# DisCo: Distributed Co-clustering with Map-Reduce

**A Case Study Towards Petabyte-Scale End-to-End Mining**

Spiros Papadimitriou Jimeng Sun

IBM T.J. Watson Research Center
Hawthorne, NY, USA
{spapadim,jimeng}@us.ibm.com

## Abstract

*Huge datasets are becoming prevalent; even as researchers, we now routinely have to work with datasets that are up to a few terabytes in size. Interesting real-world applications produce huge volumes of messy data. The mining process involves several steps, starting from pre-processing the raw data to estimating the final models.*

*As data become more abundant, scalable and easy-to-use tools for distributed processing are also emerging. Among those, Map-Reduce has been widely embraced by both academia and industry. In database terms, Map-Reduce is a simple yet powerful execution engine, which can be complemented with other data storage and management components, as necessary.*

*In this paper we describe our experiences and findings in applying Map-Reduce, from raw data to final models, on an important mining task. In particular, we focus on co-clustering, which has been studied in many applications such as text mining, collaborative filtering, bio-informatics, graph mining. We propose the* Distributed Co-clustering *(*DisCo*) framework, which introduces practical approaches for distributed data pre-processing, and co-clustering. We develop* DisCo *using Hadoop, an open source Map-Reduce implementation. We show that* DisCo *can scale well and efficiently process and analyze extremely large datasets (up to several hundreds of gigabytes) on commodity hardware.*

## 1 Introduction

It's a cliché, but it's true: huge volumes of data are collected and need to be processed on a daily basis. For example, Google now processes an estimated 20 petabytes of data per day [13] and the Internet Archive[1] is growing at 20 terabytes a month, having reached 2 petabytes sometime in 2006. Retail giants such as Walmart and online shopping stores such as Amazon and eBay all deal with with petabytes of transactional data every day.

By definition, research on data mining focuses on scalable algorithms applicable to huge datasets. But let's take things from the beginning. Natural sources of data provide them in vast quantities, but impure form. A repository may consist of, e.g., a corpus of text documents, a large web crawl, or system logs. Schemas do not arise spontaneously in nature. On the contrary, significant effort must be invested to make the data fit a given schema. Most commonly, data are collected in a multitude of unstructured or semi-structured formats. Aspects of the data that are relevant to the task at hand need to be extracted and stored in an appropriate representation. Most researchers start with the assumption that the input is in the appropriate form. However, getting the data into the right form is not trivial (see detailed discussion in Section 3).

Map-Reduce [12] is attracting a lot of attention, proving both a source for inspiration [30] as well as target of polemic [14] by prominent researchers in databases. Recently, some have questioned whether relational DBMSes are appropriate for any and all data management tasks under the sun [35, 34]. Moreover, [34] makes a strong case that bundling data storage, indexing, query execution, transaction control, and logging components into a monolithic system with a veneer of SQL is not always desirable. Starting from this call for a component-based approach, Map-Reduce is an execution engine, largely unconcerned about data models and storage schemes. In the simplest case, data reside on a distributed file system [19, 1, 26] but nothing prevents pulling data from a large data store like BigTable [7, 2, 38], or any other storage engine that (i) provides data de-clustering and replication across many machines, and (ii) allows computations to execute on local copies of the data. Arguably, Map-Reduce is powerful both for the features it provides, as well as for the features it omits, in order to provide a clean and simple programming abstraction.

Hadoop is an open source implementation of the core components necessary for Map-Reduce. It focuses on providing the necessary minimum functionality, combining simplicity of use with scalable performance[2]. However, if additional functionality is needed by an application, other open source components are available, which address e.g., key-based data access [2], or more complex job and data

[1]`http://www.archive.org/`

[2]While this article was being written, Hadoop won the TeraSort benchmark in the general purpose category, completing the task in 209 seconds using 900 eight-core nodes, beating the previous record of 297 seconds.

schema management [37, 3].

In the context we have so far described, this paper describes our experiences and findings in applying the above end-to-end philosophy and tools to a particular problem. More specifically, we focus on co-clustering or bi-clustering [24, 8] of pairwise relationships extracted from the raw data. The natural format for the relevant data features, i.e., the graph of associations between different entities, is a sparse adjacency matrix representation. Co-clustering provides a general set of tools to simultaneously cluster both rows and columns into groups, based on certain criteria. Unlike clustering which groups similar rows or columns independently, co-clustering searches for sub-matrices of rows and columns that are inter-related. Co-clustering has been studied in many different applications including text mining [15, 28], bioinformatics [24, 8], recommendation systems [18], and graph mining [6].

Powerful as it is, co-clustering is not practical to apply on large matrices (e.g., several millions of rows and columns). This paper proposes a comprehensive Distributed Co-clustering (DisCo) solution from the raw data to the end clusters. In particular, we leverage the highly successful Map-Reduce [13] both as a programming model and as an implementation testbed. More specifically, we develop DisCo using Hadoop [1], an open source package which includes a freely available implementation of Map-Reduce and has been widely embraced by both commercial and academic worlds. DisCo is a scalable framework under which various co-clustering algorithms can be implemented. Since both data pre-processing (i.e., graph extraction) and co-clustering components need efficient sequential scans over the entire data set, we only need to use the core Hadoop components.

The contributions of this paper are:

- We present a pragmatic data mining process that involves data gathering, pre-processing, analysis, and presentation.
- We design a complete distributed co-clustering solution using Hadoop.
- We demonstrate its scalability and power on mining extremely large datasets.

As is often the case, none of the individual steps is surprising in and of itself. However, we believe that the entire data mining process needs to be studied under the currently available tools for large-scale data processing. This paper illustrates our experiences, insights as well as common patterns on using Map-Reduce (Hadoop) for data mining, from the very beginning to the very end and aims to clarify some common misconceptions.

The rest of the paper is organized as follows: Section 2 provides a very brief tutorial introduction of Map-Reduce, as well as the key components it relies on. Section 3 presents a distributed data mining framework. Section 4 presents our design for distributed co-clustering using Map-Reduce and Section 5 evaluates its scalability and presents results on real-world data sets. Section 6 briefly reviews related work, both in systems as well as data mining. Finally, Section 7 concludes.

