant check will protect your previously existing data against bugs or other human mistakes.

VERTICAL PARTITIONING

- Vertical partitioning: partitioning data so that a function only accesses data relevant to its computation.
    - it can greatly make the batch layer more efficient
    - can be done by sorting data into separate folders



LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS



ILLUSTRATION

- Using the Hadoop Distributed File System

```
$ cat logins-2012-10-25.txt
alex     192.168.12.125   Thu Oct 25 22:33 - 22:46 (00:12)
bob      192.168.8.251    Thu Oct 25 21:04 - 21:28 (00:24)
charlie  192.168.12.82    Thu Oct 25 21:02 - 23:14 (02:12)
doug     192.168.8.13     Thu Oct 25 20:30 - 21:03 (00:33)
...
```

```
$ hadoop fs -mkdir /logins
$ hadoop fs -put logins-2012-10-25.txt /logins
```

The "hadoop fs" commands are Hadoop shell commands that interact directly with HDFS. A full list is available at http://hadoop.apache.org/.

Uploading a file automatically chunks and distributes the blocks across the datanodes.

```
$ hadoop fs -ls -R /logins
-rw-r--r--   3 hdfs hadoop  175802352 2012-10-26 01:38
    /logins/logins-2012-10-25.txt
```

The ls command is based on the Unix command of the same name.

ILLUSTRATION

- Using the Hadoop Distributed File System

```
$ hadoop fs -cat /logins/logins-2012-10-25.txt
alex     192.168.12.125   Thu Oct 25 22:33 - 22:46 (00:12)
bob      192.168.8.251    Thu Oct 25 21:04 - 21:28 (00:24)
...
```

```
$ hadoop fsck /logins/logins-2012-10-25.txt  -files -blocks -locations

/logins/logins-2012-10-25.txt 175802352 bytes, 2 block(s):
OK
0. blk_-1821909382043065392_1523 len=134217728
   repl=3 [10.100.0.249:50010, 10.100.1.4:50010, 10.100.0.252:50010]
1. blk_2733341693279525583_1524 len=41584624
   repl=3 [10.100.0.255:50010, 10.100.1.2:50010, 10.100.1.5:50010]
```

The file is stored in two blocks.

The IP addresses and port numbers of the datanodes hosting each block

USING THE HADOOP DISTRIBUTED FILE SYSTEM

- The small-files problem
  - computing performance is significantly degraded when data is stored in many small files in HDFS
    - MapReduce job launches multiple tasks, one for each block in the input dataset.
    - Each task requires some overhead to plan and coordinate its execution
    - because each small file requires a separate task, the cost is repeatedly incurred
  - Solution
    - Small files should be consolidated in large files

USING THE HADOOP DISTRIBUTED FILE SYSTEM

- Towards a higher-level abstraction
  - Important operations for manipulating a master dataset in HDFS
    - Appending to a dataset
    - Vertically partitioning a dataset and not allowing an existing partitioning to be violated
    - Efficiently consolidating small files together into larger files
  - We need a tool for accomplishing these tasks in an elegant manner

USING THE HADOOP DISTRIBUTED FILE SYSTEM

- Towards a higher-level abstraction

```java
import java.io.IOException;
import backtype.hadoop.pail.Pail;

public class PailMove {

  public static void mergeData(String masterDir, String updateDir)
    throws IOException
  {
    Pail target = new Pail(masterDir);
    Pail source = new Pail(updateDir);
    target.absorb(source);
    target.consolidate();
  }
}
```

Pails are wrappers around HDFS folders.

With the Pail library, appends are one-line operations.

Small data files within the pail can also be consolidated with a single function call.

DATA STORAGE IN THE BATCH LAYER WITH PAIL

- An abstraction over files and folders
  - http://github.com/nathanmarz/dfs-datastores
  - is just a Java library that uses the standard Hadoop APIs

```
public static void simpleIO() throws IOException {
  Pail pail = Pail.create("/tmp/mypail");
  TypedRecordOutputStream os = pail.openWrite();
  os.writeObject(new byte[] {1, 2, 3});
  os.writeObject(new byte[] {1, 2, 3, 4});
  os.writeObject(new byte[] {1, 2, 3, 4, 5});
  os.close();
}
```

Creates a default pail in the specified directory

Provides an output stream to a new file in the Pail

A pail without metadata is limited to storing byte arrays.

