Sentiment Analysis

- Sentiment means “a general thought, view, feeling, emotion, opinion, or sense,” and Wikipedia describes sentiment analysis (also known as opinion mining) as “the use of natural language processing, text analysis, and computational linguistics to identify and extract subjective information in source materials.”
- Bo Pang and Lillian Lee [https://arxiv.org/abs/cs/0409058] wrote that “sentiment analysis seeks to identify the viewpoint(s) underlying a text span; an example application is classifying a movie review as thumbs up or thumbs down.”
- To perform a sentiment analysis about some event, we need to teach computers what a sentiment is (i.e., how to define “positive” or “negative” and “good” or “bad”).
- This is where machine learning comes in: we must teach computers the meaning of positive, negative, and so on.

Sentiment Analysis

- The first step in this process is to build a model from a set of training data.
- After the model is built, we will use it to analyze new data.
- So what is sentiment data?
  - Typically, it is unstructured data that represents opinions and emotions contained in sources such as special news bulletins, customer support emails, social media posts (such as tweets and Facebook comments), and online product reviews.

Sentiment Analysis

- To perform a good sentiment analysis, the sentiment analysis engine has to conduct some level of speech analysis and word-sense disambiguation.
- Therefore, a sentiment analysis of a text document involves more than tokenizing words and checking them against a list of “positive” and “negative” words.
- Sometimes, we need to understand the intensity of words, and account for factors like negation and diminishers.
- For example, consider the following sentence:
  The movie was not good.

Sentiment Analysis

- "The movie was not good."
- If you just look at the individual words without considering their relationships, you might decide that the sentiment is "neutral" (since not is negative and good is positive).
- But if you look at the semantics of the whole sentence, clearly it is a negative sentiment.
- Therefore, the objective of sentiment analysis is to understand opinions expressed about some subject and distill them into discrete categories such has happy, sad, and angry or, more simply, positive, negative, and neutral.

Sentiment Examples

The following are some examples of sentiment analysis applications:

- What are bloggers saying about a brand like Toyota after a recall of the braking system in some Toyota cars?
- What is the sentiment of viewers of *Iron Man 3* before and after seeing the movie? (We can answer this by analyzing tweets before and after viewing.) Did people really like the movie?
- What is the sentiment of customers before and after the announcement of a new iPhone?
- What is the sentiment of voters before and after a presidential debate? Is there a certain sentiment toward the Democratic or Republican party?
- What is the sentiment of consumers about a customer service experience (happy or sad)? Here, for sentiment analysis, we need customer service log data and a dictionary of "happy" and "sad" words.

Sentiment Scores: Positive or Negative

Given a short sentence (such as a tweet, which is 140 characters or fewer), how do you determine whether it expresses a positive or negative sentiment? To score syntactically (by ignoring contextual semantics), you can tokenize the sentence into words and then check these words against a list of positive and negative words.[1]

For example, given the following sentences (**bold** words are positive and underlined words are negative):

- Sentence 1: "The movie was **great** and I **loved** it."
- Sentence 2: "The hamburger had a bad taste and was terrible, but I **loved** the fries."

what are the sentiment scores of sentence 1 and sentence 2?

```
sentimentScore(Sentence-1) = 1 positive +
                             1 positive
                           = 2 positives (Positive sentiment)

sentimentScore(Sentence-2) = 1 negative +
                             1 negative +
                             1 positive
                           = 1 negative (Negative sentiment)
```

In most sentiment analysis situations, checking/examining the individual words against a list of positive and negative words is not enough and might not yield the expected results. In real sentiment analysis, it is better not to rely simply on pure syntax, but rather to understand the semantics — the intensity of words, and factors like negation and diminishers. For example, consider the following sentence:

"The movie was not **great** and I did not **love** it."

In a real-world sentiment analysis, checking pure syntax here will not yield accurate results. As noted earlier, if you just look at individual words without their relationships, your pure syntactical algorithm might indicate that the sentiment is "neutral" (since *not* is negative and *great* and *love* are positive). But when you look at the semantics of the whole sentence, clearly it is a negative sentiment.

So, in a nutshell, we can say that *sentiment analysis* is the task of determining whether the statement/opinion expressed in text is positive, negative, or neutral toward a particular topic or trend.