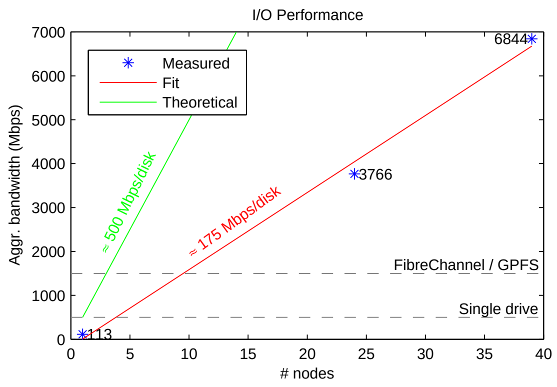


**Figure 1. Scalability for data pre-processing: processing a 350GB logfile takes about 7 minutes on 39 nodes.**

## 2 Background: Map-Reduce

Map-Reduce, originally described in [12], is a core component in an emerging ecosystem of distributed, scalable, fault-tolerant data storage, management, and processing tools [19, 7, 2, 3, 37, 26, 38, 31]. Map-Reduce is essentially a distributed grep-sort-aggregate or, in database terminology, a distributed execution engine for select-project via sequential scan, followed by hash partitioning and sort-merge group-by. It is ideally suited for data already stored on a distributed file system which offers data replication as well as the ability to execute computations locally on each data node. There are two important aspects in Map-Reduce, the programming model and the distributed execution framework. We examine those next, after first introducing a simple example we shall use.

### 2.1 Example scenario

Assume that we have network router log data, consisting of text lines such as

```
2008-02-25 13:55:07 SrcIP=129.34.20.19,DstIP=128.2.207.18,\
    Proto=TCP,SrcPort=67537,DstPort=22,...
2008-02-25 13:56:08 SrcIP=129.34.20.23,DstIP=128.2.209.1,\
    Proto=TCP,SrcPort=52391,DstPort=22,...
...
```

and we wish to extract an adjacency list of source-destination IP pairs. We could achieve this with the following snippet in Python:

```
import re   # Regular expression module
def ip_mapper (line):
  srcip = re.search('SrcIP=(.*?),', line).group(1)
  dstip = re.search('DstIP=(.*?),', line).group(1)
  return (srcip, set(dstip))

def graph_reducer (graph, (srcip, dstlist)):
  graph[srcip] = graph.get(srcip,set()).union(dstlist)
  return graph

input = open('router.log', 'r')
intermediate = map(ip_mapper, input)
graph = reduce(graph_reducer, intermediate, {})
```

We choose Python merely for illustration, to exemplify the simplicity of the underling concepts. The Python statements should easily map to similar constructs in other modern scripting languages, such as Perl and Ruby. Iterating over a file object (`input`) will yield a sequence of lines. The `ip_mapper` function will parse one line and return a source-destination pair. The call to `map` takes as input a sequence of lines and produces another sequence, of IP pairs. The second element is a set, for reasons that will become clear shortly.

The `reduce` operation accumulates all elements of an input sequence. Here, the accumulator is a dictionary (`graph`) which stores a mapping between a source IP (key) and set of destination IPs (value). The accumulation function is `graph_reducer`. Thus, the call to `reduce` above takes as input the sequence of IP pairs produced by `map` and outputs a dictionary of key-value pairs, where keys are source IPs and values are lists of destination IPs.

Even though this program structure is relatively simple, it is sufficiently general for many tasks [12, 10]. Map-Reduce allows users to execute such computations on data stored in a cluster with up to thousands of processors and petabytes of storage, *while requiring effort similar to that of writing the above Python program.*

## 2.2 Programming model and data flow

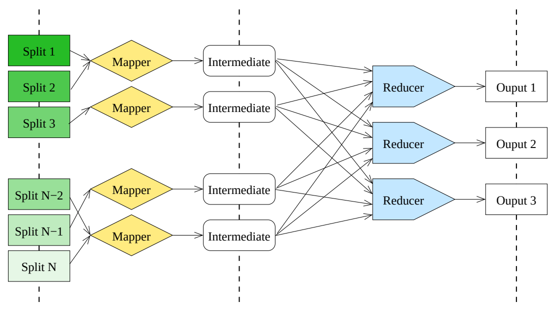
As it's name suggests, Map-Reduce draws from a well-established abstraction in functional programming. The previous example illustrates most of the programming model aspects. Formally, a computation is decomposed into a map operation followed by a reduce operation. These are specified by two functions,

$$\begin{aligned}\textsc{Mapper}: &\quad \langle k_{\text{in}}, v_{\text{in}}\rangle \mapsto \langle k_{\text{int}}, v_{\text{int}}\rangle\\ \textsc{Reducer}: &\quad \langle k_{\text{int}}, V \equiv \{v_{\text{int}}\}\rangle \mapsto \langle k_{\text{out}}, v_{\text{out}}\rangle\end{aligned}$$

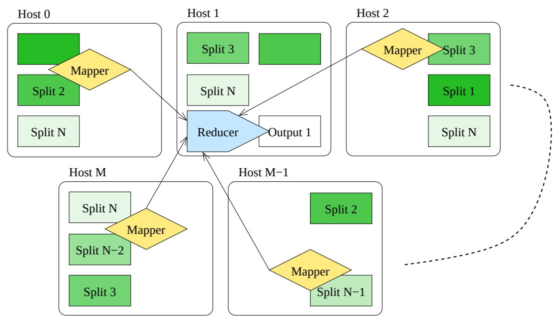
Both operate on key-value pairs, which we denote using angle brackets $\langle k, v\rangle$. The key is used primarily in the reduction step, to determine which values are grouped together. Values may carry arbitrary information.