Closes the current file

DATA STORAGE IN THE BATCH LAYER WITH PAIL

- An abstraction over files and folders (http://github.com/nathanmarz/dfs-datastores)

```
root:/ $ ls /tmp/mypail
f2fa3af0-5592-43e0-a29c-fb6b056af8a0.pailfile
pail.meta
```

The records are stored within pailfiles.

The metadata describes the contents and structure of the pail.

```
root:/ $ cat /tmp/mypail/pail.meta
---
format: SequenceFile
args: {}
```

The format of files in the pail; a default pail stores data in key/value pairs within Hadoop SequenceFiles.

The arguments describe the contents of the pail; an empty map directs Pail to treat the data as uncompressed byte arrays.

DATA STORAGE IN THE BATCH LAYER WITH PAIL

- Serializing objects into pails

```
public class Login {
  public String userName;
  public long loginUnixTime;

  public Login(String _user, long _login) {
    userName = _user;
    loginUnixTime = _login;
  }
}
```

DATA STORAGE IN THE BATCH LAYER WITH PAIL

```
public class LoginPailStructure implements PailStructure<Login>{

  public Class getType() {
    return Login.class;
  }

  public byte[] serialize(Login login) {
    ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
    DataOutputStream dataOut = new DataOutputStream(byteOut);
    byte[] userBytes = login.userName.getBytes();
    try {
      dataOut.writeInt(userBytes.length);
      dataOut.write(userBytes);
      dataOut.writeLong(login.loginUnixTime);
      dataOut.close();
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return byteOut.toByteArray();
  }

  public Login deserialize(byte[] serialized) {
    DataInputStream dataIn =
        new DataInputStream(new ByteArrayInputStream(serialized));
    try {
      byte[] userBytes = new byte[dataIn.readInt()];
      dataIn.read(userBytes);
      return new Login(new String(userBytes), dataIn.readLong());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }

  public List<String> getTarget(Login object) {
    return Collections.EMPTY_LIST;
  }

  public boolean isValidTarget(String... dirs) {
    return true;
  }
}
```

A pail with this structure will only store Login objects. (getType)

Login objects must be serialized when stored in pailfiles. (serialize)

Logins are later reconstructed when read from pailfiles. (deserialize)

The getTarget method defines the vertical partitioning scheme, but it's not used in this example.

isValidTarget determines whether the given path matches the vertical partitioning scheme, but it's also not used in this example.

```
public static void writeLogins() throws IOException {
  Pail<Login> loginPail = Pail.create("/tmp/logins",
                                      new LoginPailStructure());
  TypedRecordOutputStream out = loginPail.openWrite();
  out.writeObject(new Login("alex", 1352679231));
  out.writeObject(new Login("bob", 1352674216));
  out.close();
}
```

Creates a pail with the new pail structure

A pail supports the Iterable interface for its object type.

```
public static void readLogins() throws IOException {
  Pail<Login> loginPail = new Pail<Login>("/tmp/logins");
  for(Login l : loginPail) {
    System.out.println(l.userName + " " + l.loginUnixTime);
  }
}
```

DATA STORAGE IN THE BATCH LAYER WITH PAIL

- Batch operations using Pail
  - Pail operations are all implemented using MapReduce
    - they scale regardless of the amount of data
  - The append operation is particularly smart.
    - It checks the pails to verify that it's valid to append the pails together.
    - it won't allow to append a pail containing strings to a pail containing integers.
  - Consolidate operation merges small files to create new files

```
public static void appendData() throws IOException {
  Pail<Login> loginPail = new Pail<Login>("/tmp/logins");
  Pail<Login> updatePail = new Pail<Login>("/tmp/updates");
  loginPail.absorb(updatePail);
  loginPail.consolidate();
}
```