A Simple MapReduce Sentiment Analysis Example

Now let's go over an example for performing sentiment analysis on a given set of tweets. Let our keywords of interest be *Obama* and *Romney*, and further assume that it is an election year and you want to know the sentiment of people tweeting about these two presidential candidates. Let's assume that you want to find the sentiment trend for each candidate per day. The mapper accepts a tweet, normalizes the text, looks for a keyword of interest (*Obama* or *Romney*), counts the positive and negative keywords, and then subtracts the ratio of negative words from the ratio of positive words. To perform mapping, we need two distinct sets of words:

- A set of positive words (such as *good*, *like*, and *enjoy*)
- A set of negative words (such as *hate*, *bad*, and *terrible*)

These two sets are passed to mappers by the driver program. In Hadoop we can accomplish this by using a distributed cache (the `DistributedCache`[2] class).

map() Function for Sentiment Analysis

- We assume that there exists a normalizer and a tokenizer function for a given tweet, as shown:

```
private static List<String> normalizeAndTokenize(String tweet) {
    List<String> tokens = <normalize and tokenize all
                           the words in the tweet text>;
    return tokens;
}
```

- Using the `normalizeAndTokenize()` method we can complete the `map()` method, as shown in the next slide.

```
private Set<String> positiveWords = null;
private Set<String> negativeWords = null;
// default candidates values
private Set<String> allCandidates = {"obama", "romney"};

setup() {
   positiveWords = <load positive words from distributed cache>;
   negativeWords = <load negative words from distributed cache>;
   allCandidates = <load all candidates from distributed cache>;
}

/**
 * @param key is the date of a tweet: YYYY-MM-DD:hh:mm:ss
 * @param value is a single tweet
 */
map(key, value) {
    date = key; // date of a tweet as YYYY-MM-DD
    List<String> tweetWords = normalizeAndTokenize(value);
    int positiveCount = 0;
    int negativeCount = 0;
    for (String candidate : allCandidates) {
        if (candidate is in the tweetWords) {
            int positiveCount = <count of positive words in tweetWords>;
            int negativeCount = <count of negative words in tweetWords>;
            double positiveRatio = positiveCount / tweetWords.size();
            double negativeRatio = negativeCount / tweetWords.size();
            outputKey = Pair(date, candidate);
            outputValue = positiveRatio - negativeRatio;
            emit (outputKey, outputValue);
        }
    }
}
```

reduce() Function for Sentiment Analysis


```
/**
 * @param key is the Pair(Date, String)
 *    where
 *        Date = YYYY-MM-DD
 *        String = a candidate ("obama" or "romney")
 * @param values is a List<Double> where Double represents a ratio
 */
reduce(key, values) {
   double sumOfRatio = 0.0;
   int n = 0; // number ratios
   for (Double ratio : values) {
      n++;
      sumOfRatio += ratio;
   }
   emit (key, sumOfRatio / n);
}
```

Sentiment Analysis in the Real World

- The algorithms described in this lecture are very simplistic and trivial (they can be applied only to the simplest problems).
- Implementing a realistic context-based sentiment analysis is a very complex process, and understanding the data requires a lot of attention.
- If you're interested in learning more, Siddharth Batra and Deepak Rao describe a realistic entity-based sentiment analysis on Twitter that can be implemented in a MapReduce paradigm.
- Hu and Liu have published an "opinion lexicon" that categorizes approximately 6,800 words as positive or negative and can be downloaded from
  http://bit.ly/opinion_lexicon
  https://www.cs.uic.edu/~liub/FBS/opinion-lexicon-English.rar

- For example, positive words include love, best, cool, great, and good, while negative words include terrible, bad, hate, worst, and sucks.

- DistributedCache (org.apache.hadoop.filecache.DistributedCache) is a facility provided by Hadoop's MapReduce framework to cache text and archive files needed by applications.

Huge Cache for MapReduce

- We will discuss how to use and read a huge cache (i.e., composed of billions of key-value pairs that cannot fit in a commodity server's memory) in MapReduce algorithms.
- The algorithms presented in this lecture are generic enough to be used in any MapReduce paradigms (such as MapReduce/Hadoop and Spark).
- There are some MapReduce algorithms that might require access to some huge (i.e., containing billions of records) static reference relational tables.
- Typically, these reference relational tables do not change for a long period of time, but they are needed in either the map() or reduce() phase of MapReduce programs.

Example

- One example of such a table is a “position feature” table, which is used for *germline data* type ingestion and variant classification.
- The position feature table might have the attributes shown below (a composite key is (chromosome_id, position).

Column name	Characteristics
chromosome_id	Key-1
position	Key-2
feature_id	Basic attribute
mrna_feature_id	Basic attribute
sequence_data_type_id	Basic attribute
mapping	Basic attribute

Cont'd

- Germline refers to the sequence of cells in the line of direct descent from zygote to gametes, as opposed to somatic cells (all other body cells).