## 2.3 Data flow

This abstract computation needs to be eventually executed on a large cluster. In this section we focus on the data flow model (see Figure 2a). The map input is be partitioned into a number of *input splits*. Processing each split is assigned to one *map task*. Subsequently, all map outputs are partitioned among a number of *reduce tasks*, by hashing on the intermediate key $k_{\text{int}}$. Each reducer receives one part of



(a) Data flow



(b) Placement

**Figure 2. Overview of the Map-Reduce execution framework [12].**

the intermediate key space. Subsequently, it merges all inputs received from all mappers, sorts them based on $k_{\text{int}}$ to group equal keys together, and applies the reducer function to obtain the final results.

## 2.4 Distributed execution framework

One important feature is that the storage cluster may partially or completely overlap with the compute cluster. This is also true even when more sophisticated distributed data stores [7] are used. Therefore, computation tasks can be executed on machines hosting local copies of the input data. Map-Reduce is essentially a simple and clean framework that allows a large class of computations to be transparently executed in such a cluster architecture. Map-Reduce resembles the concept of *active disks* [32], although the actual design and implementations are very different.

Figure 2b illustrates one possible placement of the elements from Figure 2a (both data chunks, as well as computation tasks) onto cluster machines. In this simple illustration, it is possible to place all map tasks on machines hosting a local copy of their input split. If this is not possible, then data will be transmitted over the network from a remote storage node. In addition, an intermediate *combiner* can be inserted between each mapper and the final reducer. Its purpose is to combine all local map outputs (using either the same or a separate reducer function) before they are sent out to the reducers.

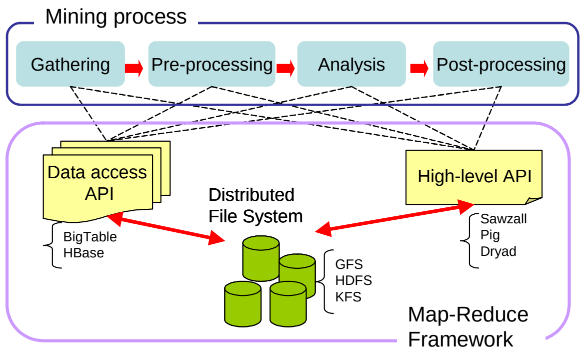


**Figure 3. Distributed mining process.**

Finally, the distributed execution model also takes care of load balancing and fault tolerance in a simple but effective way; see [12] for further information.

**Scalability** One of the main advantages of Map-Reduce is that it can transparently use any number of machines. If the volume of output data is much smaller than the volume of input data, as is typically the case, then co-location of tasks and data leads to performance improvement almost proportional to the number of nodes (see, e.g., Figure 1). We further discuss scalability in Section 5.

## 3 Distributed Mining Process

As shown in Figure 3, a distributed data mining process involves several steps: data gathering, pre-processing, analysis and post-processing, many of which involve distributed processing through either a data storage layer (such as GFS [19], HDFS[1] or KFS[26]), or a higher-level data access and job description language, such as Sawzall [31], Pig [3], or Cascading [37].

**Data gathering** This step involves identifying the source and obtaining the data. Some examples include (i) crawling millions of web pages, (ii) querying heterogeneous databases, (iii) large-scale scientific simulation, or (iv) distributed system monitoring. Most are performed in a distributed manner, and can be expressed as Map-Reduce jobs.

**Data pre-processing** After obtaining the raw data, an important step is to transform it into the appropriate format for data analysis. As a matter of fact, data cleaning often consumes the majority of time for exploratory data mining tasks. However, despite calls from several established researchers [16, 22], it has been largely ignored in the research literature. We can no longer afford to ignore this step.

Increasingly, many researchers (ourselves included) now find that they have to routinely deal with gigabytes or even terabytes of data. For example just parsing 4.5 terabytes of compressed text logs for 30 days worth of MSN instant messaging data was reported to take a total of five full days, on an eight processor machine with fast local disks [27]. We recently had similar experiences processing a 350 gigabyte raw network event log (similar to the example in Section 2.1). We needed over five hours to extract source/destination IP pairs, even though we were accessing the data over a 2Gbps Fibre Channel link to a SAN (dotted line in Figure 1). Similarly, the TREC data is 100GB of text. Pre-processing that on a single powerful machine (four cores and 32GB RAM) took several *days*. Compared to these luxury settings, we are able to achieve much better performance on a few commodity nodes running Hadoop. More importantly, setting up Hadoop required minimal effort (about two to three hours for a moderately experienced person).

Moreover, other members of our group took different approaches on the event log data. The first is the DPH (desperate Perl hacker) approach. The data were sorted and partitioned using a primary key consisting of a timestamp plus a unique record identifier. Subsequently, extracting time-dependent aggregates could be performed quickly. However, when we needed to extract a graph of IP-pairs, the pre-processing was not helpful. Furthermore, the effort to organize the data took approximately three days.

The second is the traditional database management system approach. Since the each event record contains widely different fields, depending on the event type, only the ten or so common fields were extracted (dropping all remaining possible fields, which number over one thousand but are not always present). Furthermore, without spending too much time to fine-tune MySQL's storage engine parameters, no more than a year worth of data (about a quarter of the total) could be bulk loaded successfully. Pre-processing the data into this common schema and building indices on all fields gave good performance. However, the effort required about two days and would be of no benefit if any of the dropped fields needed to be analyzed.

We are not suggesting that the alternative approaches were handled in the optimal way. However, in a relative effort-to-benefit ratio, we believe that Hadoop wins.

Specifically for co-clustering, there are two main pre-processing tasks:

- Building the graph from raw data.
- Pre-computing the transpose.

