MapReduce based algorithms for localized bisimulation

April 10, 2013
Intuitively, two nodes in a graph are bisimilar if they share basic structural properties such as labelling and neighbourhood topology. The node set of a graph can be partitioned based on this equivalence relation. Computing the bisimulation partition of a graph is a fundamental problem which can play a key role in a wide range of applications, including improving performance of XML or RDF indexing, graph compression, query processing and data analytics. Moreover, graphs which are of interest typically are massive and can contain billions of nodes and edges such as social networks or linked open data. So in order to analyse this data, in-memory solutions are infeasible and even I/O-efficient solutions do not scale well with these massive graphs.

Hence, we look for parallel or distributed solutions to process these big graphs. In this thesis we present, to our knowledge, the first efficient MapReduce-based algorithm for computing the bisimulation partition of massive graphs. We also present two optimizations for dealing with skew in the partition block sizes, which often occurs in real-world datasets. The results of an extensive empirical study are presented which show the effectiveness and scalability of our solution.
First and foremost I would like to thank my parents for the love, kindness and support during my studies, without them none of this would have been possible.

Secondly, I would like to express my greatest gratitude to George Fletcher for supervising my graduation and Yongming Luo for helping me with the theory and implementation. Without their guidance and advice this thesis would never came to be. I would also like to thank the assessment committee members, Paul de Bra and Boudewijn van Dongen as well as Jan Hidders and Yuqing Wu, for reviewing my thesis and attending my presentation, all the while providing critical feedback and insightful ideas.

Finally, I would like to thank Evert Lammerts and all the people at SURFsara for allowing me use one of the largest HADOOP clusters in the Netherlands, it is truly humbling to have so much computing power ready at the push of a button.

Yannick de Lange
1 Introduction 1

2 Preliminaries 3
  2.1 \(k\)-bisimulation 3
  2.2 Signatures 4
  2.3 MapReduce 4
  2.4 HADOOP 5
  2.5 Related work 6
  2.6 Summary 6

3 Base algorithm for computing \(k\)-bisimulation 7
  3.1 Overview 7
  3.2 Create signatures task 8
  3.3 Assign identifiers task 9
  3.4 Re-Partition nodes task 9
  3.5 Example run 10
  3.6 Summary 10

4 Optimizations for dealing with skew 13
  4.1 Optimization A: Assigned \(pId\) values 13
    4.1.1 Solution 13
    4.1.2 Example run 15
    4.1.3 Limitations 17
  4.2 Optimization B: Additional merge task 17
    4.2.1 Solution 17
    4.2.2 Example run 19
    4.2.3 Limitations 21
  4.3 Summary 21

5 Experimental set-up 23
  5.1 Datasets 23
  5.2 Environment 25
    5.2.1 HADOOP 25
5.2.2 HDFS .................................................................................. 25
5.3 Experiments ............................................................................. 25
  5.3.1 Skew analysis .................................................................... 26
  5.3.2 Parameter fitting ................................................................. 26
  5.3.3 Overall performance comparison ......................................... 27
5.4 Summary .................................................................................. 27

6 Experimental analysis ................................................................. 29
  6.1 Skew analysis and experiments on optimization strategies .... 29
  6.2 Parameter fitting ................................................................... 32
  6.3 Overall performance comparison ........................................... 34
  6.4 Experiences ........................................................................... 36
  6.5 Summary .............................................................................. 37

7 Conclusion .................................................................................. 39

A Source code .............................................................................. 41
  A.1 Overview ............................................................................... 41
  A.2 Dependencies ....................................................................... 42
  A.3 Running the algorithm .......................................................... 42

Bibliography .................................................................................. 45
Graphs have been becoming increasingly important in the last few years and have attracted attention from many different research communities, such as within data management and semantic web groups. Graphs can represent many different types, and usually large volumes, of data such as a computer network, knowledge databases or even massive social media networks. Graphs which are interesting from a practical perspective usually come in the order of billions of nodes and edges. With this large volume of data, in-memory solutions to graph problems typically become infeasible. The next logical step would be external memory algorithms, however their performance often scales linearly with the input size, which is then limited by the throughput of the I/O devices attached to the system.

One viable alternative to solving these problems are distributed solutions, which ideally provide a level of scalability linear with the number of computing resources, making use of the parallelism of the infrastructure. This allows for processing of large volumes of data with a reasonable performance as long as the number of computing resources is high enough. Recently, the MapReduce framework [DG08] has become the de-facto standard for parallel processing of large data and has gained a wide adoption in both industry and research fields.

The localized bisimulation partition of a graph, informally, is assigning each node in the graph to a unique partition block where all members have the same structural properties (e.g., node label, neighborhood topology, etc.). Creating these partitions can vastly improve performance of constructing indexes for XML or RDF [MS99, PLF+12] databases, or many other applications for general graph data such as compression [BGK03, FLWW12], query processing [KSBG02] and data analytics [Fan12]. The goal of our research is to develop a parallel solution for solving localized bisimulation partitioning of a graph, which can play a key role in a range of basic applications [SR11].

In this thesis we present a parallel MapReduce algorithm for computing localized bisimulation partitions. We give a detailed description and pseudo code together with a running example to show the inner workings of the algorithms. We also propose two optimizations to mitigate the effects of skew in the data. One optimization uses fixed values for signatures in order to spread the load more evenly, the other introduces a merge step to provide a correct solution with loosened restrictions. Furthermore, we empirically show that our solution outperforms the current state-of-the-art I/O-solution for various synthetic and real-world datasets.
Overview

The rest of the thesis is structured as follows:

- In Chapter 2 we first describe the preliminary information such as the definition of $k$-bisimulation and how to compute it on a graph, the signatures which is the foundation of the algorithm to compute the $k$-bisimulation partition, the MapReduce framework, HADOOP and the related research.

- In Chapter 3 describes the base MapReduce solution for computing the $k$-bismiulation partition we have developed.

- In Chapter 4 describes two optimizations we have developed to improve the performance.

- In Chapter 5 describes the experimental set-up for which we have ran all experiments.

- In Chapter 6 we present our results and give an empirical analysis to show that our algorithm performs efficiently.

- Finally in Chapter 7 we make concluding remarks and describe future work.

We would also like to note that a paper summarizing this research is in press [LdLF+13].
In this chapter we discuss the notion of $k$-bisimulation, signatures for computing this localized version of bisimulation, the MapReduce framework, the HADOOP implementation of MapReduce and finally related work that has been done in the area of $k$-bisimulation calculation.

2.1 $k$-bisimulation

When dealing with graphs, bisimulation is often one of the first topics to be mentioned. Bisimulation is a way to define equivalence between two graphs. Intuitively, it is assigning each node in a graph to a unique partition where all members have the same structural properties (e.g., node label, neighbourhood topology, etc.). For our algorithm, we use a localized version, so rather than looking at the complete graph we only compare nodes which are in the local neighbourhood radius of $k$ where $k \geq 0$. We do this because a localized version is less computation intensive and, for practical solutions, using localized bisimulation yields results comparable with full bisimulation.

A more formal way to describe this can be seen in Definition 1, where $N$ is a finite set of nodes, $E \subseteq N \times N$ is the set of edges, $\lambda_N$ is a function from $N$ to a set of node labels $\mathcal{L}_N$, and $\lambda_E$ is a function from $E$ to a set of edge labels $\mathcal{L}_E$.

**Definition 1** Let $k$ be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. Nodes $u, v \in N$ are called $k$-bisimilar (denoted as $u \approx^k v$), iff the following holds:

1. $\lambda_N(u) = \lambda_N(v)$,
2. if $k > 0$, then $\forall u' \in N[(u, u') \in E \Rightarrow \exists v' \in N[(v, v') \in E, u' \approx^k v' \text{ and } \lambda_E(u, u') = \lambda_E(v, v')]$],
   and
3. if $k > 0$, then $\forall u' \in N[(u, u') \in E \Rightarrow \exists v' \in N[(u, u') \in E, v' \approx^k u' \text{ and } \lambda_E(v, v') = \lambda_E(u, u')]]$.

However, we are not so much interested if two graphs are $k$-bisimilar. We are interested in partitioning the node set of a graph where all the nodes in the same partition “block” are $k$-bisimilar. So we want to assign a partition identifier to each node, denoting to which partition block it belongs. This is defined in Definition 2.

**Definition 2** A $k$-partition identifier for graph $G = \langle N, E, \lambda_N, \lambda_E \rangle$ and $k \geq 0$ is a set of $k + 1$ functions $\mathcal{P} = \{pId_0, \ldots, pId_k\}$ such that, for each $0 \leq i \leq k$, $pId_i$ is a function from $N$ to $\mathbb{N}^+$, and, for all nodes $u, v \in N$, it holds that $pId_i(u) = pId_i(v)$ iff $u \approx^i v$. 

3
However, efficiently finding the \( p\text{Id}_k \) for a node is not as trivial as it may appear. We solve this by generating signatures which represent the structural properties for each node.

### 2.2 Signatures

In order to assign a partition block to each node we build a signature of each node based on its children. The formal notion can be seen in Definition 3. Intuitively, a signature can be seen as a way to encode the structure of the sub-graph with the node as root and a locality of the current iteration.

