Large-Scale Bisimulation of RDF Graphs

Alexander Schätzle  Antony Neu  Georg Lausen  Martin Przyjaciel-Zablocki
Department of Computer Science
University of Freiburg
Georges-Köhler-Allee 051, 79110 Freiburg, Germany
schaetzle|neua|lausen|zablocki@informatik.uni-freiburg.de

ABSTRACT
RDF datasets with billions of triples are no longer unusual and continue to grow constantly (e.g., LOD cloud) driven by the inherent flexibility of RDF that allows to represent very diverse datasets, ranging from highly structured to unstructured data. Because of their size, understanding and processing RDF graphs is often a difficult task and methods to reduce the size while keeping as much of its structural information become attractive. In this paper we study bisimulation as a means to reduce the size of RDF graphs according to structural equivalence. We study two bisimulation algorithms, one for sequential execution using SQL and one for distributed execution using MapReduce. We demonstrate that the MapReduce-based implementation scales linearly with the number of the RDF triples, allowing to compute the bisimulation of very large RDF graphs within a time which is by far not possible for the sequential version. Experiments based on synthetic benchmark data and real data (DBPedia) exhibit a reduction of more than 90% of the size of the RDF graph in terms of the number of nodes to the number of blocks in the resulting bisimulation partition.

Categories and Subject Descriptors
H.3.1 [Content Analysis and Indexing]: Abstracting Methods; H.4 [Information Systems Applications]: Miscellaneous

Keywords
Semantic Web, RDF, MapReduce, Bisimulation

1. INTRODUCTION
In recent years, we have witnessed an increasing number of RDF datasets [2] made publicly available, where one of the most prominent examples is the LOD (Linked Open Data) cloud [1], a remarkable collection of interlinked RDF datasets. One of the major reasons for the propagation of RDF is its inherent flexibility to represent the full spectrum from highly structured data akin to a relational database to unstructured data as it may be found in social networks. However, the great flexibility of RDF also comes at a cost. Faced with a large-scale RDF dataset with only partially known or even unknown structural properties, no a-priori assumptions on the type(s) of data can be made [9]. This is not unusual as it is quite natural for RDF that a dataset is composed from different data sources (cf. e.g. Billion Triple Challenge dataset1). Having no prior schema or meta information about such a dataset, reducing the size of the graph while keeping its inherent structure is one way to achieve a better understanding and more efficient processing.

In this paper we are interested to apply bisimulation [19] to reduce the size of an RDF graph. Intuitively, the idea is to determine structural similar subgraphs where nodes are considered to be bisimilar if they cannot be distinguished by their outgoing paths. Bisimulation is a fundamental technique which has demonstrated its usefulness in a wide range of application fields [23]. In the context of graphs in general, RDF or similar data models it has already been studied for graph compression to achieve efficient query processing [10], for structure-aware query optimization (e.g. [14, 15, 20, 22, 25]) and automated schema extraction (e.g. [16, 17, 21]). Different from these contributions we are interested in large-scale bisimulation. There already exists a variety of efficient in-memory solutions for bisimulation computation (e.g. [8, 13]) but for large-scale RDF datasets it becomes necessary to develop a solution using external memory. Recently, an efficient external memory based algorithm for bisimulation computation has been presented in [12]. However, this algorithm is designed for directed acyclic graphs (DAGs) and consequently cannot be applied for RDF graphs as they typically contain cycles.

We provide two implementations for bisimulation reduction of RDF graphs, one for sequential execution on a single machine using SQL and one for distributed execution using MapReduce. As our main contribution we demonstrate by several experiments that the MapReduce implementation is an effective approach for bisimulation of large-scale RDF datasets as it scales linearly with the number of the RDF triples, while the SQL based approach has a rather limited scaling behavior. Experiments based on synthetic benchmark data and real data (DBPedia) exhibit a reduction of more than 90% of the size of the RDF graph in terms of the number of nodes to the number of blocks in the resulting bisimulation partition. Independently and parallel to our work, [18] also have proposed a MapReduce-based imple-

---

1 http://km.aifb.kit.edu/projects/btc-2012/
mencer of bisimulation. However, as we will detail later in this paper, our and their work cannot easily be compared because of the different emphasis.