The first step primarily involves extracting the graph (e.g., source-destination or document-term pairs) and may also involve other related tasks (such as stemming and stopword removal). During co-clustering optimization, we need to iterate over both rows and columns. Therefore, we need to pre-compute the adjacency lists for both the original graph as well as its transpose. Transposition is very similar to computing an inverted index, one of the applications Map-Reduce was originally developed for [12]. This step typically took a few minutes. In Section 5, we describe actual times on real-world data processing in detail.

| Sym. | Definition |
|---|---|
| $\mathbf{A}$ | the $m \times n$ data matrix |
| $m, n$ | Number of rows and columns. |
| $i, j$ | Row/column indices, $1 \leq i \leq m$, $1 \leq j \leq n$. |
| $a_{i,j}$ | The $(i, j)$ element of $\mathbf{A}$. |
| $\mathbf{G}$ | the $k \times l$ group matrix |
| $k, l$ | Numbers of row- and column- groups. |
| $p, q$ | Group indices, $1 \leq p \leq k$, $1 \leq q \leq l$. |
| $g_{p,q}$ | The $(p, q)$ element of $\mathbf{G}$. |
| $I_p$ | Set of rows belonging to the $p$-th row group. |
| $J_q$ | Similar to $I_p$, but for columns |
| $m_p$ | the size of $p$-th row group, $m_p \equiv \lvert I_p \rvert$, $1 \leq p \leq k$. |
| $n_q$ | the size of $q$-th col. group, $n_p \equiv \lvert J_q \rvert$, $1 \leq p \leq k$. |
| $\mathbf{r}$ | Row group assignments. |
| $\mathbf{c}$ | Column group assignments. |
| $H(.)$ | Shannon entropy function |

**Table 1. Definitions of symbols**

**Data analysis** In practice, even after data pre-processing, the data can still be too big to analyze in a centralized manner. For example, the adjacency lists for TREC are about 4GB each. At a total of over 8GB for both the original matrix and its transpose, few machines have enough memory to even load the entire graph.

Because of the huge data, we see more and more data analysis done in a distributed fashion. Without relying on different infrastructure, many analyses can be done in the same environment where data are gathered and processed, using the same Map-Reduce programming model and exploit parallelism and fault-tolerance.

The next section presents the details of our Distributed Co-clustering (DisCo) framework using Map-Reduce.

**Post-processing** The analysis results need to be visualized, or sometimes turned into the input for other applications. They can also reside in the same environment.

## 4 Co-clustering Huge Datasets

In this section we present the main design for distributed co-clustering using Map-Reduce. First, we give a very brief overview of the necessary co-clustering definitions. Then we explain how the necessary computations can be performed as map and reduce operations. Finally, we conclude with a brief description of certain important implementation considerations.

### 4.1 Definitions and overview

Matrices are denoted by boldface capital letters, e.g., $\mathbf{A}$ and vectors are denoted by boldface lowercase letters, e.g., $\mathbf{a}$. The $(i, j)$-th element of matrix $\mathbf{A}$ is $a_{ij}$, the $i$-th row of $\mathbf{A}$ is $a_{i:}$, and the $j$-th column of $\mathbf{A}$ is $a_{:j}$.

We focus on algorithms that employ a checkerboard decomposition of the original adjacency matrix into a grid of

$$\mathbf{A} = \begin{bmatrix} 0\,1\,0\,1\,1 \\ 1\,0\,1\,0\,0 \\ 0\,1\,0\,1\,1 \\ 1\,0\,1\,0\,0 \end{bmatrix} \xrightarrow[\mathbf{c} = (\,2\;1\;2\;1\;1)^{\mathrm{T}}]{\mathbf{r} = (\,2\;1\;2\;1)^{\mathrm{T}}} \left[\begin{array}{cc|ccc} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{array}\right] = \mathbf{B}$$

**Figure 4. A co-clustering example: Given $\mathrm{A}$, find group assignments $\mathrm{r}$ and $\mathrm{c}$ such that the resulting sub-matrices in $\mathrm{B}$ are highly correlated.**

sub-matrices, also allowing row and column permutations). Formally, given an $m \times n$ matrix, a co-clustering is a pair of row and column labeling vectors

$$\mathbf{r} \in \{1, 2, \ldots, k\}^m \quad \text{and} \quad \mathbf{c} \in \{1, 2, \ldots, \ell\}^n,$$

so that each element of $(r)$ is the group label $r(i)$ for the $i$-th row of the matrix, $1 \leq i \leq m$ and $1 \leq r(i) \leq k$, and similarly for the columns.

In addition to the label vectors, another key structure is the $k \times \ell$ *group matrix* $\mathbf{G}$. Different co-clustering algorithms construct $\mathbf{G}$ in different ways, but the intuition is the same: $g_{pq}$ gives the sufficient statistics for the $(p, q)$ sub-matrix, which corresponds to the intersection of $p$-th row group and $q$-th column group.

The goal is to find good group assignment vectors such that an error function is minimized. Various co-clustering algorithms have adopted different error functions, such as minimum mutual information [15], sum-squared distance [9], and code length [6]. A general co-clustering framework based on Bregman divergence [4] has been proposed, which covers the entire exponential family.

For example, in Figure 4, given the $4 \times 5$ input matrix $\mathbf{A}$, the goal is to find $\mathbf{r}$ and $\mathbf{c}$ such that after permutation according to $\mathbf{r}$ and $\mathbf{c}$, the correlated sub-matrices are grouped together. Searching for the optimal group assignment is NP-hard [15]. Therefore, a common approach is to do local search, alternating between row and column assignments, while holding the other assignment fixed. The basic steps are the following:

- **Row iteration:** Fixing the current column group assignment $\mathbf{c}$, iterate over each row, assigning to the "best" row group, finally obtaining (i) an updated $\mathbf{r}$, and (ii) an updated $\mathbf{G}$.
- **Global sync:** Based on the new labels and group matrix, the error function can be estimated to evaluate whether an improvement was achieved or not.
- **Column iteration and sync:** Fix $\mathbf{r}$ and perform a similar iteration over columns, to obtain updated $\mathbf{c}$ and $\mathbf{G}$.

The high-level pseudo-code is listed in Procedure 1. This large family of co-clustering algorithms, which includes all those cited above, satisfies two key conditions:

1. The error function can be computed using $\mathbf{r}$, $\mathbf{c}$, and $\mathbf{G}$, without resorting to the raw data.