- Mutations in germline cells are transmitted to offspring; those in somatic cells are not.

http://medical-dictionary.thefreedictionary.com/germline

Cont'd

- In expressing your solution in the MapReduce paradigm, either in `map()` or `reduce()`, given a `key=(chromosome_id, position)`, you want to return a `List<String>` where each element of the list comprises the remaining attributes

  {feature_id, mrna_feature_id, sequence_data_type_id, mapping}.

- For the germline data type, a position feature table can have up to 12 billion records (which might take about 2 TB of disk space in a MySQL or Oracle DB).

- Now imagine that your mapper or reducer wants to access the position feature table for a given `key=(chromosome_id, position)`.

- Since you will be firing many requests (several million per second) of this type, this will bring your database server to its knees and it will not scale.

Cont'd

- One possible solution is to cache all static data into a hash table and then use the hash table instead of a relational table.

- However, as you will see, this is not a proper solution at all and will not scale — the size of such a hash table will be over 4 TB (a hash table's metadata takes quite a bit of space) and it will not fit in the memory of today's commodity servers.

Implementation Options

- So what are the optimal and pragmatic options for caching 12 billion records in a MapReduce environment?
- We will present several options and discuss the feasibility of their implementations:

Option #1

- Use a relational database (such as MySQL or Oracle).
- This option is not a viable solution and will not scale at all.
- A mapper or reducer will constantly hit the database, and the database server will not be able to handle thousands or millions of requests per second.
- Even if we use a set of replicated relational databases, this option still will not scale out (since the number of database connections is limited for a large set of Hadoop clusters).

Option #2

- Use a memcached server.
- Memcached is an in-memory key-value store for small chunks of arbitrary data (e.g., strings, objects) from the results of database calls, API calls, or page rendering.
- If for every few slave nodes you have a dedicated memcached cluster (which would be very costly in a large cluster environment), then this is a viable and proper option.
- This solution will not scale out due to the high cost of having so many expensive memcached servers.

Option #3

- Use a Redis server.
- Redis is an open source, BSD licensed, advanced key-value store.
- It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, and sorted sets.
- As with memcached, if for every few slave nodes you have a dedicated Redis server (which would be very costly in a large cluster environment), this is a viable and proper option.
- This solution will not scale out due to the high cost of having so many Redis servers (for 2 TB of key-value pairs, Redis requires at least 12 TB of RAM).

Option #4

- Use a MapReduce join between your input and 12 billion static records.
- The idea is to flatten the 12 billion records and then use a MapReduce join between this and your input.
- Perform the join on the `key=(chromosome_id, position)`.
- This is a viable option if your input is huge too (meaning billions rather than millions).
- For example, for a germline ingestion process, the VCF file will not have more than 6 million records.
- It is not appropriate to join a table of 6 million records against 12 billion records (as it wastes lots of time).

VCF File

- Variant call format is a text file format.
- It contains metainformation lines, a header line, and data lines, each containing information about a position in the genome.
- For more details, see
  http://bit.ly/variant_call_format

Option #5

- Partition the 12 billion static records into small chunks (each chunk will be 64 MB, which you can easily load into RAM and evict when it's not needed) and use an LRU-Map (LRU stands for least recently used) hash table.
- This simple but elegant idea works very well: it scales out and does not require dedicated cache servers.
- This solution is detailed in the following slides.
- No matter what, in a distributed environment the mapper or reducer does not have access to an unlimited amount of memory/RAM.
- This solution will work in any MapReduce environment and does not require extra RAM for caching purposes.
- The LRU map is a local cache on a Linux filesystem, and every cluster worker node will have an identical copy of it (which occupies less than 1 TB of hard disk space).
- Using solid disk drives (SSDs) improves the cache performance by 8 to 10 times over regular disk drives.

Formalizing the Cache Problem

- In a nutshell, given a set of 12 billion records, our goal is to find an associated value for a given composite key

```
key=(chromosome_id, position)
```

- Let's say that you are going to ingest a VCF file (this is our input file, comprising about 5 million records for the germline data type) into your genome system.

- For every record of a given VCF, you need to find out:
  - Mutation class
  - List of genes
  - List of features

Reading VCF Records

- To find this detailed information for each VCF record, you need a set of static tables such as the position feature table outlined at the beginning of this lecture.

- As we discussed before, a position feature table may contain as many as 12 billion records.