The paper is structured as follows. In Section 2 we start with some preliminaries and a detailed definition of bisimulation for RDF graphs as well as an description of a bisimulation reduction algorithm for single machines. Sections 3 and 4 describe our SQL and MapReduce implementations, respectively, complemented by a comprehensive evaluation. A comparison with related work is provided in Section 5. Finally, we conclude with a short summary and outline of future work in Section 6.

2. BISIMULATION OF RDF GRAPHS

An RDF graph $G = (V, L, E)$ is given by its nodes $V$ (i.e. subjects and objects), a set $L$ of labels (i.e. predicates) and a set of edges $E$ where $E \subseteq V \times L \times V$. The elements of $E$ are called triples and are written as $(s, p, o)$, where $s$ represents a subject, $p$ a predicate and $o$ an object. A binary relation $R \subseteq V \times V$ is called a bisimulation of an RDF graph $G$ if for all $s, s' \in V$, where $(s, s') \in R$, the following two conditions hold:

1. If $s \xrightarrow{p} o \in E$, then $\exists o' \in V : s' \xrightarrow{p} o' \in E \wedge (o, o') \in R$.
2. If $s' \xrightarrow{p} o \in E$, then $\exists o \in V : s \xrightarrow{p} o \in E \wedge (o, o') \in R$.

If a bisimulation $R$ exists and for some $(s, s') \in V$ we have $(s, S') \in R$, then $s$ and $s'$ are bisimilar.

The problem we are interested in is to find the equivalence classes of the largest bisimulation of an RDF graph $G = (V, L, E)$. That is, we want to construct a partition $P = \{B_1, \ldots, B_n\}$ of $V$, i.e. a set of disjoint blocks $B_i$ with $\bigcup_{1 \leq i \leq n} B_i = V$, such that the blocks of $P$ are largest with respect to bisimilarity of their elements. We call such a partition a bisimulation partition. This can be used to build up an reduced RDF graph $G' = (P, L, E')$ preserving the inherent structural properties of $G$ where nodes are the blocks of $P$ and $E' \subseteq P \times L \times P$ is the set of edges. An edge $B_i \xrightarrow{p} B_j$, $1 \leq i,j \leq n$, is introduced in $G'$ whenever there exists an edge $s \xrightarrow{p} o$ in $G$, where $s \in B_i$ and $o \in B_j$.

![Figure 1: Simplified RDF graph (a) and its bisimulation indicated by a reduced representation (b).](image)

Figure 1: Simplified RDF graph (a) and its bisimulation indicated by a reduced representation (b).

Figure 1 can be used to illustrate the formal definitions introduced so far. A given RDF graph is shown in (a) and the corresponding reduced graph in (b). Note that we can reduce the two nodes 2 and 3 to one single node represented by the block $\{2,3\}$, respectively nodes 5, 6 and 9 to one single node represented by the block $\{5,6,9\}$. For any two nodes $s, s'$ in the same block $B$ it is guaranteed that for each path $s \xrightarrow{p} s_1 \xrightarrow{p} s_2 \xrightarrow{p} \ldots$ starting at $s$ there exists a path starting at $s'$ of the same length which contains the same sequence of predicates $s' \xrightarrow{p} s'_1 \xrightarrow{p} s'_2 \xrightarrow{p} \ldots$, where in addition $s_i, s'_i$ are elements of the same block $B_i, i \geq 1$. This property can easily be verified for block $\{2,3\}$ and block $\{5,6,9\}$. Moreover, node 4 cannot form a common block with 2 and 3, as paths originating at node 4 are different from paths originating at node 2 and 3.