**Procedure 1** CC ($\mathbf{A}$, $k$, $l$)

```
1:  Initialize $\mathbf{r}$ and $\mathbf{c}$.
2:  Compute the group statistics matrix $\mathbf{G}$.
3:  **repeat**
4:      **for each** row $i = 1..m$ **do**
5:          **for each** row group label $p = 1..k$ **do**
6:              Assign $r(i) \leftarrow p$ if this minimizes error
7:      Update $\mathbf{G}$, $\mathbf{r}$
8:      Do the same for columns
9:  **until** cost does not decrease
10: **return** $\mathbf{r}$ and $\mathbf{c}$
```

2. The decision in line 5 of Algorithm 1 can be made based on $\mathbf{r}$, $\mathbf{c}$ and $\mathbf{G}$ from the previous assignments, and on the values $a_{i:}$ of the $i$-th row, without resorting to the values of other rows or columns.

These conditions are central, but also quite broad. The first is essentially a statement about the sufficient statistics (they can be computed as aggregates over each sub-matrix), whereas the second is a statement about the optimization strategy (local, greedy search).

## 4.2 Co-clustering with Map-Reduce

Next, we seek map and reduce functions to perform the alternating updates using the Map-Reduce framework.

The idea is to initiate two Map-Reduce jobs for row and column iterations, and a synchronization step in between to update the global parameters $\mathbf{G}$, $\mathbf{r}$, and $\mathbf{c}$.

The pseudo-code is the same as before except we need two kind of Map-Reduce jobs: (i) initializing the group matrix $\mathbf{G}$ and label vectors $\mathbf{r}$ and $\mathbf{c}$, and (ii) performing row or column iteration. We use random initialization, and a simple Map-Reduce job can be formed (omitted for space).

Now we discuss how to formulate a Map-Reduce job for row and column iteration. Figure 5 shows how we express one iteration over rows as a Map-Reduce computation. The iteration over columns operates on the adjacency list of the transpose matrix in a similar way. This is the basic building block, where most of the time is actually spent.

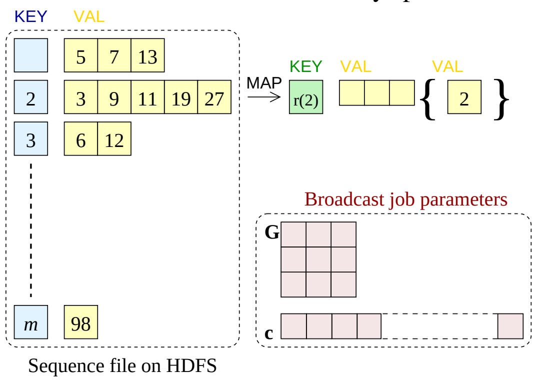


**Figure 5. One iteration over rows as a Map-Reduce job.**

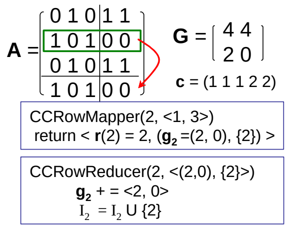


**Figure 6. Running map and reduce functions.**

**Map-function** The adjacency list is stored on HDFS as a sequence file of key-value pairs. The key is the row index $i$ and the value is the adjacency list, i.e., $\{j | 1 \leq j \leq n, a_{ij} \neq 0\}$, along with the values $a_{ij}$. The group matrix $\mathbf{G}$, as well as the column labels $\mathbf{c}$ are globally broadcast to all mappers. Given this information, the mapper can compute the locally optimal row label $r(i)$ for each row $i$, as well as the associated per-column statistics for that row. The labels $r(i)$ are the intermediate keys. The intermediate values comprise of the row group statistics $\mathbf{g}_i$ and the membership information $\{i\}$. The pseudocode for CCRowMapper is shown in Procedure 2.

**Procedure 2** CCRowMapper ($k$, $v$)

```
**Globals:** Cluster statistics $\mathbf{G}$, labels $\mathbf{c}$
    Source node is $i \equiv k$
    Adjacency list of $i$ is $a_{i:} \equiv V$
    Compute row statistics $\mathbf{g}_i := \text{RowStatistics}(a_{i:}, \mathbf{c})$
    **for each** group label $p = 1..k$ **do**
        **if** assigning $i$ to $p$ would lower cost **then**
            $r(i) \leftarrow p$
    **emit** $\langle r(i), (\mathbf{g}_i, \{i\}) \rangle$
```

Note that updating the row group statistics $\mathbf{g}_i \in \mathbb{R}^\ell$ varies for different co-clustering algorithms. In the experiments we rely on the cross-association cost function [6], which uses the number of non-zero columns per column group. Formally, $\mathbf{g}_i(p) := \#\{j | a_{ij} \neq 0, \mathbf{c}(j) = p\}$. For example, in Figure 6, $\mathbf{g}_2 = (2, 0)$ because the second row has two non-zero columns in the first column group, and none in the second column group.

**Reduce function** The reducer merges the row group statistics and group members for each cluster label. For example, in Figure 6, the intermediate key-values $2, < (2, 0), 2 >$ are aggregated by vector addition over the 2nd row in $\mathbf{G}$, $\mathbf{g}_2$, and the set union of row 2 to $I_2$. The pseudocode for CCRowReducer in more detail is as follows.

**Global sync** Finally, we need to collect the new results for the $\mathbf{G}$ matrix and $\mathbf{r}$ row-label vector, as shown in CollectResults.

**Overall picture** This building block is then used in the alternating minimization to find a co-clustering for given $k$ and $\ell$ [6, 4], as well as to search for $k$ and $\ell$ themselves [6].