**Definition 3** Let \( G = \langle N, E, \lambda_N, \lambda_E \rangle \) be a graph, \( k \geq 0 \), and \( \mathcal{P} = \{ p\text{Id}_0, ..., p\text{Id}_k \} \) be a \( k \)-partition identifier for \( G \). The \( k \)-bisimulation signature of node \( u \in N \) is the pair \( \text{sig}_k(u) = (p\text{Id}_k(u), L) \) where:

\[
L = \begin{cases}
\emptyset & \text{if } k = 0 \\
\{(\lambda_E(u, u'), p\text{Id}_{k-1}(u')) | (u, u') \in E\} & \text{if } k > 0
\end{cases}
\]

From this it follows that for each distinct signature we assign a unique number, so the fact can be made as seen in Proposition 1.

**Proposition 1** \( p\text{Id}_k(u) = p\text{Id}_k(v) \) if and only if \( \text{sig}_k(u) = \text{sig}_k(v) \) (\( k \geq 0 \)).

From this it follows that each unique \( p\text{Id}_k \) will be a unique bisimulation partition. So each node’s \( k \)-bisimulation partition block can be determined by its \( k \)-bisimulation signature, which in turn is determined by the \((k-1)\)-bisimulation partition of the graph. This means that in order to compute the \( k \)-bisimulation partition we first need to compute the \( 0 \)-th to \((k-1)\)-th bisimulation partitions.

Luo et al. [LFH+12] describes an I/O-efficient algorithm to compute the bisimulation partitions, we use a parallel approach in which we distribute work among multiple computing nodes.

### 2.3 MapReduce

MapReduce has become a very popular way to develop parallel applications which run on a distributed cluster due to its simple concepts. It was originally proposed by Dean et al. [DG08] who worked at Google. Generally, in a MapReduce algorithm there are two steps: the Map step and the Reduce step. The Map step reads in the input and assigns a key to the input record. The Reduce step receives all values which have the same key and can process these groups.

More formally we get the following:

- **map** \( (k, v) \rightarrow \text{list}(k', v') \)
- **reduce** \( (k', \text{list}(v')) \rightarrow \text{list}(v'') \)

Here we can see that the Map can output a single key-value pair or multiple key-value pairs which do not necessarily need to be the same keys and values. The Reduce step gets a list of values for a key and can, after some computation, output a new list of values. In essence, the input can be broken into smaller chunks and distributed to different computation nodes which can run in parallel and similarly for the Reduce step where the input can be processed once all keys are assigned.

A simple MapReduce example is calculating the in-degree of nodes. In the Map step we process each edge in the input and use the target node as key and 1 as value, these two values are emitted (or outputted) as a pair. Pseudo-code for this map task can be seen in Algorithm **InDegreeMapper**, where
target[\textit{edge}] returns the target node ID of the given edge. In the \textit{Reduce} step we then simply sum all the values per key which can be seen in Algorithm \textit{InDegreeReducer}. This gives us the in-degree of each node.

\textbf{Algorithm} \textit{InDegreeMapper}(key, edge)  
1. \textit{output} \langle \text{target[\textit{edge}]}, 1 \rangle

\textbf{Algorithm} \textit{InDegreeReducer}(key, values)  
1. \textit{output} \langle key, \sum_{i \in \text{values}} i \rangle

\textbf{Figure 2.1:} In-degree example

Figure 2.1 shows an example run of the InDegree MapReduce algorithm, where the edges are in a tuple format of \langle source, label, target \rangle. In the \textit{Map} step we map each target node with the value 1 and in \textit{Reduce} set we then sum the values for each key. This results in the 5 pairs seen at the right in Figure 2.1.

We can start multiple \textit{Map} processes in parallel, each processing a different subset of the input. Once all the maps are completed, we can have multiple \textit{Reduce} processes just as with the \textit{Map} step. In addition, we can already start copying information from mappers to reducers before all the map tasks are done. This saves additional time and further speeds up the running time.

\section{2.4 HADOOP}

There are various implementation of the MapReduce framework available for use. One is HADOOP, which is developed as an open-source project and is hosted by the Apache foundation. It was originality created by Doug Cutting and Michael J. Cafarella who worked at Yahoo at the time [Caf].

The recent HADOOP version consists of two major components, the HADOOP Distributed File System (HDFS or DFS) and the MapReduce component which reads the data from the HDFS. The HDFS is derived from the Google File System (GFS) [GGL03] which allows for distributed storage of large volumes of data. The MapReduce tasks use the HDFS for input and output of data. Additional advantages of using the HDFS is graceful handling of faults and easy to extend with additional storage.

HADOOP itself is written in Java. All MapReduce jobs developed for HADOOP are also made in Java and tasks need to be submitted to the JobTracker as a jar file. One only needs to develop the logic that goes on in the \textit{Map} and \textit{Reduce} tasks while all the other work, like bookkeeping or reading the input, is done by the HADOOP framework.
2.5 Related work

For computing the $k$-bisimulation partition, most prior research has been done for in-memory computation. One of the more well known algorithms is that of Paige et al. [PT87] where partition of a set are calculated based on a binary relation. A variant with bisimulation as binary relation is then an in-memory algorithm for this problem. The running time of this algorithm would be $O(m \log n)$ where $m$ is the size of the edges and $n$ the size of the nodes. A more recent algorithm with a similar running time was developed by Dovier et al. [DPP04] which could handle larger datasets than that of Paige et al.

However these two algorithms were for the non-localized version of bisimulation. It is often the case with larger datasets that bisimulation partition will be very costly to compute. To solve this, there are various algorithms which compute $k$-bisimulation partition. For instance, Kaushik et al. [KSBG02] has proposed a $A(k)$ index which uses local (or $k$) bisimulation for indexing by making partitions. Work of Yi et al. [YHSY04], Qun et al. [QLO03] and Fletcher et al. [FVW+09] all build on this notion of localized bisimulation to improve certain aspects. One large disadvantage all these algorithms and data structures have is that they are all calculated in-memory. This makes them unsuitable for huge datasets.

One exception to this is the algorithm of Hellings et al. [HFH12] which computes bisimulation using an external-memory approach. However, this only works on Directed Acyclic Graphs (DAGs). So to our knowledge, the only external-memory algorithm for computing localized bisimulation on Directed Graphs (which may contain cycles) is by Luo et al. [LFH+12].

While there has been research on distributed [BO05] and parallel [RL98, SSZ95] computation of bisimulation partitions, these do not utilize the MapReduce framework and hence are not directly applicable to our problem. Other research that does look towards the MapReduce framework for solving graph problems [Coh09, LD10] does not focus on bisimulation partitions.

During this study we experienced various types of skew in both input and output which limited the effects of the parallel computing. There has been much research from the system side to tackle this problem, both in terms of estimating costs of MapReduce system [GARK12] and in ways to modify the system to mitigate the skew effects. This can be done using statistical methods [GARK11, IJL+10] or dynamically [KBHR12, VBBE12]. However, it is also possible to deal with skew from an algorithm design perspective [LS10], which is what we present in this thesis.

2.6 Summary

This chapter discussed the three important notions of $k$-bisimulation, signatures and MapReduce. The first being an equivalence relation which, informally, is assigning each node in a graph to a unique partition block where all members have the same structural properties (e.g., node label, neighbourhood topology, etc.). Secondly, the signatures are a way to encode this structural properties and finally, what MapReduce is and how it works. We also gave a brief explanation what the HADOOP framework is, and what underlying techniques were used in the development. Finally we gave a summary of previous studies related to computing $k$-bisimulation partition as well as literature to estimate and handle skew within a MapReduce system.
In this chapter we describe our base algorithm for computing $k$-bisimulation partitions. We briefly explain our data structures. We then explain the purpose of the tasks which make up the base algorithm: Signature, Identifier and RePartition, and explain what they each do. Finally, we use an example to show how each task works in practice.

### 3.1 Overview

The Base Algorithm is a basic implementation for computing $k$-bisimulation partition without any significant optimizations. It is meant as a baseline and a start for further optimizations. The base algorithm consists of three MapReduce tasks per iteration: The Signature, Identifier and RePartition task.

The following definitions are being used:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pId_0$</td>
<td>The $pId$ assigned initially</td>
</tr>
<tr>
<td>$pId_{old}$</td>
<td>The $pId$ assigned in the previous iteration, in the case of the first iteration this is equal to $pId_0$</td>
</tr>
<tr>
<td>$pId_{new}$</td>
<td>The $pId$ assigned in the current iteration</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>Node ID of an node or source node ID of an edge. The same notation is used since for some algorithms there is no distinction between node and edge</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Target node ID</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Label of a node or edge</td>
</tr>
</tbody>
</table>

These definitions are being used as properties of a node or edge element $e$ as $pId_0[e]$, which retrieves the $pId_0$ value of element $e$.