Inspired by [6], the algorithm to compute a bisimulation reduction is based on the “naive method” in [13]. In spite of its worst-case time-complexity of $O(|E||V| + |V|^2)$, the practical performance is sufficiently good since typically only a small number of algorithm iterations are sufficient to compute a bisimulation partition. Moreover, as we shall demonstrate in the next sections, this algorithm rather naturally leads to a sequential implementation in SQL and a distributed implementation based on the MapReduce paradigm.\(^2\)

Algorithm 1: Bisimulation reduction algorithm

```
input : RDF graph $G = (V, L, E)$
output: Bisimulation partition, represented by $ID^f$

1 // Initially all nodes are in a single block
2 for $s \in V$ do $ID^0(s) = 0$
3 $count_{old} \leftarrow 0$, $count_{new} \leftarrow 1$
4 $k \leftarrow 0$
5 while $count_{old} \neq count_{new}$ do
6   for $s \in V$ do
7     $sig^{k+1}(s) = \{p, ID^k(o) \mid s \xrightarrow{p} o \in E, p \in L\}$
8     $ID^{k+1}(s) = \text{hash}(sig^{k+1}(s))$
9   end
10  $count_{old} \leftarrow count_{new}$
11  $count_{new} \leftarrow |\{ID^{k+1}(s) \mid s \in V\}|$
12  $k \leftarrow k + 1$
13 end
14 $ID^f \leftarrow ID^k$
```

The structure of the bisimulation reduction algorithm is exemplified by Algorithm 1. The algorithm starts with an initial partition where all nodes are in a single block. It then iteratively splits the blocks to derive a refined partition until it forms a bisimulation partition. This is done using the signature of a node $s$ with respect to a partition $P$, denoted by $\text{sig}(s)$, which is defined as the set of $s$’s outgoing edges to blocks of $P$: $\text{sig}(s) = \{p, B_i \mid s \xrightarrow{p} o \in E \wedge o \in B_i \in P\}$. Hence, a partition is a bisimulation partition iff any two members of every block have the same signature with respect to that partition. Thus, the signature of a node could be used to identify its block. However, as signatures may be large, we use an identifier function $ID : V \rightarrow N$, that indicates to which block a node belongs by computing a hash value from the signature of the respective node $v \in V$. Deduced from that, the partition resulting from $k$ iterations of the algorithm, denoted by $P^k$, can be represented by $ID^k$.

Let $f$ be the number of algorithm iterations required to compute the final partition $P^f$, i.e. the bisimulation partition. If only $k, 1 \leq k \leq f$, iterations are performed, the blocks of the corresponding partition $P^k$ define a so called

\(^2\)The distributed algorithm presented in [6] is based on unbounded asynchronous message passing channels, a software architecture quite different from MapReduce.
\( k \)-bisimulation. Intuitively, the nodes in a block of a \( k \)-bisimulation are equivalent with respect to their paths of length up to \( k \). Continuing our discussion of Figure 1, the RDF graph shown in (a) has a bisimulation \( (b) \) and gives rise to the partitions shown in Table 1. The parameter \( k \) in the table indicates the respective blocks for \( k \)-bisimulation. It is interesting to see, for example, that up to \( k = 2 \) nodes \( \{2, 3, 4\} \) form a block; however after one iteration more this block is split and 4 is assigned to a block different from the block to which nodes 2, 3 are assigned. Note further, that certain blocks become final before the final iteration \( f = 3 \).

Table 1: The table contains the partitions resulting from \( k \)-bisimulation of the RDF graph shown in Figure 1.

<table>
<thead>
<tr>
<th>( s )</th>
<th>( k = 1 )</th>
<th>( k = 2 )</th>
<th>( k = f = 3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sig</td>
<td>ID</td>
<td>sig</td>
</tr>
<tr>
<td>1</td>
<td>{(a, 0)}</td>
<td>1</td>
<td>{(a, 2)}</td>
</tr>
<tr>
<td>2</td>
<td>{(b, 0)}</td>
<td>2</td>
<td>{(6, 3)}</td>
</tr>
<tr>
<td>3</td>
<td>{(b, 0)}</td>
<td>2</td>
<td>{(6, 3)}</td>
</tr>
<tr>
<td>4</td>
<td>{(b, 0)}</td>
<td>2</td>
<td>{(6, 3)}</td>
</tr>
<tr>
<td>5</td>
<td>{(c, 0)}</td>
<td>3</td>
<td>{(c, 1)}</td>
</tr>
<tr>
<td>6</td>
<td>{(c, 0)}</td>
<td>3</td>
<td>{(c, 5)}</td>
</tr>
<tr>
<td>7</td>
<td>{(c, 0)}</td>
<td>3</td>
<td>{(c, 5)}</td>
</tr>
<tr>
<td>8</td>
<td>{(c, 0)}</td>
<td>3</td>
<td>{(c, 5)}</td>
</tr>
<tr>
<td>9</td>
<td>{(c, 0)}</td>
<td>3</td>
<td>{(c, 5)}</td>
</tr>
<tr>
<td>10</td>
<td>{(c, 0)}</td>
<td>3</td>
<td>{(c, 5)}</td>
</tr>
</tbody>
</table>