**Procedure 3** CCROWREDUCER $(k, V)$

```
Row group label is $p \equiv k$
Initialize $\mathbf{g}_p \leftarrow 0$, $I_p \leftarrow \emptyset$
**for each** map value $(\mathbf{g}, I) \in V$ **do**
    $\mathbf{g}_p \leftarrow$ COMBINESTATISTICS$(\mathbf{g}_p, \mathbf{g})$
    $I_p \leftarrow I_p \cup I$
**emit** $\langle p, (\mathbf{g}_p, I_p)\rangle$
```

**Procedure 4** COLLECTRESULTS

```
Initialize $\mathbf{G} \leftarrow 0$, $\mathbf{r} \leftarrow 0$
**for reduce output** $\langle p, (\mathbf{g}_p, I_p)\rangle$ **do**
    $g_{p:} \leftarrow \mathbf{g}_p$
    $r(i) \leftarrow p$, for all $i \in I_p$
**return** $\mathbf{G}$ and $\mathbf{r}$
```

For the latter, some additional modifications are necessary. We can still decide which cluster to split in a way similar to [6], but the computation to decide how to split it (i.e., which row/column goes to the new group) is not parallelizable. We found that using a 50-50 random split (so that half of the rows/columns go into the new cluster and half remain in the old cluster) is very effective and often yields better results than the criterion of [6]; it always gives better results if we exploit cluster resources by doing multiple random trials.

### 4.3 Implementation

Finally, we conclude with some brief remarks about implementation considerations. Figure 7 shows the execution timeline for a row iteration Map-Reduce job. In this case, there is one reduce task at the top of the figure. All other are map tasks, shown in green.

**Performance tuning** Even though Map-Reduce hides most of the complexities of distributed execution, there are still some parameters that need to be decided appropriately, to improve performance. Setting them is relatively intuitive, but does require some care. We quantify these in Section 5. The first parameter has to do with thread pool sizes, which needs to be configured for the cluster at hand. We found that setting it to the number of cores per node plus 50% is sufficient to achieve reasonably good node utilization. The other important parameters are number of map tasks (which can be implicitly determined by setting the minimum input split size, as well as explicitly set), as well as the number of reduce tasks. The input split size can be increased for large input files, to avoid unnecessary task startup overheads. The number of reducers should be set according to the size of the intermediate key space, as well as the expected number of intermediate pairs (after combination on each map task). For co-clustering, the number of distinct intermediate keys is equal to the number of clusters ($k$ or $\ell$) and the number of intermediate pairs is proportional to that. These are typ-

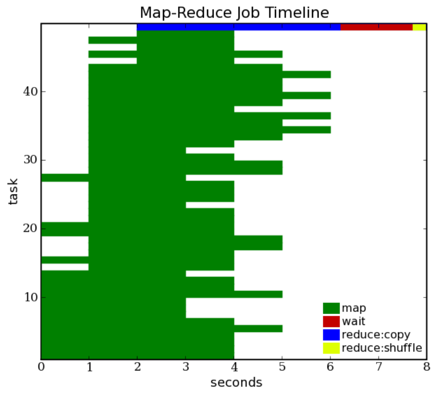


**Figure 7. Execution timeline for a row iteration map-reduce job.**

ically small numbers, so it is best to use one reduce task. However, for building the graph in the pre-processing step, as well as for inverting the matrix, one reduce task becomes a bottleneck, so it is best to increase this, up to the number of machines available in the cluster.

**Search randomization** As described in Section 4.2, we do random cluster splits. We can further exploit the availability of many machines to do multiple split trials and pick the best. In this case, we can achieve up to 10% better final cost objectives than [6]—in addition to the scalability benefits that the Map-Reduce framework offers us.

## 5 Experiments

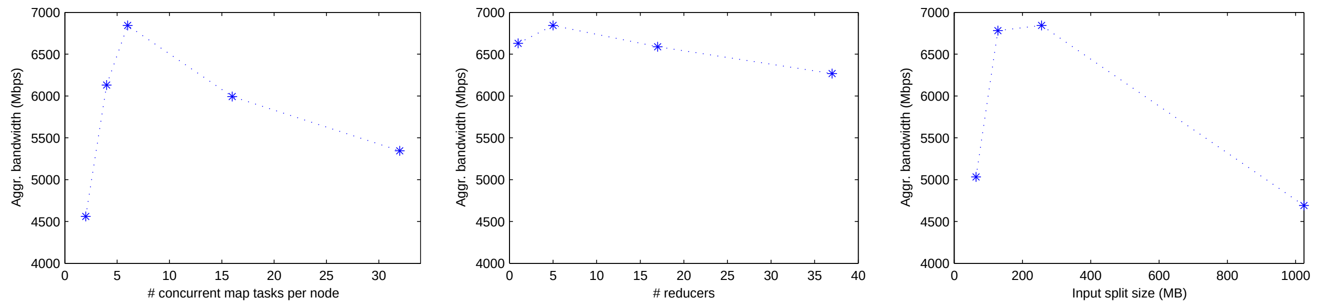
In this section we describe running time measurements from our Hadoop deployment of DisCo. We focus on scalability, as well as performance tuning. We should note that, although we did consider alternative approaches (see discussion in Section 3), these proved impractical in the long term.

### 5.1 Setup

We performed all experiments on 39 nodes in our cluster. These were blade servers with two dual-core processors (typically Intel Xeon 2.66GHz, with a few 3GHz machines), all with 8GB RAM and all running Linux RHEL4.

The machines were located in four different blade-center enclosures, three of which were in the same rack. All have Gigabit ethernet connections. The switching fabric within one blade-center enclosure has an aggregate bandwidth of 4Gbps. The same holds for the switches between blade centers, only those links are shared with a larger number of other nodes that may interfere with network traffic.

Each of the blades had a locally attached hard drive. Blade servers typically have small and relatively slow hard drives. The drives in these machines are SATA and can achieve a sustained read performance (measured by `cat >/dev/null` of a large file) of about 65MB/sec, or



(a) no. of concurrent maps (b) no. of reduce tasks (c) input split size

**Figure 8. Performance tuning using various parameters, for the pre-processing step.**

roughly 500Mbps. The total capacity of our HDFS cluster was just 2.4 terabytes. HDFS block size was set to 64MB (default value). For large files we used a replication factor of 2, and for smaller files (less than a few hundred megabytes) we used a replication factor between 8–12 (depending on file size), so we can a larger number of mappers (ideally all of them) that work on local replicas of the data.