- The first step of the germline ingestion process is to find records from a position feature table for a given

  ```
  key=(chromosome_id, position)
  ```

  (every VCF record will have *chromosome_id* and *position* fields)

An Elegant, Scalable Solution

- The solution presented here is a local cache solution.
- This means that every worker/slave node of a MapReduce/Hadoop or Spark cluster will have its own local cache (on a hard disk, which is cheap enough), and there will not be any network traffic for accessing cache components and data.
- The local cache will reside in an SSD or HDD and be brought to memory on an as-needed basis via the LRU map caching technique.
- Assuming that we have a relational table called position_feature (as defined in the beginning) that has over 12 billion records, the following slides show how this works:

STEP 1

- First, we partition the 12 billion records by `chromosome_id` (1, 2, 3, ..., 24, 25).
- This will give us 25 files (let's label these files *chr1.txt, chr2.txt, ..., chr25.txt*), where each record has the following format (note key=(chromosome_id, position) returns multiple records from the relational table):
  <position><;><Record1><:><Record2><:>...<:><RecordN>
- and each Record<i> comprises the following:
  <feature_id><,><mrna_feature_id><,><sequence_data_type_id><,><mapping>
- Therefore, each raw cache data file (*chr1.txt, chr2.txt, ..., chr25.txt*) corresponds to the following SQL query (you may repeat this script for each `chromosome_id`)

```
select position,
       GROUP_CONCAT(feature_id, ',',
                    mrna_feature_id, ',',
                    seq_datatype_id, ',',
                    mapping SEPARATOR ':')
       INTO OUTFILE '/tmp/chr1.txt'
       FIELDS TERMINATED BY ';'
       LINES TERMINATED BY '\n'
   from position_feature where chromosome_id = 1 group by position;
```

STEP 2

- Next, we sort each of these files (*chr1.txt, chr2.txt, ..., chr25.txt*) by position and generate *chr1.txt.sorted, chr2.txt.sorted, ..., chr25.txt.sorted*.

STEP 3

- Since memory is limited (say, to 4 GB) for each mapper/reducer of a MapReduce job, we partition each sorted file into chunks of 64 MB (without breaking any lines).

- To do so, we execute the following command:

```
#!/bin/bash
sorted=/data/positionfeature.sorted
output=/data/partitioned/
for i in {1..25} ; do
  echo "i=$i"
  mkdir -p $output/$i
  cd $output/$i/
  split -a 3 -d -C 64m $sorted/$i.txt.sorted $i.
done
exit
```

For example, for `chromosome_id=1` we will have:

```
# ls -l /data/partitioned/1/
-rw-rw-r-- 1 hadoop hadoop 67108634 Feb 2 09:30 1.000
-rw-rw-r-- 1 hadoop hadoop 67108600 Feb 2 09:30 1.001
-rw-rw-r-- 1 hadoop hadoop 67108689 Feb 2 09:30 1.002
...
-rw-rw-r-- 1 hadoop hadoop 11645141 Feb 2 09:33 1.292
```

Note that each of these partitioned files has a range of `position` values (since they are sorted by `position`). We will use these ranges in our cache implementation. Therefore, given a `chromosome_id=1` and a `position`, we know exactly which partition holds the result of a query. Let's look at the content of one of these sorted partitioned files:

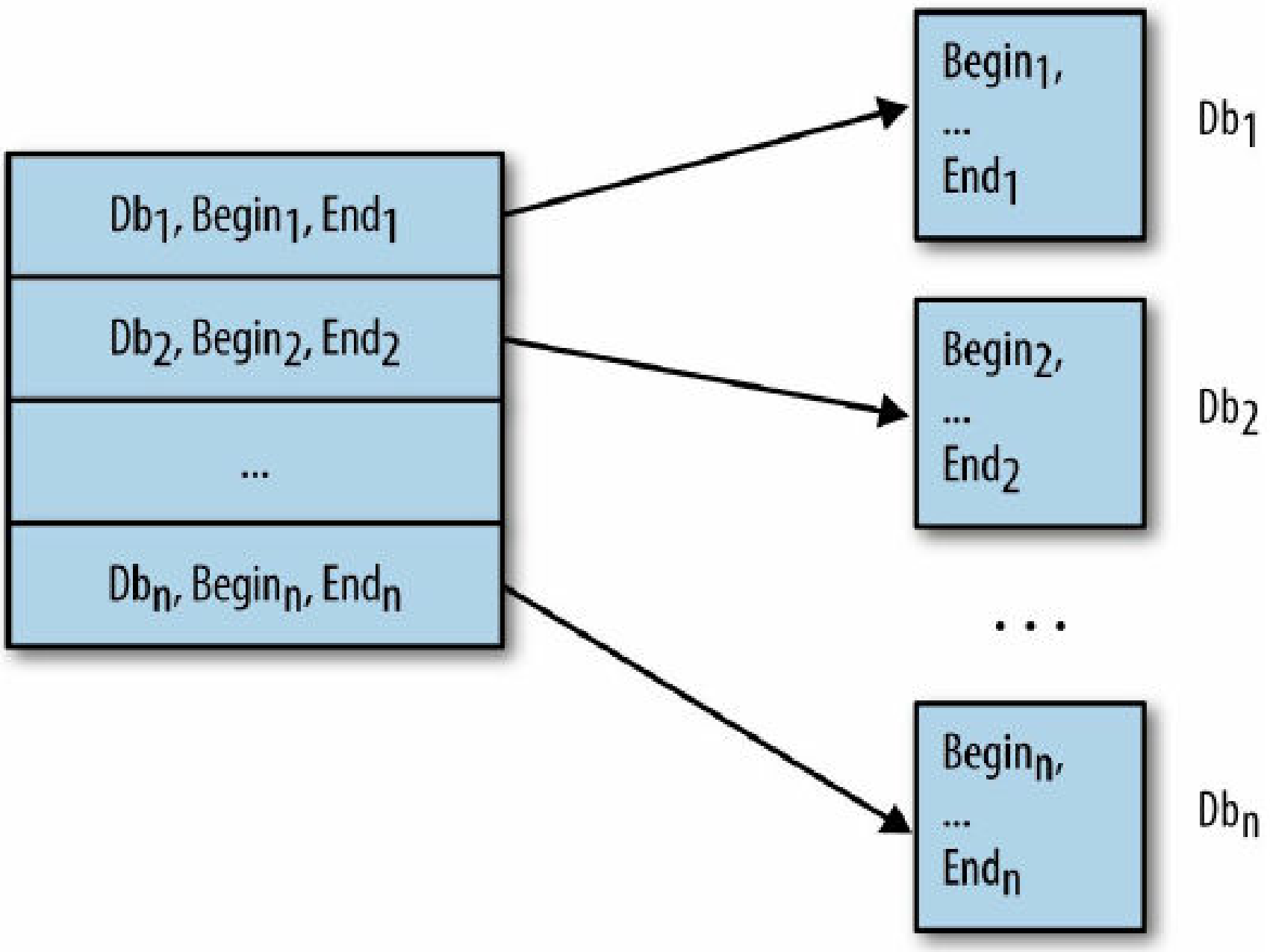
```
# head -2 /data/partitioned/1/1.000
6869;35304872,35275845,2,1
6870;35304872,35275845,2,1

# tail -2 /data/partitioned/1/1.000
790279;115457,21895578,12,2:115457,35079912,3,3:...
790280;115457,21895578,12,2:115457,35079912,3,3:...
```

You can see that all positions are sorted within each partition. To support metadata for all partitions using LRU Map, we need an additional data structure to keep track of (`begin`, `end`) positions. For each partitioned file we will keep the (`partition name`, `begin`, `end`) information. For example, for `chromosome id=1` and `chromosome id=2` we get:

```
# cat 1/begin_end_position.txt
1.000;6869;790280
1.001;790281;1209371
1.002;1209372;1461090
...
1.292;249146130;249236242

# cat 2/begin_end_position.txt
2.000;33814;1010683
2.001;1010684;1494487
2.002;1494488;2132388
...
2.279;242617420;243107469
```



Huge cache partition data structure

STEP 4

- Next, we will convert each partition (of 64 MB) into a hash table, implemented by MapDB.
- Since memory is limited for each mapper/reducer, we use LRUMap<K,V>(N) to hold at most N MapDB data structures (each MapDB-persistent map corresponds to one sorted partition of 64 MB.
- The idea of LRUMap<K,V> is to hold at most N partitions such that N × 64 MB will be smaller than the memory available for each mapper/reducer.
- When you insert the N + 1st entry into LRUMap<K,V>(N), the oldest entry will be evicted (then you can properly close the MapDB object, which releases all memory and closes all file handles).
- For the LRU Map implementation, we use
  org.apache.commons.collections4.map.LRUMap<K,V>

MapDB

- MapDB is implemented by Jan Kotek and its source code is hosted at https://github.com/jankotek/MapDB
- MapDB provides concurrent maps, sets, and queues backed by disk storage or off-heap memory.
- It is a fast and easy-to-use embedded Java database engine.