Table 2: PL/SQL vs. MapReduce-based bisimulation.

| dataset | #triples | \( |P| \) | SQL time (in s) | MapReduce time (in s) |
|---------|----------|------|----------------|-----------------------|
| SP2Bench | 1 M | 1876 | 113 (170) | 92 (170) |
|         | 10 M | 13530 | 1215 (349) | 1293 (349) |
|         | 100 M | 105685 | 20194 (1936) | 21194 (1936) |

3. **BISIMULATION USING SQL**

In this section we are going to describe an implementation of Algorithm 1 using SQL. The main purpose of this task is to motivate the application of MapReduce for computing the bisimulation of large RDF graphs as demonstrated in the next section. To this end we will evaluate the SQL-based implementation on various sizes of RDF graphs produced by the SP2Bench benchmark [24]. For small graphs with size up to 1M edges the SQL-version will give quite acceptable efficiency. This and the simplicity of the SQL procedures required for implementation justifies the following discussion of the SQL-version.

The implementation of the algorithm is realized using Oracle PL/SQL. The major difficulty to be managed is to implement the critical set-operations in line 6 and 7 of the algorithm. This part of the algorithm is implemented by procedure newPartition (cf. Algorithm 2). The hash values are computed using the Oracle function \texttt{ora_hash} which accepts as input values of a table collection type and computes a hash value independent of the order of the elements in the collection. The while-loop of the algorithm is implemented by procedure bisimulation as shown in Algorithm 3.

We use two tables as major data structures. Table \texttt{rdfGraph(Sub, Pred, Obj, objInd)} represents the given RDF graph in the obvious way: by \texttt{objInd} we distinguish those nodes, which do not appear as subjects in the graph and therefore will not receive a signature later on. These nodes remain in partition 0 during computation and thus need a separate treatment in procedure newPartition. Table \texttt{Partitions(theLevel, Node, sigHash)} contains information about all signatures in all partitions. theLevel is used for identifying partitions, incremented by 1 starting from 0 to denote the initial partition. Node identifies nodes and sigHash is a hash value computed from the signature of the respective node for the respective partition.

An element of a signature is a pair of a predicate and the respective block of the partition previously being derived. From each such pair a hash value is computed (cf. line 9 in Algorithm 2). Thus a signature of a node with respect to a partition is represented by a hash value computed from a set of hash values, where to maintain a set of numbers a table collection type denoted numberSet is used (cf. line 8 in Algorithm 2). The iterative procedure to compute a bisimulation partition is shown in Algorithm 3. After initializing table \texttt{Partitions} with the initial partition 0, procedure newPartition is called within the WHILE-loop as many times as signatures change between two consecutive iterations. Any such change triggers a split of a block. Therefore, as termination condition we can compare the number of blocks of two consecutively derived partitions.

We computed the bisimulation partition of RDF graphs produced by SP2Bench with 1 M, 10 M and 100 M triples (i.e. edges). The system infrastructure used was given by Oracle11g installed on a Intel Xeon E5-2640, 6C, 2.5 GHz and 90 GB RAM. Table 2 summarizes the experiments - for reference we already included in brackets the execution times resulting from MapReduce as explained in detail in the next section. We see that MapReduce outperforms the PL/SQL-based version for 10 M, already.

4. **BISIMULATION USING MAPREDUCE**

MapReduce is used for automated parallel and distributed computations on a cluster of machines. It operates on data stored in a distributed file where fault tolerance is achieved by replication. The workflow of a MapReduce iteration consists of three phases: a Map phase, a Shuffle & Sort phase and a Reduce phase (cf. Figure 2). For our implementation we use the popular Apache Hadoop framework\(^3\), an open-source implementation of Google’s MapReduce.