For the input data for we have two tables: nodes and edges. The nodes table has a structure:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node Id ($\sigma$)</td>
<td>The node ID</td>
</tr>
<tr>
<td>$\perp$ ($\lambda$)</td>
<td>Symbol to check if the the record is a node, useful when joined on the edge table</td>
</tr>
<tr>
<td>$pId_0$</td>
<td>The initial $pId$ value</td>
</tr>
<tr>
<td>$pId_{old}$</td>
<td>The $pId$ assigned in the previous iteration</td>
</tr>
</tbody>
</table>

The edge table:
The source node ID

Label of the edge

The target node ID

The $pId$ assigned in the previous iteration to the target node

The Signature task generates signatures based on a join of the nodes and edges table. The Identifier task then assigns a $pId$ value to each signature and finally the RePartition task partitions the nodes so it can be used in the next iteration. Each task is started as a separate MapReduce task in the HADOOP framework and are ran in sequence, starting with the Signature task. Each task is discussed in-depth in the following sections.

### 3.2 Create signatures task

The Signature MapReduce task does the bulk of the work for the base algorithm. It starts by joining the nodes and the edges table. This outputs for every node ID a list containing the corresponding node and the incoming edges for that node. This is done by the HADOOP as a pre-processing step and it passes the output as input for the mapper where every node ID is the key and the result of the join is the value parameter (or values in this case). We then assign the node’s $pId_{old}$ value to the edges and map all the values, where for a node we map on the node ID ($\sigma$) and the edge on the source node ID (also $\sigma$). The pseudo code for this task can be seen in $\text{SignatureMapper}$.

**Algorithm $\text{SignatureMapper}(key, values)$**

1. $pId_{old} \leftarrow 0$
2. \textbf{for} value $\in$ values
3. \hspace{1em} do $e \leftarrow (key, value)$
4. \hspace{2em} if $pId_{old} = 0$
5. \hspace{3em} then $pId_{old} \leftarrow pId_{old}[e]$
6. \hspace{3em} output $(\sigma[e], (\bot, pId_0[e], pId_{old}[e]))$
7. \hspace{2em} else output $(\sigma[e], (\lambda[e], \tau[e], pId_{old}))$
8. For each group we can get the $pId_{old}$ value of the node and assign it to the edges. Notice that in order to work, the node must be the first value in the values list. This is guaranteed by the join which joins edges to nodes.

The $\text{Reduce}$ step of the Signature-task creates the signatures for each node. This can be seen in Algorithm $\text{SignatureReducer}$. Here we build the signature by concatenating for each edge label and $pId_{old}$ values according to Definition 3.

**Algorithm $\text{SignatureReducer}(key, values)$**

1. $pId_0 \leftarrow \text{nil}; S \leftarrow \emptyset; n = \text{nil}$
2. \textbf{for} value $\in$ values
3. \hspace{1em} do $e \leftarrow (key, value)$
4. \hspace{2em} if $e$ is an edge
5. \hspace{3em} then $S = S \cup \{(\lambda[e], pId_{old}[e])\}$
6. \hspace{3em} else $pId_0 \leftarrow pId_0[e]$
7. $S' \leftarrow [x | x \in S]$
8. sort $S'$ lexicography, first on $\lambda$ then on $\sigma$
9. signature $\leftarrow (pId_0, \text{concatenate}(S'))$
10. output $(key, (\bot, pId_0, \text{signature}))$

8
3.3 Assign identifiers task

Due to the nature of distributed programs, shared memory is usually a bottleneck which is why HADOOP has a ‘Shared Nothing’ design. However, this makes it not as trivial to assign unique $pId$ values to nodes in a distributed way. We solve this by mapping the nodes based on the signature. This will distribute the nodes such that nodes with the same signature will be grouped and end up in the same reducer. This can be seen in IdentifierMapper.

**Algorithm** IdentifierMapper$(key, value)$
1. output $(\text{signature}[value], (\text{key}, \bot, pId_0[value]))$

Now that the nodes are grouped on signature we can assign a $pId_{\text{new}}$ value to each node. Since we cannot share a counter between reducers, we do this by defining a range per reducer with length $n$. This ensures that each reducer can resolve the collisions, because in the worst case each node has a different signature and is mapped to the same reducer. The reducer algorithm can be seen in IdentifierReducer.

**Algorithm** IdentifierReducer$(key, values)$
1. $pId_{\text{new}} \leftarrow 0$
2. if seen $key$ before // $key$ is the signature
3. then $pId_{\text{new}} \leftarrow$ previously assigned $pId$
4. else $pId_{\text{new}} \leftarrow$ generate new $pId$
5. for node $\in$ values
6. do output $(\sigma[node], (\bot, pId_0[e], pId_{\text{new}}))$

We now have assigned a unique $pId$ for each signature and assigned it to the correct nodes. So in essence we are done. However, to use the output of this iteration for the input of the SignatureMapper we cannot properly join the newly obtained nodes onto the edges. In order to do the map-side join, the keys and the partitions need to be the same for both nodes and edges, but the current input is partitioned on signature and not node ID.

In order to fix this, we have another task, the RePartition-task which solves this.

3.4 Re-Partition nodes task

Since the current nodes have already the node ID as first value we can simply read the records in and directly output them. This can be seen in RePartitionMapper and RePartitionReducer. The HADOOP framework will do the partitioning and sorting for us.

**Algorithm** RePartitionMapper$(key, value)$
1. output $(key, value)$

**Algorithm** RePartitionReducer$(key, values)$
1. for value $\in$ values
2. do output $(key, value)$

Once completed, the output is now in the correct partitions and sorted so it can be used as input for the SignatureMapper. So a simple observation is that for the last iteration we do not need the RePartition-task, which means one less in the total number of runs.
3.5 Example run

Using the graph shown in Figure 3.1 we get the input shown, seen in the left most tables of Figure 3.2, where the top two tables are the nodes and are encoded as a tuple containing \((nID, -, pId_0, pId_{old})\), where \(-\) is used instead of \(\perp\). The bottom two tables are the edges and are encoded as \((\sigma, \lambda, \tau, 0)\). Note that the edges have a \(pId_{old}\) value of 0.

In Figure 3.2 we can see that the first step is the Map-Side-Join where the edges are joined on the nodes. This created the groups as seen in the second column of the Join + Map part. The groups then get assigned a \(pId_{old}\) value based on the node in the group. The updated values are coloured red. They are then mapped on the \(\sigma\) value and passed to the reducer. The Reduce step will then output the nodes with the generated signatures. It will also output the number of nodes so we can use it to create the ranges in the next task.

In Figure 3.3 shows the process of assigning a \(pId\) value to the signatures. In the Map step we hash the signature and assign a reducer to each node. (The first reducer is denoted as \(r_1\) and so on.) In the Reduce step we then assign the \(pId_{new}\) value based on the range assigned to the reducer. This can be seen where the second reducer starts with 7 (or \(n + 1\)) since it’s range is \(n + 1\) to \(2n\). It can also be seen that it outputs partitions which do not correspond to the initial nodes dataset.

This re-partitioning can be seen in Figure 3.4, where we simply map on the node ID and output the values. It now matches the input again so we can use it again as input for the Signature-task.

If we then run the same tasks again for a second iteration we see the same thing happening again in the tasks. The only difference is now that we have \(pId_{old}\) value for the nodes in the input of the Map step of the Signature-task so we do not have to set it since it is already present. This can be seen in Figure 3.5 and Figure 3.6. Also if only two iterations are executed, the RePartiton-task does not need to be executed, but for the sake of a complete example, this can also be seen in Figure 3.7.

3.6 Summary

We have described our encoding of graphs and the three tasks involved in the base algorithm for computing the \(k\)-bisimulation partition. The Signature task generates a signature for each node using a map-side join on the edge table. The Identifier task then assigns each signature a unique \(pId\) value and finally, the RePartition task re-partitions the nodes so they can be used in the next iteration. We then show with a running example how each task processes the data.
3.6 SUMMARY

Nodes:
- 1 - 1 0
- 2 - 1 0
- 3 - 2 0
- 4 - 2 0
- 5 - 2 0
- 6 - 2 0

Edges:
- 3 ℓ 1
- 1 w 2
- 2 w 2
- 5 ℓ 2
- 4 ℓ 3
- 1 ℓ 4
- 2 ℓ 6

Figure 3.2: Signature-task, first iteration

Figure 3.3: Identifier-task, first iteration

Figure 3.4: RePartition-task, first iteration
CHAPTER 3. BASE ALGORITHM FOR COMPUTING K-BISIMULATION

Figure 3.5: Signature-task, second iteration

Figure 3.6: Identifier-task, second iteration

Figure 3.7: RePartition-task, second iteration (if executed)
Optimizations for dealing with skew

Skew is ubiquitous in real world data. In this chapter we present two optimizations to the Base algorithm, presented in Chapter 3, where we mitigate some of problems which occur with skewed partition block sizes. In the first, optimization A, we deal with skew in the signature distribution by assigning a fixed $pId$ to certain signatures so we can assign them at random reducer. Distributing signatures will balance the load more evenly among reducers. In the second, optimization B, we deal with skew by loosening some of the restrictions set in the base algorithm, which helps to balance load more efficiently. Using an extra MapReduce task we correct some of the results from looser restrictions.

4.1 Optimization A: Assigned $pId$ values

In this section we describe our first optimization, which still follows the solution of the base algorithm but with a small addition where we store some additional information which we later use for eliminating skew.