DATA STORAGE IN THE BATCH LAYER WITH PAIL

- Vertical partitioning with Pail

```
public class PartitionedLoginPailStructure extends LoginPailStructure {
  SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");

  public List<String> getTarget(Login object) {
    ArrayList<String> directoryPath = new ArrayList<String>();
    Date date = new Date(object.loginUnixTime * 1000L);
    directoryPath.add(formatter.format(date));
    return directoryPath;
  }

  public boolean isValidTarget(String... strings) {
    if(strings.length != 1) return false;
    try {
      return (formatter.parse(strings[0]) != null);
    }
    catch(ParseException e) {
      return false;
    }
  }
}
```

Logins are vertically partitioned in folders corresponding to the login date.

The timestamp of the Login object is converted to an understandable form.

isValidTarget verifies that the directory structure has a depth of one and that the folder name is a date.

DATA STORAGE IN THE BATCH LAYER WITH PAIL

- Vertical partitioning with Pail

```
public static void partitionData() throws IOException {
  Pail<Login> pail = Pail.create("/tmp/partitioned_logins",
                              new PartitionedLoginPailStructure());
  TypedRecordOutputStream os = pail.openWrite();
  os.writeObject(new Login("chris", 1352702020));
  os.writeObject(new Login("david", 1352788472));
  os.close();
}
```

1352702020 is the timestamp for 2012-11-11, 22:33:40 PST.

1352788472 is the timestamp for 2012-11-12, 22:34:32 PST.

```
root:/ $ ls -R /tmp/partitioned_logins
2012-11-11  2012-11-12  pail.meta

/tmp/partitioned_logins/2012-11-11:
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile

/tmp/partitioned_logins/2012-11-12:
d8c0822b-6caf-4516-9c74-24bf805d565c.pailfile
```

Folders for the different login dates are created within the pail.

DATA STORAGE IN THE BATCH LAYER WITH PAIL

- Pail file formats and compression

```
public static void createCompressedPail() throws IOException {
  Map<String, Object> options = new HashMap<String, Object>();
  options.put(SequenceFileFormat.CODEC_ARG,
              SequenceFileFormat.CODEC_ARG_GZIP);
  options.put(SequenceFileFormat.TYPE_ARG,
              SequenceFileFormat.TYPE_ARG_BLOCK);
  LoginPailStructure struct = new LoginPailStructure();
  Pail compressed = Pail.create("/tmp/compressed",
                         new PailSpec("SequenceFile", options, struct));
}
```

Contents of the pail will be gzip compressed.

Blocks of records will be compressed together (as compared to compressing rows individually).