The cluster is shared with other users. The namenodes as well as the jobtracker were placed on two separate master nodes, different from the set of 39 nodes used for storage and computation. Under normal operation, we use a nice level of five, but for timing experiments we raised it to zero and repeated each measurement three times, taking an average. All code[3] was implemented completely in Java (with some parts using the GNU Trove collections for efficiency) and we used Sun JDK version 1.6.0_03 for everything. We used three real datasets, summarized in Table 2.

| Dataset | Raw | Graph | Size | Non-zeros |
|---|---|---|---|---|
| ISS | 350GB | 170MB | 2,483,513×1,283,449 | 9M |
| TREC | 100GB | 4.3GB | 1,237,744×61,241 | 359M |
| Netflix | — | 1.1GB | 480,189×17,770 | 100M |

**Table 2. Summary of datasets.**

## 5.2 Scalability and performance

We report our experiences with respect to scalability and performance tuning. In particular, we first study wall-clock time versus number of nodes, to characterize scalability. Next, we study performance tuning and the dependence on the following parameters: (i) minimum input split size (which equivalently determines the number of maps), (ii) maximum number of concurrent map tasks per node, and (iii) number of reducer tasks.

Figures 1 and 8 shows the results for the pre-processing step on the ISS data. The default values are: 39 nodes, 6 concurrent maps per node, 5 reduce tasks, and 256MB input split size. First, we observe that for this task, aggregate throughput scales almost linearly with respect to number of nodes (Figure 1). Next, we observe that as we increase the number of concurrent map tasks, we achieve better utilization of each node. The optimum is reached at a number slightly larger than the number of cores on each node. Increasing much beyond that starts causing unnecessary overheads (although we never reached the point of thrashing). The number of reducers for this task that gives peak throughput is about 5. One reducer becomes a bottleneck, whereas a larger number seems to create unnecessary overheads. Finally, as we increase block size, we see that a small size causes overheads since there is a large number of map tasks, and therefore a larger number of HTTP request to each reducer, to transfer intermediate results. However, as we increase the split size to several multiples of HDFS block size, it becomes much more difficult to place map tasks on local copies of the data, so performance degrades due to unnecessarily high network traffic.

Finally, Figure 9 shows aggregate throughput versus number of nodes for one co-clustering iteration. Compared to Figure 1, we see a scaleup at about the same rate (135Mbps/node versus 175Mbps/node) up to about 10 nodes. Performance reaches a plateau after about 25 nodes, corresponding to about $20 \pm 2$ seconds per iteration.

The bandwidth falls off because, as the job size decreases, framework overheads begin to dominate processing time. Based on closer investigation, there are two overheads: (i) fundamental ones, related to transmitting task parameters, spawning the processes that will execute them, storing results on HDFS, and so on; and (ii) some design decisions that are specific to Hadoop, which uses busy loops with a hardcoded sleep interval[4].

Despite these partly "artificial" limitations[5], there are two important observations. First, compared to our previous main-memory implementation, where the performance-critical inner loop is written in C (as a MEX module; the rest of the code is in Matlab), $20 \pm 2$ seconds per iteration is equal or better to what we can get on a machine with 48GB of RAM, due to thrashing. Second, as the dataset sizes grow in future, our implementation will achieve linear scaleup. Thus, we achieve our goal of aiming towards peta-scale mining.

[3]See `http://www.bitquill.net/trac/wiki/PCC/Start`.

[4]We decreased the largest ones, which were 5 seconds, to 0.5 seconds for these experiments; still, the job client and job tracker spent a total of about 1–3 seconds in sleep. We always report wall-clock times.

[5]In fairness, Hadoop is not primarily optimized for many short jobs.

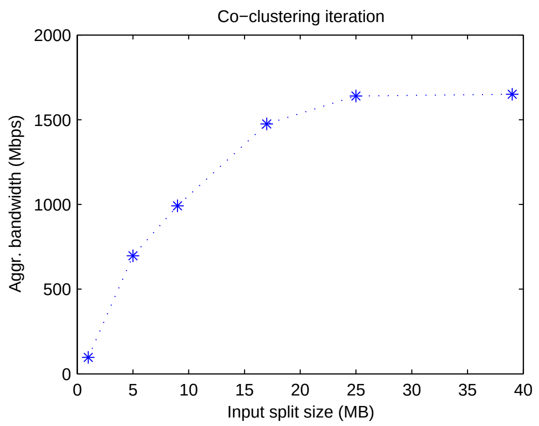


**Figure 9. Co-clustering iteration scalability (TREC data, 4GB on HDFS after post-processing): scales at the same rate as Figure 1 up to ten nodes, due to the relatively small dataset size.**

The general behavior of the co-clustering iterations with respect to the other three parameters are almost identical with those in Figure 8 (except that the absolute numbers are smaller and in accordance to Figure 9), and omitted for space. One difference (when running with a default of 17 nodes in the cluster) is that the peak in bandwidth versus number of reducers occurs at one reducer, rather than five. This is expected, since the size of intermediate results is much smaller, as pointed out in Section 4.3.

**Pre-processing** Finally, we report the proportion of time spent on pre-processing versus co-clustering. As discussed before, pre-processing is often overlooked, even though it is a pre-requisite for any mining task. Figure 10 describes our experience and should be taken with a grain of salt. To have some common ground for comparison, the time used for co-clustering is that for 30 pairs of row and column iterations. The actual number of iterations varies from one to over one hundred, although the average we observed is around 100 iterations. We should also point out that pre-processing here is performed with map-reduce – otherwise, pre-processing itself takes up to days. Even one might argue that going back to the original data is something that happens rarely, we still wish to point out that (i) this may not always be the case, and (ii) the effort required is still significant.

## 6 Related work

### 6.1 Map-Reduce framework

As a programming model, Map-Reduce adopts a extremely simple but powerful abstraction from functional programming. Many data processing tasks can be easily formulated as Map-Reduce jobs as shown in [13]. Inspired by that, many higher-level programming abstractions have been implemented for large-scale data processing, such as Sawzall [31], Dryad [25] and DryadLINQ, PIG [3], SPADE [17], FREERIDE-G [21], and Sector [23], among others.