4.1.1 Solution

When analysing the data we found that there is some degree of skew in the partition block sizes. Further analysis showed that there are some partition blocks which contain a large number nodes. In the $IdentifierReducer$ task we can see that we have the signature as key value. Since a partition block is associated with a signature, which in turn is equivalent to a $pId$, it becomes clear that each block is processed by a single reducer process. So when we have a block with a large number of nodes, the $Reduce$ task will take some time to complete and thus will increase the overall running time.

When inspecting the data further we also observed that there are only a few partition blocks which are extremely large. So a solution is to treat these blocks differently, but first we need to know what signatures are associated with these blocks. We do this by letting each $SignatureReducer$ task output a count of nodes for each signature next to the regular output. Since this is a local count, we then merge all the local counts into a global count. From this we pick the $f$ largest partition blocks and assign a $pId$ value to it and send them as configuration items to the Identifier task. Since these signatures are assigned beforehand, it no longer requires that for those signatures that they need to be processed by the same reducer. So what we do is for the assigned signatures, the associated nodes
can be distributed randomly. This will even out the load per reducer and thus reducing the total running time.

The Signature task now consists of the original Map step \((\text{SignatureMapper})\) and the updated Reduce step as seen in \(\text{AssignedSignatureReducer}\). We now have a \(\text{Scount}\) which is a map containing the signatures as key and the count as values. With each creation of a signature, the map is incremented for that signature.

HADOOP also supports cleanup functions, which are called when the reducer has successfully completed. At this point we output the \(f\) highest counted signatures with the count as seen in \(\text{AssignedSignatureCleanup}\) using a Priority Queue where the priority is the count and the value is the signature.

In the following sections, code that is marked with red are changes to the base algorithm.

**Algorithm** \(\text{AssignedSignatureReducer}(\text{key}, \text{values}, \text{Scount})\)

1. \(\text{pId}_0 \leftarrow \text{nil} ; \text{S} \leftarrow \emptyset ; n = \text{nil}\)
2. for value in values
3. do \(e \leftarrow (\text{key}, \text{value})\)
4. if \(e\) is an edge
5. then \(S = S \cup \{ (\lambda[e], pId_{old}[e]) \}\)
6. else \(pId_0 \leftarrow pId_0[e]\)
7. \(S' \leftarrow \{x | x \in S\}\)
8. sort \(S'\) lexicography, first on \(\lambda\) then on \(\sigma\)
9. signature \(\leftarrow (pId_0, \text{concatenate}(S'))\)
10. \(\text{Scount}[	ext{signature}] \leftarrow \text{Scount}[	ext{signature}] + 1\)
11. output \(\langle \text{key}, (\bot, pId_0, \text{signature}) \rangle\)

**Algorithm** \(\text{AssignedSignatureCleanup}(\text{Scount}, f)\)

1. \(Q \leftarrow \text{Priority Queue}\)
2. for signature \(\in \text{Scount}\)
3. do \(Q\).put(\(\text{Scount}[\text{signature}]\), signature)
4. for \(i\) in 0 to \(f\)
5. do output \(\langle Q\).pop()\)

Now that we have the \(f\) highest counted signatures per reducer, we aggregate and again pick the \(f\) highest counted signatures. This is then added to the configuration of the Identifier task as the assigned signatures. Since we must now make the distinction between assigned and not-assigned signatures and where to distribute them we introduce a Partitioner in addition to the previous \(\text{IdentifierMapper}\) and the new \(\text{AssignedIdentifierReducer}\). This is also part of the HADOOP framework and is responsible for sending mapped values to reducers. Normally this is done by hashing the key and then using a modulo operation with the number of reducers. However, we want to distribute the assigned signatures at random and the others default. This is seen in \(\text{AssignedIdentifierPartitioner}\).

Finally we have the new \(\text{AssignedIdentifierReducer}\) which first check if the signature is assigned. If so, that \(\text{pId}\) value is used, if not then the reducer is the same as before: we generate a new \(\text{pId}\) if we have not yet seen the signature or else use the old one. The RePartition task is as it was before.

**Algorithm** \(\text{AssignedIdentifierPartitioner}(\text{key}, \text{values}, \text{numberOfReducers})\)

1. \(\text{pId}_{\text{new}} \leftarrow 0\)
2. if \(\text{key}\) is assigned
3. then return random value between 1 and \(\text{numberOfReducers}\)
4. else return \#key \(\mod\) \(\text{numberOfReducers}\)
Algorithm AssignedIdentifierReducer(key, values)
1. $pId_{\text{new}} \leftarrow 0$
2. if key is assigned
3. then $pId_{\text{new}} \leftarrow \text{assigned pId}$
4. else if seen key before // key is the signature
5. then $pId_{\text{new}} \leftarrow \text{previously assigned pId}$
6. else $pId_{\text{new}} \leftarrow \text{generate new pId}$
7. for node $\in$ values
8. do output $\langle \sigma[\text{value}], (\perp, pId_0[e], pId_{\text{new}}) \rangle$

4.1.2 Example run

Consider the graph seen in Figure 4.1. We can see that the nodes 3, 4, 5 and 6 all share the same topological features and are therefore bisimilair to each other, so the signature for those nodes will be equal. When calculating the bisimulation partition for this graph, there will be 3 partition blocks: one with node 1, one with node 2 and one with nodes 3, 4, 5 and 6. This means we will have a skew in the partition sizes.

![Figure 4.1: Graph with skewed partition sizes](image)

What we want to do is detect which signatures will be the ‘heavy hitters’ and assign a fixed $pId$ value to it beforehand. In this example we will set only a single signature fixed, so $f = 1$ where $f$ is the number of fixed $pId$ values. In Figure 4.2 we can see that next to the regular output we also output the $f$ largest signatures which have the highest node count. Since the first reducer has three nodes to process, each with a distinct signature, we output an arbitrary signature with the count. For the second reducer, we can see that the count was 3 for signature 2, $\{(\ell, 1)\}$. We then aggregate the values and we get the $f$ largest count, which was 2, $\{(\ell, 1)\}$ and we assign 1 as the $pId$. To compensate, the reducer ranges are offset with the value of $f$. So the first reducer in this example has a range of 2 to 7 and second 8 to 14. So the input values are mapped based on signature as seen in Figure 4.3, and if the signature was fixed they are random. We can see that where node 4 is at the second reducer, and 3, 5, and 6 are at the first. All the other signatures are processed as before.

Finally, we run the RePartition task to put the nodes in the correct format for the next iteration. This has not changed and can be seen in Figure 4.4 for this example.
CHAPTER 4. OPTIMIZATIONS FOR DEALING WITH SKEW

Figure 4.2: Signature-task

Figure 4.3: Identifier-task with one assigned signature

Figure 4.4: RePartition-task
4.1.3 Limitations

While this solves the problem of the skew in the Identifier task, there are a couple of problems with this method. First, the number of assigned signatures ($f$) needs to be set at the start of the MapReduce task. This requires some insight into the amount to skew present in the dataset. So, initial analysis is needed to guarantee optimal running times.

Furthermore, while the local count is accurate, when merged, the global count might not be. So in some special cases we might not make the correct decision as to what signature needs to be fixed. Consider the example shown in Figure 4.5, here we have 5 reducers which each processed nodes which resulted in two different signatures. All reducers had one node which had signature 1 ($s_1$) and another signature with more than one node. So if the number of fixed signatures is 1, each reducer will output the signature with the highest count, which will never be $s_1$. So when merged we never see $s_1$ in the result. However, if we look globally, we see that $s_1$ was actually the signature with the largest partition. Thus while we assign $s_4$ a fixed $pId$, we should have actually done that with $s_1$.

![Figure 4.5: Special merge case](image)

Luckily, this only occurs when the data is very uniformly distributed on nodes, i.e., when the number of nodes is small and the number of assigned signatures is very small. From our observation we have seen that in real world-data that, even locally on the reducers, these large partitions still dominate the result. Nonetheless, the merged values must be considered an estimate and not the real count per signature.

4.2 Optimization B: Additional merge task

In this section we present our second optimization for dealing with skew. We loosen some of the restriction we set in the base algorithm to better distribute nodes and introduce another task to make sure we get the correct result.

4.2.1 Solution

One of the reasons skew is present at the Identifier step is because we want to process nodes with the same signature in the same reducer to prevent that same signature is assigned multiple $pId$ values. Another optimization is to loosen this restriction and fix any $pId$ values which should have been the same. However this requires another task, the Merge task.

Again, what we want to do is distribute the larger partition blocks among multiple reducers to even the load. What we do for this optimization is look at the size of the partition blocks in the last iteration. We do this because partition blocks will never grow, only split. This means that the last iteration block size is a good indication what the current block size will be. We therefore introduce
\(P_{\text{old}}\) and \(P_{\text{new}}\) where \(P_{\text{old}}\) holds the partition block size for a signature in the last iteration and \(P_{\text{new}}\) the new partition block size. For the first iteration we define \(P_{\text{old}} = n\) where \(n\) is the total number of nodes in the dataset.

So the first change we made is that the SignatureMapper now also appends the \(P_{\text{old}}\) to the nodes and edges it processes. The same for the SignatureReducers so there is no real significant changes to the Signature task. The Identifier \textit{Map} task also now passes the partition block count.