STEP 5

- The final step is to sort our input (which will use the cache; note that sorting input such as a VCF file can be done in under 3 seconds) by key=(chromosome_id, position).
- This sorting will minimize the eviction of MapDB entries from LRUMap<K,V>(N).
- The sorting of an input file is a huge design/implementation criterion and will have a big impact on the performance of our MapReduce job (germline ingestion).
- If we don't sort our input file, the eviction rate might be very high.
- Conversely, sorting the input file will minimize the eviction rate.
- Note that sorting the input file is very fast; it should not take more than a few seconds (you can do so through a Linux sort command or by using MapReduce/Hadoop).

Implementing the LRUMap Cache

- We willwalk through how to implement our elegant, scalable solution to managing a huge cache size in MapReduce.

Extending the LRUMap Class

- To implement the LRU map, I selected LRUMap<K,V>, which is a Map implementation with a fixed maximum size that removes the least recently used (LRU) entry if an entry is added when the map is full.
  [org.apache.commons.collections4.map.LRUMap]

- The LRU map algorithm works on the get and put operations only.

- Iteration of any kind, including setting the value by iteration, does not change the order of entries (for details, you may consult the Apache Commons Collections).

- For our cache implementation, we extend the LRUMap class (the MapDBEntry class is a simple class that represents a sorted partition of 64 MB as a Map data structure implemented in MapDB).

```java
import org.apache.commons.collections4.map.LRUMap;
import org.apache.commons.collections4.map.AbstractLinkedMap.LinkEntry;

public class CustomLRUMap<K, V> extends LRUMap<K, V> {

    private K key = null;
    private V value = null;
    private LinkEntry<K, V> entry = null;

    public CustomLRUMap(final int size) {
        super(size);
    }

    @Override
    protected boolean removeLRU(final LinkEntry<K, V> entry) {
        System.out.println("begin remove LRU entry ...");
        this.entry = entry;
        this.key = entry.getKey();
        this.value = entry.getValue();

        if (key instanceof String) {
            String keyAsString = (String) key;
            System.out.println("evicting key="+keyAsString);
        }

        if (value instanceof MapDBEntry) {
            // release resources held by MapDBEntry
            MapDBEntry mapdbEntry = (MapDBEntry) value;
            mapdbEntry.close();
        }

        return true; // actually delete entry from the LRU map
    }
}
```

Testing the Custom Class

- Following Example hows how CustomLRUMap<K,V> works.
- The CustomLRUMap<K,V> class extends the LRUMap<K,V> class and redefines the eviction policy by the removeLRU() method.
- In this example, we keep only three map entries at any time. No matter how many entries you add to the LRUMap object, the LRUMap.size() cannot exceed 3 (the maximum size is given at the time of the CustomLRUMap<K,V> object's creation).

```
# cat CustomLRUMapTest.java

import org.apache.commons.collections4.map.LRUMap;

public class CustomLRUMapTest {
   public static void main(String[] args) throws Exception {
      CustomLRUMap<String, String> map = new CustomLRUMap<String, String>(3);
      map.put("k1", "v1");
      map.put("k2", "v2");
      map.put("k3", "v3");
      System.out.println("map="+map);
      map.put("k4", "v4");
      String v = map.get("k2");
      System.out.println("v="+v);
      System.out.println("map="+map);
      map.put("k5", "v5");
      System.out.println("map="+map);
      map.put("k6", "v6");
      System.out.println("map="+map);
   }
}
```

Running this test produces the following output:


```
# javac CustomLRUMapTest.java
# java CustomLRUMapTest
map={k1=v1, k2=v2, k3=v3}
begin removeLRU...
evicting key=k1
v=v2
map={k3=v3, k4=v4, k2=v2}
begin removeLRU...
evicting key=k3
map={k4=v4, k2=v2, k5=v5}
begin removeLRU...
evicting key=k4
map={k2=v2, k5=v5, k6=v6}
```