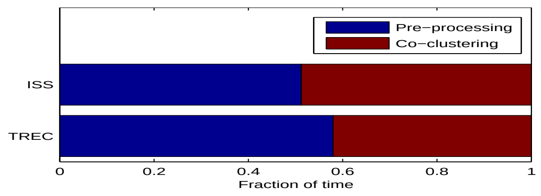


**Figure 10. Fraction of time spent pre-processing vs. co-clustering.**

Much of the power of Map-Reduce derives from its use of the Google File System (GFS) [19] (or similar file systems such as HDFS [1] and KFS [26]) as the underlying data store. GFS is similar to other distributed file systems such as [20, 33, 36, 5] in that it employs a distributed storage cluster. However, they employ block-addressable storage and a centralized metadata server (which essentially stores the mapping between filenames and a list of blocks and their locations in the storage cluster). Several higher-level data storage abstractions provides a convenient data access and storage API for Map-Reduce tasks, such as Bigtable [7], HBase [1], and Hypertable [38].

As data grows, data mining and machine learning applications also start to embrace the Map-Reduce paradigm for e.g., news personalization [11], or several machine learning algorithms on multicore architectures [10]. Compared to them, our focus is to illustrate a complete data mining process involving multiple interconnected steps that all require large-scale data processing.

### 6.2 Co-clustering

Co-clustering has been studied in many different applications including text mining [15, 28], genes and experimental conditions in bioinformatics [24, 8], recommender systems [18], and graph mining [6].

Many co-clustering algorithms have been proposed, depending on the cluster shapes, the properties of input data, and optimization objectives. Different cluster shapes include checkerboard partitions, single bicluster, exclusive row and column partitions and overlapping partitions. For a detailed discussion, see survey [29]. In this work, we focus on checkerboard partitioning such as those in [4, 15, 6].

Various of optimization criteria have been proposed, such as minimum mutual information [15], sum-squared distance [9], and code length [6]. A general co-clustering framework based on Bregman divergence [4] has been proposed for covering the entire exponential family. In this work, we utilize the code length objective, but the algorithm can apply to the other cases with minor modifications.

## 7 Conclusions

In summary, this paper presents our findings and valuable lessons from designing a framework for a holistic approach to data mining, in the context of the co-clustering task. Our experiences (Section 3) led us to consider a distributed infrastructure. Given the decreasing prices of magnetic storage and the ever increasing rate of data collection, the necessity of data mining algorithms on distributed infrastructures is clear. In a growing open source ecosystem of scalable, distributed data processing and management components, Map-Reduce is emerging as the predominant elementary abstraction for distributed execution of a large class of data-intensive processing tasks.

Co-clustering has many important applications, including text mining, graph mining and collaborative filtering. This paper is a case study on a distributed co-clustering framework, presenting our design and describing the lessons we learned. We demonstrate that we can achieve I/O rates that exceed those of high-performance storage systems (e.g., SAN over Fibre Channel running a high performance file system such as GPFS), using relatively low-cost components. More importantly, performance scales almost linearly with the number of machines/disks. Finally, we demonstrate results on large, real-world data sets.

## 8 References

[1] Hadoop. `http://hadoop.apache.org/core/`.

[2] HBase. `http://hadoop.apache.org/hbase/`.

[3] PIG. `http://incubator.apache.org/pig/`.

[4] A. Banerjee, I. Dhillon, J. Ghosh, S. Nerugu, and D. S. Modha. A generalized maximum entropy approach to Bregman co-clustering and matrix approximation. In *KDD*, 2004.

[5] CFS. Lustre file system. `http://www.lustre.org/`.

[6] D. Chakrabarti, S. Papadimitriou, D. Modha, and C. Faloutsos. Fully automatic cross-associations. In *KDD*, 2004.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[8] Y. Cheng and G. M. Church. Biclustering of expression data. In *ISMB*, 2000.

[9] H. Cho, I. Dhillon, Y. Guan, and S. Sra. Minimum sum-squared residue co-clustering of gene expression data. In *SDM*, 2004.

[10] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *NIPS*, 2006.

[11] A. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. In *WWW*, 2007.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *CACM*, 51(1), 2008.

[14] D. J. DeWitt and M. Stonebraker. MapReduce: A major step backwards. `http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html`.

[15] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *KDD*, 2003.

[16] U. Fayyad. From mining the web to inventing the new sciences underlying the internet. `http://www.sigkdd.org/kdd2007/program.html#invited2`, 2008.

[17] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *SIGMOD*, 2008.

[18] T. George and S. Merugu. A scalable collaborative filtering framework based on co-clustering. In *ICDM*, 2005.

[19] S. Ghemawat, H. Gobioff, , and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[20] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *ASPLOS*, 1998.

[21] L. Glimcher and G. Agrawal. FREERIDE-G: Enabling distributed processing of large datasets. In *HPDC*, 2008.

[22] R. Grossman. Data mining FAQ. `http://www.rgrossman.com/dm.htm`.

[23] R. L. Grossman and Y. Gu. Data mining using high performance clouds: Experimental studies using sector and sphere. In *KDD*, 2008.

[24] J. A. Hartigan. Direct clustering of a data matrix. *J. Am. Stat. Assoc.*, 67(337), 1972.

[25] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[26] Kosmix. Kosmos distributed file system (KFS). `http://kosmosfs.sourceforge.net/`.

[27] J. Leskovec and E. Horvitz. Planetary-scale views on an instant-messaging network. In *WWW*, 2008.

[28] H. Li and N. Abe. Word clustering and disambiguation based on co-occurence data. In *COLING-ACL*, 1998.

[29] S. C. Madeira and A. L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM TCBB*, 1, 2004.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[31] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Prog. J.*, 13(4), 2005.

[32] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 2001.

[33] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[34] M. Seltzer. Beyond relational databases. *CACM*, 51(7), 2008.

[35] M. Stonebraker and U. Cetintemel. One size fits all: An idea whose time has come and gone. In *ICDE*, 2005.

[36] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[37] C. K. Wensel. Cascading. `http://www.cascading.org/`.

[38] Zvents. Hypertable. `http://hypertable.org/`.