Just as with the assigned signatures we introduce a partitioner for the Identifier task which is responsible for where the outputted data from the \textit{Map} to a \textit{Reduce} task. We define a threshold \(m_{\text{rload}}\) to what the partition block size needs to exceed in order to be distributed among multiple reducers. When we have a uniform dataset with no skew, we can assume that the load of each reducer should be \(\frac{\text{number of nodes}}{\text{number of reducers}}\) and where all reducers finish at the same time. So for the threshold is defined as the most optimal distribution of reducers, so \(m_{\text{rload}} = \frac{\text{number of nodes}}{\text{number of reducers}}\). Partition blocks which have a size of greater than the threshold will be distributed among multiple reducers, the number of ‘extra’ reducers required, \(r\), is defined as \(r = \frac{P_{\text{old}}}{m_{\text{rload}}}\). So the final partition code can be seen in \textit{MergeIdentifierPartitioner}, where we randomly distribute nodes of partition block exceeding \(m_{\text{rload}}\) among more nodes.

\begin{algorithm}
\begin{algorithmic}
\Procedure{MergeIdentifierPartitioner}{key, values, numberOfReducers}
\State \(P_{\text{old}} \leftarrow \text{partitionSize}[\text{value}]\)
\State \(m_{\text{rload}} \leftarrow \frac{\text{number of Nodes}}{\text{numberOfReducers}}\)
\If{\(P_{\text{old}} > m_{\text{rload}}\)}
\State \(r \leftarrow \frac{P_{\text{old}}}{m_{\text{rload}}}\)
\State \text{return} \((\#\text{key} + \text{random value between} \ 0 \ \text{and} \ r) \ \text{mod} \ \text{numberOfReducers}\)
\ElsIf{\(\text{seen} \ \text{key before} // \text{key is the signature}\)}
\State \text{return} \((\#\text{key} \ \text{mod} \ \text{numberOfReducers})\)
\EndIf
\EndProcedure
\end{algorithmic}
\end{algorithm}

Most of the significant changes are made to the Identifier \textit{Reduce} step and RePartition \textit{Map} step. First, we want to know the (local) partition block size, so in order to add that to each node, we first need to count the nodes per block. So we introduce a queue \(Q\) to which we add all the nodes (for each reducer) and increment a counter \(c\). From this we output all the values in \(Q\) and add the value of \(c\) as the partition count. Finally, in order to create the mapping we need to know what \(p\text{Id}\) the reducer assigned to what signature. So for each signature (key) we also output the assigned \(p\text{Id}_{\text{new}}\) value together with the partition size \(c\).

\begin{algorithm}
\begin{algorithmic}
\Procedure{MergeIdentifierReducer}{key, values}
\State \(p\text{Id}_{\text{new}} \leftarrow 0, \ c \leftarrow 0\)
\State \(Q \leftarrow []\)
\If{\text{seen} \ \text{key before} // \text{key is the signature}}
\State \text{then} \(p\text{Id}_{\text{new}} \leftarrow \text{previously assigned} \ p\text{Id}\)
\Else
\State \(p\text{Id}_{\text{new}} \leftarrow \text{generate new} \ p\text{Id}\)
\EndIf
\For{\text{node} \in \text{values}}
\State \text{do} \ Q.\text{append}(\text{node})
\State \(c \leftarrow c + 1\)
\EndFor
\For{\text{node} \in Q}
\State \text{do} \ output \ \langle \sigma[\text{node}], (\bot, p\text{Id}_0[e], p\text{Id}_{\text{new}}, c) \rangle\)
\EndFor
\State \text{output} \ \langle \text{key}, (p\text{Id}_{\text{new}}, c) \rangle\)
\EndProcedure
\end{algorithmic}
\end{algorithm}

Now that we have the local mapping per reducer, we need to merge this into a global mapping so we can guarantee that we have unique \(p\text{Id}\) values. We do this by introducing a new task: the Merge task. What it does is to read in all the outputted signature mappings and merges this into one. Since some signatures have multiple \(p\text{Id}\) values, we pick one of the local \(p\text{Id}\) values as global \(p\text{Id}\). We then
output the local pId values with the global pId values to finalize the map. The pseudo code for this can be seen in MergeMapper and MergeReducer.

**Algorithm** MergeMapper(key, value)
1. output ⟨key, value⟩

**Algorithm** MergeReducer(key, values)
1. QpId ← {}, total ← 0
2. for value ∈ values
3. do QpId ← QpId ∪ {pId[value]}
4. total ← total + partitionCount[value]
5. pId_global ← some value from QpId
6. for pId_local ∈ QpId
7. do output ⟨pId_global, (pId_local, total)⟩

With the HADOOP framework, every reducer outputs to a separate file. At the end of the Merge task we merge all the separate files into one single map by simply concatenating all the files. Now that we have a map from local to global pId values, we can correct them. We then read in the pId map at the Map step of the RePartition task so we have the data available at each Map process. So when mapping a node we simply look up the local pId value and the partition size and output the global one to finalize the pId value. This can be seen in MergeRePartitionMapper, the reduce remains the same as it was before.

**Algorithm** MergeRePartitionMapper(key, value, pId_map)
1. pId_global, P_new ← pId_map[pId[value]]
2. output ⟨key, (⊥, pId0[value], pId_global, P_new)⟩

While we introduce a new task which has additional overhead, we gain the option to better distribute the nodes among reducers which should eliminate the skew and hopefully justifies the extra running time of the Merge task.

### 4.2.2 Example run

For this example we use the same graphs as before, which can be seen in Figure 4.1. As before we have four nodes (nodes 3, 4, 5, and 6) which fall in the same partition and two nodes which are both in their own partition. In Figure 4.6 we can see the Signature task which is the same as the base algorithm with the exception that we now add a partition count (P_old). Since this is the first iteration, we set this at 0.

In Figure 4.7 we can see that the reducer not only outputs the nodes but also the local pId mapping with the signatures. Furthermore we update the nodes with the local count. While we normally would partition on the P_old value, since this is 0, we simply use n that we get in the Signature task.

From the Signature task we will do the merge on the local pId values which were outputted. We can see this in Figure 4.8, where we key on signature and in the reduce step we can resolve any multiple pId values we assigned to the same signature. The output is then ready to be merged and used in the RePartition task. This can be seen in Figure 4.9 where we update the local pId values with the ones from the map as well as the P_new value. As a final result, we have the three partitions as before with their size.
CHAPTER 4. OPTIMIZATIONS FOR DEALING WITH SKEW

Figure 4.6: Signature-task

Figure 4.7: Identifier-task with the local pld data

Figure 4.8: Merge-task

Figure 4.9: RePartition-task
4.2.3 Limitations

One of the limitations of optimization B is that after the Merge task we need to synchronize the data. That means that we need to collect all the separate output files and merge them into one by reading them from the distributed storage. The larger the files, and the more there are, the slower this becomes since this cannot be done in parallel.

One small optimization we have done is to only include global $pId$ values which have multiple local $pId$ values. This reduces the file size and thus lowers the total communication costs.

Another option is to not merge the files but ship all the separate files to the maps and merge at the map side of the RePartition task. One downside is that we do that merging multiple times (number of reducers) which extends the running time of the maps. It also increases the total communication costs, although not much, since the data is the same, just the general file overhead.

4.3 Summary

In this chapter we have presented two optimizations to the base algorithm for dealing with skew in partition block sizes. In the first optimization we fix some signatures which are responsible for the heavy skew and distribute them evenly among the reducers. The second optimization deals with skew by relaxing some of the constraints set by the base algorithm and introduces another task to make sure we get the correct results.

For both optimizations we use the running example to show how they work. Furthermore, we described the limitations of the optimizations and when it would be best to apply them.
In this chapter we discuss various datasets with interesting properties and will give for each dataset a brief description. Next we describe the hardware and software that has been used for running the experiments. Finally, an overview of experiments that we performed for the empirical analysis is given, along with a motivation.

5.1 Datasets

We use a number of datasets to test the running time of the algorithm and compare that to the running time of the original algorithm of Luo et al. [LFH+12]. So for the most part the datasets are selected to be the same as Luo et al. with the addition of some extra synthetic datasets for measuring the influence of skew in the degree of nodes.

jamendo

Jamendo is a large repository of Creative Commons licensed music, based in France.\(^1\) The dataset used for testing is based on the RDF (Resource Description Framework) data which contains artists, tracks, records, performances, signals and timelines. Since it is RDF data, that means we can see each triple as an edge in the graph. In total, this dataset counts 486,320 nodes and 1,049,647 edges and is the smallest dataset which we tested on.

linkedmdb

LinkedMDB is a repository of movie information which aims to be the first open semantic web database for movies.\(^2\) It also includes references to other databases and webpages which contain more information. Again, this data is in RDF format and each triple in the dataset is an edge in the graph. In total, this dataset counts 2,330,695 nodes and 6,147,996 edges.