Using MapDB

- How do we create a persistent Map<K,V> using MapDB?
- Example in the next slide shows how to create a MapDB for a sorted partition file using the GenerateMapDB class.

```java
import org.mapdb.DB;
import org.mapdb.DBMaker;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.File;
import java.util.concurrent.ConcurrentNavigableMap;

public class GenerateMapDB {

    public static void main(String[] args) throws Exception {
        String inputFileName = args[0];
        String mapdbName = args[1];
        create(inputFileName, mapdbName);
    }

    public static void create(String inputFileName, String mapdbName)
        throws Exception {
        // Configure and open database using builder pattern.
        // All options are available with code autocompletion.
        DB db = DBMaker.newFileDB(new File(mapdbName))
                .closeOnJvmShutdown()
                .make();

        // Open a collection. TreeMap has better performance than HashMap.
        ConcurrentNavigableMap<String,String> map =
           db.getTreeMap("collectionName");

        //
        // line = <position><;><v><:><v>:...:<v>
        //  where <v> has the following format:
        //  <feature_id><,><mrna_feature_id><,><sequence_data_type_id><,><mapping>
        //
        String line = null;
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new FileReader(inputFileName));
            while ((line = reader.readLine()) != null) {
                line = line.trim();
                String[] tokens = line.split(";");
                if (tokens.length == 2) {
                    map.put(tokens[0], tokens[1]);
                }
                else {
                    System.out.println("error line="+line);
                }
            }
        }
        finally {
            reader.close(); // close input file
            db.commit(); // persist changes to disk
            db.close(); // close database resources
        }
    }
}
```

QueryMapDB Class

- To query the MapDB database, we can write a very simple class, QueryMapDB, as shown in the next slide.

```java
import java.io.File;
import java.util.Map;
import org.mapdb.DB;
import org.mapdb.DBMaker;
import org.apache.log4j.Logger;
/**
 * This class defines a basic query on MapDB.
 *
 * @author Mahmoud Parsian
 *
 */
public class QueryMapDB {

    public static void main(String[] args) throws Exception {
        long beginTime = System.currentTimeMillis();
        String mapdbName = args[0];
        String position = args[1];
        THE_LOGGER.info("mapdbName="+mapdbName);
        THE_LOGGER.info("position="+position);
        String value = query(mapdbName, position);
        THE_LOGGER.info("value="+value);
        long elapsedTime = System.currentTimeMillis() -beginTime;
        THE_LOGGER.info("elapsedTime (in millis) ="+elapsedTime);
        System.exit(0);
    }

    public static String query(String mapdbName, String key) throws Exception {
        String value = null;
        DB db = null;
        try {
            db = DBMaker.newFileDB(new File(mapdbName))
                .closeOnJvmShutdown()
                .readOnly()
                .make();
            Map<String, String> map = map = db.getTreeMap("collectionName");
            value = map.get(key);
        }
        finally {
            if (db != null) {
                db.close();
            }
        }
        return value;
    }
}
```

Testing MapDB: put()

The following code segments show how to create MapDB entries:

```
# javac GenerateMapDB.java
# javac QueryMapDB.java

# cat test.txt
19105201;35302633,35292056,2,1:20773813,35399339,2,1
19105202;35302633,35292056,2,1:20773813,35399339,2,1
19105203;35302633,35284930,2,1:35302633,35292056,2,1:20773813,35399339,2,1
19105204;35302633,35284930,2,1:35302633,35292056,2,1:20773813,35399339,2,1

# java GenerateMapDB test.txt mapdbtest
# ls -l mapdbtest*
-rw-r--r--  1 mahmoud  staff  32984 Feb 11 16:40 mapdbtest
-rw-r--r--  1 mahmoud  staff  13776 Feb 11 16:40 mapdbtest.p
```

As you can see from the generated output, MapDB generates two files (*mapdbtest* and *mapdbtest.p*) for each persistent hash table.

Testing MapDB: get()

The following code segments show how to query MapDB entries:

```
# java QueryMapDB mapdbtest 19105201
16:41:16 [QueryMapDB] - mapdbName=mapdbtest
16:41:16 [QueryMapDB] - position=19105201
16:41:16 [QueryMapDB] - value=35302633,35292056,2,1:20773813,35399339,2,1
16:41:16 [QueryMapDB] - elapsedTime (in millis) = 2

# java QueryMapDB mapdbtest 19105202
16:41:21 [QueryMapDB] - mapdbName=mapdbtest
16:41:21 [QueryMapDB] - position=19105202
16:41:21 [QueryMapDB] - value=35302633,35292056,2,1:20773813,35399339,2,1
16:41:21 [QueryMapDB] - elapsedTime (in millis) = 2

# java QueryMapDB mapdbtest 191052023333
16:41:55 [QueryMapDB] - mapdbName=mapdbtest
16:41:55 [QueryMapDB] - position=191052023333
16:41:55 [QueryMapDB] - value=null
16:41:55 [QueryMapDB] - elapsedTime (in millis) = 1
```

MapReduce Using the LRUMap Cache

Now that we have an efficient LRU map cache, we may use it in either the `map()` or `reduce()` function. To use the `LRUMap` cache in `map()` or the `reduce()`, you need to do the following (this can be applied in the mapper or reducer class):

1. Initialize and set up the `LRUMap` cache. This step can be accomplished in the `setup()` method. This will be done once.