Creates a new pail to store Login options with the desired format

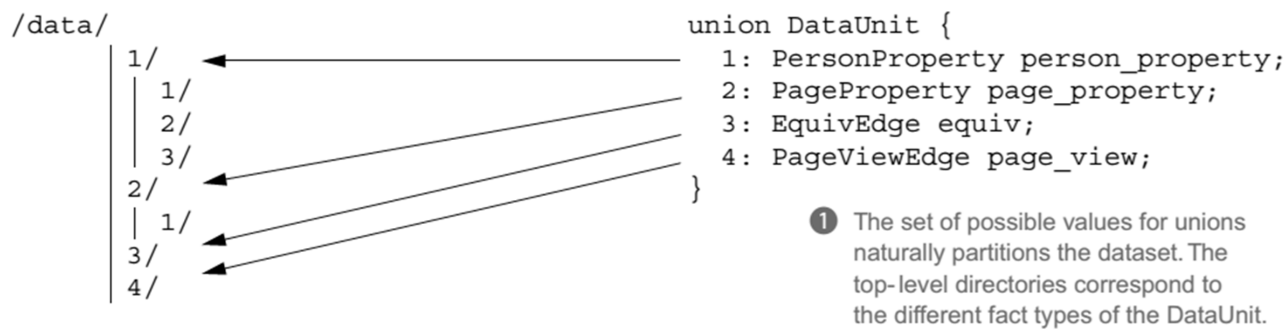
DATA STORAGE IN THE BATCH LAYER WITH PAIL

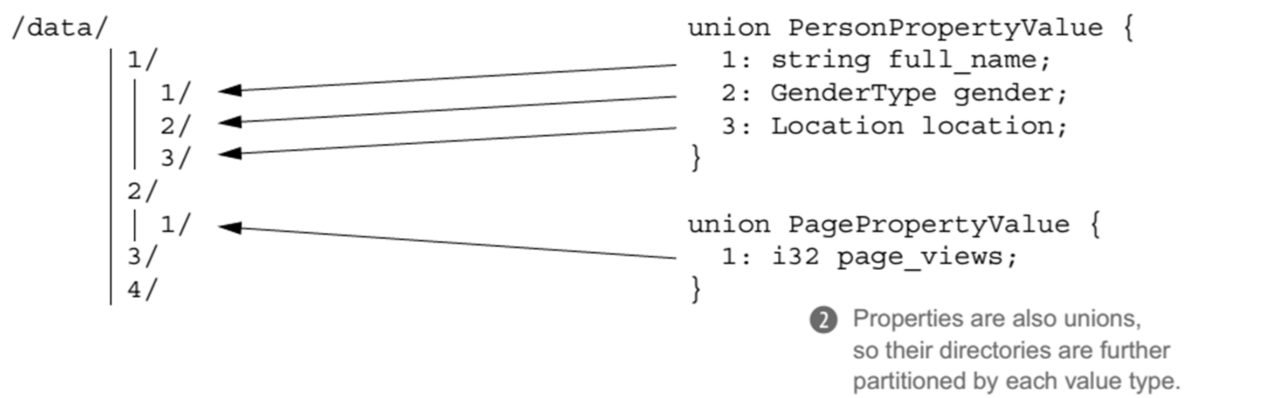
- Summarizing the benefits of Pail

Operation	Criteria	Discussion
Write	Efficient appends of new data	Pail has a first-class interface for appending data and prevents you from performing invalid operations—something the raw HDFS API won't do for you.
	Scalable storage	The namenode holds the entire HDFS namespace in memory and can be taxed if the filesystem contains a vast number of small files. Pail's consolidate operator decreases the total number of HDFS blocks and eases the demand on the namenode.
Read	Support for parallel processing	The number of tasks in a MapReduce job is determined by the number of blocks in the dataset. Consolidating the contents of a pail lowers the number of required tasks and increases the efficiency of processing the data.
	Ability to vertically partition data	Output written into a pail is automatically partitioned with each fact stored in its appropriate directory. This directory structure is strictly enforced for all Pail operations.
Both	Tunable storage/ processing costs	Pail has built-in support to coerce data into the format specified by the pail structure. This coercion occurs automatically while performing operations on the pail.
	Enforceable immutability	Because Pail is just a thin wrapper around files and folders, you can enforce immutability, just as you can with HDFS directly, by setting the appropriate permissions.

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- How to map SuperWebAnalytics.com schema to folders:





STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- Steps to use HDFS and Pail for SuperWebAnalytics.com
  1. create an abstract pail structure for storing Thrift objects
     - Thrift serialization is independent of the type of data being stored
     - cleaner code by separating this logic
  2. derive a pail structure from the abstract class for storing SuperWebAnalytics.com Data objects
  3. define a further subclass that will implement the desired vertical partitioning scheme

don’t worry about the details of the code
this code works for any graph schema

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- A structured pail for Thrift objects

```
public abstract class ThriftPailStructure<T extends Comparable>
  implements PailStructure<T>
{
  private transient TSerializer ser;
  private transient TDeserializer des;

  private TSerializer getSerializer() {
    if(ser==null) ser = new TSerializer();
    return ser;
  }

  private TDeserializer getDeserializer() {
    if(des==null) des = new TDeserializer();
    return des;
  }

  public byte[] serialize(T obj) {
    try {
      return getSerializer().serialize((TBase)obj);
    } catch (TException e) {
      throw new RuntimeException(e);
    }
  }

  public T deserialize(byte[] record) {
    T ret = createThriftObject();
    try {
      getDeserializer().deserialize((TBase)ret, record);
    } catch (TException e) {
      throw new RuntimeException(e);
    }
    return ret;
  }

  protected abstract T createThriftObject();
}
```

Java generics allow the pail structure to be used for any Thrift object.