---

\(^1\)http://dbtune.org/jamendo/

\(^2\)http://linkedmdb.org/
bsbm

BSBM (Berlin SPARQL Benchmark) is a generator for creating RDF graphs which has been build around an e-commerce use case in which a set of products is offered by different vendors. The graph was originally used to have data to benchmark a mix of queries that illustrated the search and navigation patterns of a consumer looking for a product. We will be using a graph that counts 8,886,068 nodes and 34,872,182 edges.

dblp

DBLP is bibliography in the field of computer science and is hosted at the Universität Trier, in Germany. Started in the 1980s, the database lists more than 2.3 million (2,160,375 as of 9 Jan. 2013) items. This data too is in RDF format and the dataset we will be using counts 23,000,670 nodes and 50,203,406 edges.

wikilinks

Wikilinks is a graph of wikipedia where each node represents a page and each edge one or more references (hyperlinks) between pages. The data contains just the links (edges), so no further transformations needed to be done. The version we will use counted 5,710,993 nodes and 130,160,392 edges.

dbpededia

DBPedia is also related to wikipedia, but containing more data. The dataset is in RDF format and thus contains more information than the wikilinks dataset. The version we are using is a bit older and thus contains less pages. The dataset counted 38,15,135 nodes and 115,305,444 edges.

twitter

Twitter is a dataset gathered from the micro-blogging service with the same name. The dataset was composed in July 2009 and at the time contained around 41 million users. As of writing, there are more than 200 million active users. Even then, twitter was large and this dataset it is the largest we will be testing for the experiments, counting 41,652,230 nodes and 1,468,365,182 edges.

random

The random dataset is generated by us using a random in and out degree on the nodes. This gives us a uniform dataset with respect to the generation of signatures, since these rely on the children of a node and thus the out degree. The dataset counted 10,000,000 nodes and 200,000,000 edges.

---

3 http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/
4 http://datahub.io/dataset/l3s-dblp
5 http://haselgrove.id.au/wikipedia.htm
6 http://challenge.semanticweb.org/btc.html
The power dataset is generated by us but the out-degree count follows a power law, meaning that there are a high number of nodes with a low out-degree, and few nodes with a high out-degree. This dataset was generating using a Zipfian generator as described by Gray et al.\cite{GSE94}, and in counted 8,386,165 nodes and 200,000,000 edges.

5.2 Environment

The environment used for testing is the SURFsara cluster\footnote{https://www.surfsara.nl/systems/hadoop}, located in Amsterdam. The SURFsara cluster is a made up of 72 nodes in two racks. Of the 72 nodes, 66 are used for processing the data, the other 6 are used for management of the cluster. Each node had an 8 core CPU (AMD Opteron Socket G34 Eight-Core 6128) which has a speed of 2.0 GHz and 64GB of RAM. For storage, each node has 4 2TB hard disks, which amounts to a total of 460 TB of storage which can be used. All server are housed in two racks with each two Top of Rack switches with a backbone of 60 GbE. The cluster has an uplink of 10 GbE where each rack has a 1 GbE switch for management.

5.2.1 HADOOP

This cluster runs HADOOP version 0.20 and can have 528 maps and 528 reducers run in parallel. However, most of the time other users will also be running tasks which claim map and reducer processes. The cluster tries to distribute map and reduce processes evenly among users, however, when a map or reduce task takes some time to complete, it cannot be redistributed to other users until it completes. This, sometimes, causes jobs to wait until maps or reducers free up.

Development was done using the JAVA interface for HADOOP in order to use all the features that framework has to offer. Development was done using Eclipse. Time measurements are extracted from the HADOOP log files and measure the wall clock.

5.2.2 HDFS

The HDFS (HADOOP Distributed File System) is an open-source distributed file system which is similar to that of the Google File System \cite{GGL03}. It has a default block size of 64MB which are replicated to different nodes, this allows for graceful recovery of hardware failures. These blocks also allow for splitting large data among multiple nodes, thus storing large amounts of data.

Due to distributed nature, the file system cannot be mounted in an operating system. This means that reading and writing data to the file system requires interaction using the HADOOP command-line tool.

5.3 Experiments

The experiments consist of running the algorithms for the different datasets and measuring the time from which the job was started, this is not necessary when the job was submitted due to resource sharing, but we measure from the start of the first map task. This removed the extra ‘waiting’ that
sometimes occurs when running a job. The time itself is based on that of the wall clock. For all experiments a \( k \) value of 10 is used to match the results produced by Luo et al. \([LFH+12]\).

One other thing due to the shared cluster is that results may be influenced by others. In order to minimize this, we run each experiment 5 times and remove the lowest and highest running times and average the others. This should give a better representation of the actual running time.

More information regarding the source-code and its invocation can be found in Appendix A.

### 5.3.1 Skew analysis

In order to see where the skew in the data occurs we will be looking at various metrics for all the datasets. First we will look at the input distribution for each \textit{Reduce} process, this should indicate if there is any skew in the input. Secondly we will look at running time of the \textit{Reduce} processes. We would expect that a larger input size matches a higher running time.

We will do this by running all three algorithms from Chapters 3 and 4 on all the datasets just a single iteration. This should give an insight how the data is being distributed.

### 5.3.2 Parameter fitting

Running time is dependent on a number of factors. First, the number of \textit{Reduce} processes matters greatly. If there are too few reducers, the algorithm does not take full advantage of parallel processing. If there are too many reducers, the overhead of initializing and communication will dominate the algorithm and might even take longer than with fewer \textit{Reduce} processes. In order to get the optimal number of \textit{Reduce} processes we will run the fixed signature algorithm with increasing number of \textit{Reduce} processes. This number will be calculated by taking the minimum number of reducers for \textsc{Hadoop} to correctly process the datasets and then run with increasing multiplications.

We chose this optimized algorithm over the base algorithm due to possible skew in the input, since with a high amount of skew the bottleneck is the reduce task that takes the longest to complete and not the level of parallelism. Using this optimized version will eliminate some of the skew and thus give a better indication as to what number of reducers is optimal.

Another parameter that needs to be fitted is for the fixed signature algorithm, that of the number of fixed signatures. We will test how varying the number of fixed signatures will influence the running time. We would expect that increasing this number would reduce the running time, however there might be a break-even point where the extra work does not outweigh the reduction in running time. We test this by simply running multiple tests where we increase the number of fixed signatures and see at what value the running time does not significantly increase.

The final parameter we will be analysing is the threshold of the merge algorithm. This value determined when a signature should be distributed among multiple reducers. So in order to see what value of the threshold is most optimal we will take various ratios of the threshold and compare the running time and see when this is lowest. We would expect that with a too low threshold the merging will take much longer and thus will increase the running time, with a too high threshold, the algorithm will do nothing in the merge and will be similar to the base algorithm since nothing will be distributed among multiple reducers.
5.4. SUMMARY

5.3.3 Overall performance comparison

Finally, we look at the overall performance of all the algorithms, including the original algorithm by Luo et al. [LFH*12]. We will compare the running times and check if and when the algorithms are faster. We would expect that for larger datasets the MapReduce approach will yield better results and for heavily skewed datasets with respect to the partitions, the optimized algorithms will be faster.

5.4 Summary

In this chapter we described each dataset that will be used for the experiments and how they were obtained. Furthermore, we gave a description of which version of HADOOP was used and the limitation of the optimizations. Finally, we describe the three major areas we will focus our empirical analysis on. First, we will look what effects skew have and how effective our proposed optimizations are in mitigating the skew. Secondly, we will look at various parameters of all three algorithms and test which give the most optimal results. Third and finally, we will look at the overall performance of the algorithms and how they perform in relation to the I/O-efficient algorithm proposed by Luo et al. [LFH*12].
In this chapter we present the results of the experiments described in Section 5.3 and analyse them to verify the predictions that have been made. The first experiment is how much skew influences the results and where it comes from. We also look into how efficiently the proposed optimizations work in mitigating the skew. Next we look into to what parameters give the most optimal results for each of the algorithms. Finally, we compare the overall running time from the algorithms presented with that of Luo et al. [LFH+12].

6.1 Skew analysis and experiments on optimization strategies

![Image showing skew in reducers per dataset](image.png)

**Figure 6.1:** Skew in the reducers per dataset for the base algorithm
As stated before, we observed an uneven balance in workload for some datasets during the first experiments with the base algorithm. For some datasets we observed that some Reduce processes which took more than 5 minutes to finish while others would finish in only seconds. This would indicates that during the Map step a skewed distribution of keys was produced. To validate this, we plotted the individual input sizes and running times for some datasets and can be seen in Figure 6.1a and Figure 6.1b respectively. Each bar represents the running time per Reduce process and the running times are sorted descending on completion time.

It can be seen that there are some Reduce processes in the Identifier step which get a large input while most of the others are more evenly distributed. Since the Identifier Reduce step receives signatures from the Identifier Map we can see that we have indeed a skewed distribution on signatures, meaning there are some signatures which correspond to a large amount of nodes. This in turn translates to the running time, where we can see the same type of skew. So this means that, for instance with dbpedia, that there are one or two processes which take a considerably long while the others tasks are ‘waiting’ for those to complete.

From this observation we trace back the behaviour to the data. It becomes clear that the data is skewed is many ways, for instance, in Figure 6.2 we show the cumulative distribution of partition block size, i.e., number of nodes assigned to the block, to the number of partition blocks having the give size.

![Cumulative distribution of partition block size](image)

**Figure 6.2:** Cumulative distribution of partition block size (PB Size) to partition blocks having the given size for real-world datasets

We can see that for all the datasets, in particular those which showed skewed running time, that the block size shows a power-law distribution property. This indicates that indeed we have few partition blocks which contain a high number of nodes and many blocks which contain a low number of nodes. This in turn explains why we observed the skew.