2. Use the `LRUMap` cache in `map()` or `reduce()`. This will be done many times.

3. Close the `LRUMap` cache objects (to release all unnecessary resources). This step can be accomplished in the `cleanup()` method. This will be done once.

To make these steps easier, we define a service class called `CacheManager`, which can be used as shown

CacheManager usage

```
 1 try {
 2       //
 3       // initialize cache
 4       //
 5       CacheManager.init();
 6
 7       //
 8       // use cache
 9       //
10       String chrID = ...;
11       String position = ...;
12       List<String> valueAsList = CacheManager.get(chrID, position);
13 }
14 finally {
15       //
16       // close cache
17       //
18       CacheManager.close();
19 }
```

CacheManager Definition

- CacheManager is a service class that provides three basic functionalities: opening the cache, the service (retrieving the desired values), and closing the cache.
- Since it is a service class, all methods are defined as static.
- The BeginEndPosition object implements the partition data structures such that you can get the database name (a 64 MB hash table) for a given key of (chrID, position).

```
public class CacheManager {

   private static final int DEFAULT_LRU_MAP_SIZE = 128;
   private static int theLRUMapSize = DEFAULT_LRU_MAP_SIZE;

   private static CustomLRUMap<String, MapDBEntry<String, String>>
      theCustomLRUMap = null;
   private static BeginEndPosition beginend = null;
   private static String mapdbRootDirName =
      "/cache/mapdb/pf";
   private static String mapdbBeginEndDirName =
      "/cache/mapdb/pf/begin_end_position";
   private static boolean initialized = false;

   public static void setLRUMapSize(int size) {
      theLRUMapSize = size;
   }

   public static int getLRUMapSize() {
      return theLRUMapSize;
   }

   // initialize the LRUMap
   public static void init() throws Exception {...}

   // initialize the LRUMap
   public static void init(int size) throws Exception {...}

   // close the cache database in the LRUMap
   public static void close() throws Exception {...}

   // get value from the cache database for a given (chrID, position).
   public static String get(int chrID, int position) throws Exception {...}

   // get value from the cache database for a given (chrID, position).
   public static String get(String chrID, String position) throws Exception {...}

   // close the cache database in the LRUMap cache
   public static void close() throws Exception {...}
}
```

Initializing the Cache


```
public static void setLRUMapSize(int size) {
    theLRUMapSize = size;
}

public static int getLRUMapSize() {
    return theLRUMapSize;
}

// initialize the LRUMap
public static void init() throws Exception {
    if (initialized) {
        return;
    }
    theCustomLRUMap =
       new CustomLRUMap<String, MapDBEntry<String, String>>(theLRUMapSize);
    beginend = new BeginEndPosition(mapdbBeginEndDirName);
    beginend.build(mapdbRootDirName);
    initialized = true;
}

// initialize the LRUMap
public static void init(int size) throws Exception {
    if (initialized) {
        return;
    }
    setLRUMapSize(size);
    init();
}
```

CacheManager usage

```
public static String get(int chrID, int position) throws Exception {
    return get(String.valueOf(chrID), String.valueOf(position));
}

/**
 * Get value from the cache database (value is the snpID)
 * for a given (chrID, position).
 * @param chrID=chrID
 * @param position=position
 */
public static String get(String chrID, String position) throws Exception {
    String dbName = getDBName(chrID, position);
    if (dbName == null) {
        return null;
    }
    // now return the cache value
    MapDBEntry<String, String< entry = theCustomLRUMap.get(dbName);
    if (entry == null) {
        entry = MapDBEntryFactory.create(dbName);
        theCustomLRUMap.put(dbName, entry);
    }
    return entry.getValue(position);
}

private static String getDBName(String chrID, String position) {
    // query parameters are: chrID and position
    List<Interval> results = beginend.query(chrID, position);
    if ((results == null) || (results.isEmpty()) || (results.size() == 0)) {
        return null;
    }
    else {
        return results.get(0).db();
    }
}
```

CacheManager: closing the cache

```
public static void close() throws Exception {
    if (theCustomLRUMap != null) {
        for (Map.Entry<String, MapDBEntry<String, String>>
            entry : theCustomLRUMap.entrySet()) {
            entry.getValue().close();
        }
    }
}
```

- This lecture presented a scalable huge cache solution that can be used by the map() and reduce() functions.
- This solution is very simple and can be implemented by just using commodity servers.