TSerializer and TDeserializer are Thrift utilities for serializing objects to and from binary arrays.

The Thrift utilities are lazily built, constructed only when required.

The object is cast to a basic Thrift object for serialization.

A new Thrift object is constructed prior to deserialization.

The constructor of the Thrift object must be implemented in the child class.

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- A basic pail for SuperWebAnalytics.com

```
public class DataPailStructure extends ThriftPailStructure<Data> {
  public Class getType() {
    return Data.class;
  }

  protected Data createThriftObject() {
    return new Data();
  }

  public List<String> getTarget(Data object) {
    return Collections.EMPTY_LIST;
  }

  public boolean isValidTarget(String... dirs) {
    return true;
  }
}
```

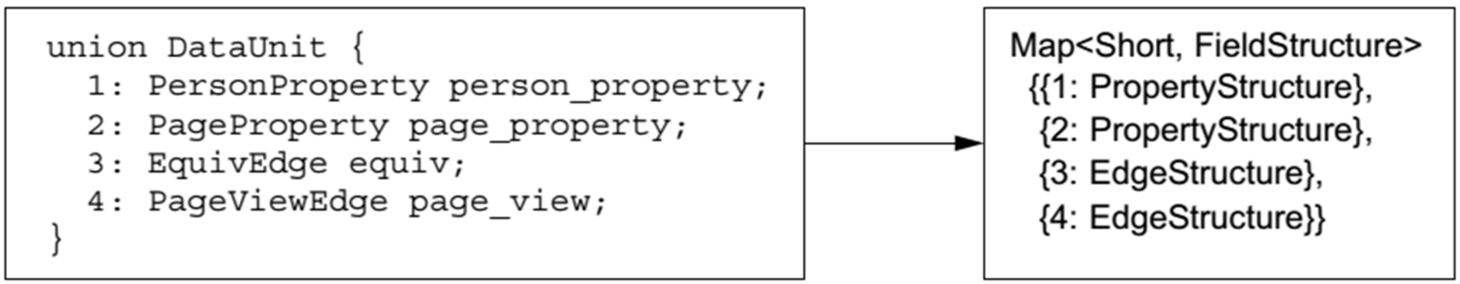
Specifies that the pail stores Data objects (annotates `public Class getType()`)

Needed by ThriftPailStructure to create an object for deserialization (annotates `protected Data createThriftObject()`)

This pail structure doesn't use vertical partitioning. (annotates `public List<String> getTarget(Data object)`)

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- A split pail to vertically partition the dataset
  - SplitDataPailStructure creates a map between Thrift IDs and classes to process the corresponding type



  - The SplitDataPailStructure is responsible for the top-level directory of the vertical partitioning
    - it passes the responsibility of any additional subdirectories to the other classes (FieldStructure interface)

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- A split pail to vertically partition the dataset

```
public class SplitDataPailStructure extends DataPailStructure {

  public static HashMap<Short, FieldStructure> validFieldMap =
    new HashMap<Short, FieldStructure>();

  static {
    for(DataUnit._Fields k: DataUnit.metaDataMap.keySet()) {
      FieldValueMetaData md = DataUnit.metaDataMap.get(k).valueMetaData;
      FieldStructure fieldStruct;
      if(md instanceof StructMetaData &&
        ((StructMetaData) md).structClass
          .getName().endsWith("Property"))
      {
        fieldStruct = new PropertyStructure(
          ((StructMetaData) md).structClass);
      } else {
        fieldStruct = new EdgeStructure();
      }
      validFieldMap.put(k.getThriftFieldId(), fieldStruct);
    }
  }

  // remainder of class elided
}
```

FieldStructure is an interface for both edges and properties.

Thrift code to inspect and iterate over the DataUnit object

Properties are identified by the class name of the inspected object.