In order to mitigate the skew, we proposed two optimizations in Chapter 4. The first optimization that we proposed is where we assign some signatures a pid value beforehand and distribute the nodes with those signatures at random. This should even out the workload since we will balance the number of nodes each reducers processes. We can see the result of this in Figure 6.3, where we again show the input distribution and the time.

![Input distribution and running times](image)

We can see in Figure 6.3a that not much has change in the Signature task, which was to be expected since there is not specific partitioning on the keys. However, in the Identifier task we can see that we have a more balanced distribution and the large spikes are no longer present. Because the Identifier task is more balanced, the RePartition task is therefore also more balanced which is also a welcome consequence. This can also be seen in Figure 6.3b, where we show the running times for each process.
6.1. SKEW ANALYSIS AND EXPERIMENTS ON OPTIMIZATION STRATEGIES

(a) Input size of individual *Reduce* tasks

(b) Running time of individual *Reduce* tasks

Figure 6.3: Skew in the reducers per dataset for optimization A

(a) Input size of individual *Reduce* tasks

(b) Running time of individual *Reduce* tasks

Figure 6.4: Skew in the reducers per dataset for optimization B
There are no longer processes which dominate the others in running time and in turn have a better spread load among all reducers. This in turn should produce a better overall running time.

The second optimization we propose introduces a merge step to correct some of the loosened restrictions of the base algorithm. Because we no longer require that each signature should be processed on the same reducer, we should be able to spread workload better and in turn should result in a more uniform running time for each of the processes. The result of this can be seen in Figure 6.4.

In Figure 6.4a and 6.4b we again see the same thing as with the fixed signatures. Another thing we can observe is that the signatures are even better distributed since the results now show almost a uniform distribution for both input and running time for each Reduce process. Another thing we can observe is that the Merge task has a low running time, so we can already see that the additional task might be worth the cost and improve the overall running time.

In total we can see that both optimizations do improve the workload per reducer and should in turn improve the overall running time of the algorithms.

### 6.2 Parameter fitting

One key aspect of parallel algorithms is the ability to scale. Scale can come in many forms, usually as a parameter of the algorithm. We will look at three key parameters which dictate the performance and analyse which performs best.

**Number of reducers**

One of the main scaling parameters is the number of resources, as we increase this there should be a decrease in the running time. However, there is a limit where adding more resource does not affect the running time or can even have a negative effect. The main scaling component for MapReduce is the number of Map and Reduce processes, since these can run in parallel. Picking the correct number for each dataset is therefore important. If there are too few reducers we do not fully take advantage of parallelization, too many reducers and the overhead will actually increase our running time.

So in order to see what number of reducers is most optimal for our algorithm we ran a series of tests of different datasets with the same algorithm and increasing the number of reducers. From this we can select the most optimal number of reducers. We started out with the minimal number of reducers, which is calculated by dividing the edge dataset size in Megabytes (MB) by the maximum block size for HADOOP, which is 64 MB, and rounding up. We selected the edge table since it is generally larger than the nodes table. For instance, Jamendo has a edge size of $16.432\, MB$ which results in $\lceil \frac{16.432\, MB}{64\, MB} \rceil = 1$, for Twitter we get $\lceil \frac{28.426.706\, MB}{64\, MB} \rceil = 445$. This way we have a number of reducers which depends on the input size.

In Figure 6.5 we see the result of this, where $x$ is defined as $x = \lceil \frac{\text{edge size}}{64\, MB} \rceil$. The results are shown for optimization A, since this algorithm can successfully deal with skew while still being similar to the base algorithm. We do not want skew for these results since more reducers means we need to distribute the input better, and as we have seen without a way to mitigate skew this cannot be done correctly. As for the results, we can see that the most optimal number of reducers is somewhere between 2 and 3 times the minimum number of reducers. We further see that adding more reducers will not necessary improve the running time, where with 4 times the minimum number there is a slower running time than that with 1 time the minimum.
6.2. PARAMETER FITTING

We therefore pick 3 times the number of reducers to be most optimal because it gives us the running time close to that of the most optimal while still providing that extra bit of reducers to fully benefit of parallization.

Number of fixed signatures

In the fixed signature algorithm we have the number of signatures we pre-define and pass along to the identifier task using configuration items. This number must be chosen with care, since too few fixed signatures and we do not compensate for enough skew, too many fixed signatures and we generate too much overhead to gain any advantage.

Like with the number of reducers we simply ran the algorithm a number of times with different datasets and increased the number of fixed signatures. The results of this can be seen in Figure 6.6.

We can see that in most cases around 2 assigned signatures seems to be most optimal or increasing this number has little effect. Only for twitter would it be beneficial to increase the number of assigned signatures further. This is because the Twitter dataset is slightly less skewed than, for instance, DBPedia. However, to keep testing parameters consistent, we have selected 2 assigned signatures for all tests with the fixed signature algorithm.
Threshold ratio

For the Merge algorithm we have set a threshold which the (previous) partition much exceed before we decide to distribute the signature among multiple reducers. It becomes obvious that varying this threshold will have an impact on how well skew is handled. The approach is again to select different values for this threshold and see how it influences the running time.

![Figure 6.7: Running times per dataset with varying threshold ratio](image)

The result can be seen in Figure 6.7, where the ratio is simply multiplied with the threshold, meaning that a lower ratio is a lower threshold and more signatures will be distributed. We can see that varying this threshold has little impact. This is likely due the fact that the larger partition will exceed this threshold since they are disproportionally large and therefore the smaller will most likely never exceed the threshold unless set very low. However, we can see that around 1 times the original threshold we get the most optimal result. This was to be expected since this is the most optimal load for each reducer. Only for heavily skewed datasets such as DBPedia it might be worth increasing the threshold further, but for a more general case we can select our original guess of $\frac{\text{number of nodes}}{\text{number of reducers}}$.

### 6.3 Overall performance comparison

Now that we have determined which parameters work best for most datasets we can compare the overall running time and how the algorithms stack up against the I/O efficient algorithm. We do this by simply plotting the running time of the algorithms for the datasets, this can be seen in Figure 6.8.

During the tests for the Merge algorithm with the Random dataset the Reduce processes of the HADOOP framework gave a timeout which caused the experiments to fail. Further analysis showed that due to the uniform nature of the dataset, almost all signatures ended up in the mapping file which was loaded in the configure step of the task. Because this took significant amount of time, HADOOP kills this task and after a number of retries reports a failure. We estimate that the running time would be considerably high when this timeout would be ignored and would not be a viable solution. However, the base algorithm still performs well even with this type of dataset and since there is no skew there is no real reason to use one of the optimizations.

From Figure 6.8 we can observe a number of things. First of all, our proposed algorithms are indeed faster than the I/O-efficient algorithm with either the base algorithm or an optimization. However, for the smaller datasets this might not be a fair comparison. This is due to that our MapReduce algorithm runs primarily in-memory and does not require external (slow) memory. For the larger
6.3. OVERALL PERFORMANCE COMPARISON

datasets, the parallelization does improve the running time considerably. This is most apparent in the case for the Twitter dataset where we now have a running time which is an order of magnitude faster.

We can also see that overall that both optimization performs very consistently. This is further evidence that we are able to scale well with increasing input sizes. The only exception to this is Twitter. For the Twitter dataset we have a higher running time than all the others. This is most likely due to that we request more reducers than SURFsara cluster currently has. As stated before, we use three times the minimum number of reducers which results in total of 1335 reducers for the twitter dataset. However, the cluster only has 528 reducers which can run at a given time. This means that almost a third of the Reduce processes get queued until other processes are done. This will extend the running time, and seeing as we almost have three times as many reducers are available and it can explain why the running times is about three times as high as with other datasets. This could be a reason why the base algorithm is faster than the merge algorithm, regardless of the skew in the dataset (See in Figure 6.1). We think that the shorter running processes being picked up later while the longer running processes are still running. So instead of waiting for the longer to finish it will pick up another task and therefore spread the workload better among all the available processes. We would expect that if the cluster is being upgraded to have more resources, we would see an increase in running time and better performance of the optimizations.

The final observation we can make is that we are indeed able to mitigate heavy skew in the partition block sizes with our optimizations. We can see that the assigned signature algorithm (Optimization A) performs best in most cases, except for Jamando, Random and Power. For the smaller datasets the extra work seems to cause to much overhead, but as we scale the size of the dataset, the extra works pays off and we have a significant difference in running time compared to the base algorithm. The more general merge algorithm (Optimization B) performs better than the base algorithm for all datasets except Twitter. But as stated before, we would expect that if we were able to scale the cluster we would see in increase in performance and thus would expect that it is indeed faster. As expected, the additional merge task adds too much extra work to mitigate skew compared to the assigned signature algorithm. However, results are still comparable.

To further study the different behaviours among the MapReduce implementation we look at two datasets in particular: DBPedia and Twitter. In Figure 6.9 a plot can be seen that shows the running time per iteration for each of the developed algorithms.
From this figure we can see that in the first four iterations for the Twitter dataset, the base algorithm struggles mostly due to skew in the partition block sizes. However, when looking at both the optimizations for these first iterations we see that we have effectively mitigated the skew. Only after the fifth iteration the merge algorithm takes more time to complete the following iterations. This is due to the map files, which the Merge task processes, are becoming too large and the overhead is no longer negligible. This is also reflected in the overall comparison, where the base algorithm is faster than the merge algorithm for the Twitter dataset. The fixed signature approach does performing equally to the base algorithm after the fifth iteration, which results in an improved total running time.