If class name doesn't end with "Property", it must be an edge.

```
protected static interface FieldStructure {
  public boolean isValidTarget(String[] dirs);
  public void fillTarget(List<String> ret, Object val);
}
```

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- A split pail to vertically partition the dataset
  - FieldStructure usage for vertical partitioning of the table

```
// methods are from SplitDataPailStructure

public List<String> getTarget(Data object) {
  List<String> ret = new ArrayList<String>();
  DataUnit du = object.get_dataunit();
  short id = du.getSetField().getThriftFieldId();
  ret.add("" + id);
  validFieldMap.get(id).fillTarget(ret, du.getFieldValue());
  return ret;
}
```

The top-level directory is determined by inspecting the DataUnit.

Any further partitioning is passed to the FieldStructure.

```
public boolean isValidTarget(String[] dirs) {
  if(dirs.length==0) return false;
  try {
    short id = Short.parseShort(dirs[0]);
    FieldStructure s = validFieldMap.get(id);
    if(s==null)
      return false;
    else
      return s.isValidTarget(dirs);
  } catch(NumberFormatException e) {
    return false;
  }
}
```

The validity check first verifies the DataUnit field ID is in the field map.

Any additional checks are passed to the FieldStructure.

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- A split pail to vertically partition the dataset
  - EdgeStructure class is trivial

```
protected static class EdgeStructure implements FieldStructure {
  public boolean isValidTarget(String[] dirs) { return true; }
  public void fillTarget(List<String> ret, Object val) { }
}
```

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- A split pail to vertically partition the dataset
  - The PropertyStructure class

```
protected static class PropertyStructure implements FieldStructure {
  private TFieldIdEnum valueId;
  private HashSet<Short> validIds;

  public PropertyStructure(Class prop) {
    try {
      Map<TFieldIdEnum, FieldMetaData> propMeta = getMetadataMap(prop);
      Class valClass = Class.forName(prop.getName() + "Value");
      valueId = getIdForClass(propMeta, valClass);

      validIds = new HashSet<Short>();
      Map<TFieldIdEnum, FieldMetaData> valMeta
        = getMetadataMap(valClass);
      for(TFieldIdEnum valId: valMeta.keySet()) {
        validIds.add(valId.getThriftFieldId());
      }
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
```

The set of Thrift IDs of the property value types (annotating `private HashSet<Short> validIds;`)

A Property is a Thrift struct containing a property value field; this is the ID for that field. (annotating `private TFieldIdEnum valueId;`)

Parses the Thrift metadata to get the field ID of the property value (annotating `valueId = getIdForClass(propMeta, valClass);`)

Parses the metadata to get all valid field IDs of the property value (annotating `validIds.add(valId.getThriftFieldId());`)

STORING THE MASTER DATASET FOR SUPERWEBANALYTICS.COM

- A split pail to vertically partition the dataset
  - The PropertyStructure class

```
public boolean isValidTarget(String[] dirs) {
  if(dirs.length < 2) return false;
  try {
    short s = Short.parseShort(dirs[1]);
    return validIds.contains(s);
  } catch(NumberFormatException e) {
    return false;
  }
}
```

The vertical partitioning of a property value has a depth of at least two.

```
public void fillTarget(List<String> ret, Object val) {
  ret.add("" + ((TUnion) ((TBase)val)
    .getFieldValue(valueId))
    .getSetField()
    .getThriftFieldId());
  }
}
```

Uses the Thrift IDs to create the directory path for the current fact

```
private static Map<TFieldIdEnum, FieldMetaData>
  getMetadataMap(Class c)
{
  try {
    Object o = c.newInstance();
    return (Map) c.getField("metaDataMap").get(o);
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

getMetadataMap and getIdForClass are helper functions for inspecting Thrift objects.

```
private static TFieldIdEnum getIdForClass(
  Map<TFieldIdEnum, FieldMetaData> meta, Class toFind)
{
  for(TFieldIdEnum k: meta.keySet()) {
    FieldValueMetaData md = meta.get(k).valueMetaData;
    if(md instanceof StructMetaData) {
      if(toFind.equals(((StructMetaData) md).structClass)) {
        return k;
      }
    }
  }

  throw new RuntimeException("Could not find " + toFind.toString() +
    " in " + meta.toString());
}
```