For the DBPedia dataset both optimizations perform comparably each iteration, where the Merge algorithm is slightly slower. This is also visible in the result for the overall comparison, where both optimizations are able to mitigate skew efficiently and improve the running time significantly.

### 6.4 Experiences

MapReduce solutions take a different way of thinking about tackling problems than ordinary non-parallel solutions. Sharing global data is out of the question and clever tricks need to be devised in order to solve trivial problems. Our algorithms were developed based on previous work, and while the overall principals and steps are similar, simply trying to re-write existing algorithms is not trivial. We developed our solution by starting with simply generating the signatures. Once we had these it becomes more straightforward what needs to be done in order to complete the actual algorithm.

Another thing we experienced using the SURFsara cluster is that since it is a shared cluster, sometimes other people run jobs which request Map or Reduce jobs that take a very long time to finish. When the cluster is not very busy they may end up getting all map or reducer processes which the cluster has, so it might take some time before processes free up. During this time, newly posted jobs need to wait until a process becomes available to them. Therefore it is important to measure from the start of the first Map process, rather than the time you submit the job. Furthermore, results are heavily dependent on how much resources are available and how many users have submitted jobs. So it is important to measure multiple runs of a single experiment to eliminate some of the problems which occur when sharing resources.
6.5 Summary

In this chapter have presented the results of the described experiments from Chapter 5. We have first seen that skew can indeed be mitigated by applying the proposed optimizations. Furthermore, we have shown that the most optimal results can be obtained with three times the minimum number of reducers, which is $\lceil \frac{\text{edge size}}{64.0\text{MB}} \rceil$. The most optimal number of assigned signatures is two and the most optimal threshold is $1 \cdot \frac{\text{number of nodes}}{\text{number of reducers}}$ for most datasets. Finally we have shown that our algorithm performs better with all datasets but the largest difference is with massive datasets such as Twitter, which was the original goal of the project. We have also shown that both optimizations prove effective in mitigating skew in the partition block sizes and improve the running time.
In this thesis we have presented, to our knowledge, the first MapReduce algorithm for computing $k$-bisimulation partitions of big graphs. Furthermore, during our analysis we witnessed a skewed workload in the algorithm execution. To address this, we also presented two optimizations to our algorithm design for dealing with skew in the data to further improve the performance.

An extensive empirical analysis shows that our solution efficiently produces bisimulation partition results for both synthetic and real-world large datasets and that the proposed solution scales well with the MapReduce infrastructure. Furthermore, we showed that the proposed optimizations give a better performance by efficiently dealing with skew.

**Future work**

Some interesting research questions are still open.

**Map-side Join**

One of the limitations of the HADOOP map-side join implementation is that it tries to load both datasets fully into memory. This becomes a problem when there is skew in the input nodes. For the SP2B dataset (which was omitted) there was one node which had a too high in-degree. This caused one file to be relatively large and triggered a heap size error with the HADOOP implementation. A more I/O efficient and less generic solution would solve this problem. One extra bit of information that we have is that the nodes do not contain duplicates, only the edges do. This extra bit of information could be used to work around the limitation of the HDFS.

**Adaptive optimization**

In the results it is clear that the merge optimization seems to work best if the data is skewed but not heavily skewed. The fixed signature optimization seems to work best when there are only a few signatures which dominate the rest. When there is little or no skew, doing the extra work might even negatively impact performance. With some extra statistics (or adapt those used by the local assigned signatures) a choice can be made per iteration which solution might work best and thus adapt to skew even better.
Application in related research problems

Because bisimulation is a fundamental graph problem, many other research topics use bisimulation partition as a key role to solve their problems. Applying our algorithm for processing Big Data can further increase performance and efficiently in other related problems such as graph compression, query processing, data analytics and indexing for XML and RDF databases. We hope that our results can provide new insight in related work.
In this Appendix we provide a brief overview of the source code of the three proposed algorithms and how to apply them.

A.1 Overview

All three algorithms are developed in JAVA using Eclipse and the source code consists of 3 Eclipse Projects. The source code is fully documented and all relevant classes are distributed among a number of packages. We will discuss each package next.

`nl.tue.win`

This package contains the Main class and the Partitioner class used in the optimizations. The Main class for each algorithm is the `KBiSim.java` and when creating a JAR file, should be supplied as such.

`nl.tue.win.reducers`

This package contains the `Reduce` classes for each task, which are named accordingly. They each extend from the `org.apache.hadoop.mapred.MapReduceBase` class and override the relevant method such as `reduce` and `configure`.

`nl.tue.win.mappers`

This package contains the `Map` classes for each task, which are named accordingly. They each extend from the `org.apache.hadoop.mapred.MapReduceBase` class and override the relevant method such as `map` and `configure`.

`nl.tue.win.util`

This package contains various classes which help with processing data. Most notably is the `Composite` class which is used to guarantee and input order at the Signature `Reduce` step.
A.2 Dependencies

The only dependency for the algorithms is HADOOP 0.20 and all of the containing libraries. The .classpath file contains the following libraries:

```
hadoop-0.20/lib/xmlenc-0.52.jar
hadoop-0.20/lib/slf4j-log4j12-1.4.3.jar
hadoop-0.20/lib/slf4j-api-1.4.3.jar
hadoop-0.20/lib/servlet-api-2.5-6.1.14.jar
hadoop-0.20/lib/oro-2.0.8.jar
hadoop-0.20/lib/mockito-all-1.8.0.jar
hadoop-0.20/lib/log4j-1.2.15.jar
hadoop-0.20/lib/kfs-0.2.2.jar
hadoop-0.20/lib/junit-3.8.1.jar
hadoop-0.20/lib/jetty-util-6.1.14.jar
hadoop-0.20/lib/jetty-6.1.14.jar
hadoop-0.20/lib/jets3t-0.6.1.jar
hadoop-0.20/lib/jasper-runtime-5.5.12.jar
hadoop-0.20/lib/jasper-compiler-5.5.12.jar
hadoop-0.20/lib/hsqldb-1.8.0.10.jar
hadoop-0.20/lib/core-3.1.1.jar
hadoop-0.20/lib/commons-net-1.4.1.jar
hadoop-0.20/lib/commons-logging-api-1.0.4.jar
hadoop-0.20/lib/commons-logging-1.0.4.jar
hadoop-0.20/lib/commons-httpclient-3.0.1.jar
hadoop-0.20/lib/commons-el-1.0.jar
hadoop-0.20/lib/commons-codec-1.3.jar
hadoop-0.20/lib/commons-cli-1.2.jar
hadoop-0.20/hadoop-0.20.2-tools.jar
hadoop-0.20/hadoop-0.20.2-core.jar
hadoop-0.20/hadoop-0.20.2-ant.jar
```

All of the listed jar files are shipped in the releases of HADOOP and should be present in the 0.20 version.

A.3 Running the algorithm

In order to run the algorithm, HADOOP needs to be installed on the machine and should be configured correctly to be able to access the HDFS and the JobTracker. Furthermore, it is assumed that the input data is present in the HDFS.

Since we require that the input must be partitioned and sorted for the MapSide-join, we have an initial command which does. It can be invoked using:

```
hadoop jar KBiSimMR.jar /hdfs/input/folder prep.datasetName numberOfReducers
```

This assumes that the input data is located for the nodes and edges table at /hdfs/input/folder/data/nodes_datasetName.txt and /hdfs/input/folder/data/edges_datasetName.txt respectively. The input tables should have one element at each row. For the nodes the format “srcNodeID - labelID 0” is expected per row, for the edges “srcNodeID labelID destNodeID”. Running this
will execute two MapReduce tasks, one for each tables and will output the result in the folders
/hdfs/input/folder/data/datasetName/nodes and /hdfs/input/folder/data/datasetName/edges.

For running the base algorithm, the following command can be used with the base algorithm jar
(named KBiSimMR.jar):

```
hadoop jar KBiSimMR.jar /hdfs/input/folder datasetName numberOfReducers
numberOfIterations
```

The algorithm assumes that the data is located in the folders /hdfs/input/folder/data/datasetName/
nodes and /hdfs/input/folder/data/datasetName/edges and this will be used for the MapSize-
Join in the Signature Task. Furthermore, numberOfReducers and numberOfIterations should be an
integer. The final output after the number of iterations will be located on the HDFS in the folder
/hdfs/input/folder/output.

For running the fixed signature algorithm, the following command can be used with the fixed signature
algorithm jar:

```
hadoop jar KBiSimMR.jar /hdfs/input/folder datasetName numberOfReducers
numberOfIterations numberOfFixedSignatures
```

The numberOfFixedSignatures should be an integer.

For running the merge algorithm, the following command can be used with the merge algorithm jar:

```
hadoop jar KBiSimMR.jar /hdfs/input/folder datasetName numberOfReducers
numberOfIterations thresholdRatio
```

The thresholdRatio should be a float, i.e., 1.5.