\(^3\)\text{http://hadoop.apache.org/}
In the MapReduce-framework, the user has to implement the map function which receives the data locally available to the machine as input key-value pairs and computes key-value pairs as output. These then are sorted according to the output key and distributed over the network in such a way that all values for a distinct key are located on the same machine. The user-defined reduce function receives a key and all values belonging to this key as input and outputs result-values. The following signatures show the input and output of the map and reduce function, respectively [7]:

map: (inKey, inValue) -> list(outKey, tmpValue)
reduce: (outKey, list(tmpValue)) -> list(outValue)

4.1 Implementation

For each iteration of the bisimulation reduction algorithm, two MapReduce jobs are created. The first one, called update job, computes the new signature and updates the ID for each node while the second one, called counting job, counts the number of distinct IDs. The program terminates, if the number of distinct IDs does not change in two consecutive counting jobs. The following signatures show the input and output of the map and reduce function, respectively [7]:

map: (inKey, inValue) -> list(outKey, tmpValue)
reduce: (outKey, list(tmpValue)) -> list(outValue)

Algorithm 2: PL/SQL-procedure for computation of a new (refined) partition

```sql
create procedure newPartition (x in number) as
begin
insert into Partitions with newRound as (0)
select R.Sub, R.Pred, P.sigHash
from rdfGraph R, Partitions P
where R.Obj = P.Node and (P.theLevel = x-1 or R.objInd = 1)
select x as theLevel, T.Node, T.sigHash
from ( select S.Node, ora_hash(cast(collect(S.hashID) as numberSet)) as sigHash
         from ( select distinct Sub as Node, Pred, sigHash, ora_hash(pred || cast(sigHash as NCHAR(100))) as hashID
                   from newRound) S
         group by S.Node ) T;
end newPartition;
```

Algorithm 3: PL/SQL-procedure for computation of a bisimulation partition

```sql
create procedure bisimulation as
begin
i number := 0; j1 number := 1; j2 number := 0;
begin
select distinct 0 as theLevel, Sub as Node, 0 as sigHash from rdfGraph union
select distinct 0 as theLevel, Obj as Node, 0 as sigHash from rdfGraph where objInd = '1';
while ([j1 <> j2]) loop
i := i+1; j2 := j1;
newPartition(i);
select count(distinct sigHash) into j1 from Partitions where theLevel = i;
end loop;
end bisimulation;
```

Figure 3: Workflow of two iterations.
to count the number of distinct IDs) and node 10 has no outgoing edges (i.e. the signature is empty).

Algorithm 4 outlines the map function of the update job. It receives a triple from the RDF graph as input. After parsing the triple, two output key-value-pairs are written which both have the input triple as value. The key is a custom composite-key consisting of the subject or the object of the triple respectively and a number indicating whether the first part of the key was subject or object in the triple. In the following Shuffle & Sort phase, each reducer then receives all triples of a node s where s is either subject or object. These triples are sorted by the second part of the composite key using the secondary sort functionality of Hadoop, resulting in all triples with s as subject being sorted before those triples with s as object. This enables us to first compute the new signature and ID of s and then output the triples with updated IDs within the same reduce function.

Algorithm 4: update job - map(key, value)

```
input : key: byte offset in input file, can be ignored
  value: a quad (s,p,o, ID^k(o))
1 (s,p,o, ID^k(o)) ← parseQuad(value)
2 emit((s,0), (s,p,o, ID^k(o)))
3 emit((o,1), (s,p,o, ID^k(o)))
```

Algorithm 5 shows the reduce function of the update job which uses the discussed ordering of the triples to first compute the signature and ID of a node s. If the key is subject of the triple, the predicate and the current ID of the object are added to the signature of s. Once the signature is complete, the new ID of s is calculated and used to update the ID in the remaining quads where s is object. The hash function must output the same value for two identical sets of elements independent of the element order. The following counting job counts all distinct IDs and compares it with the previous count. There may be nodes that do not have incoming edges and hence do not appear as an object in a triple. These nodes are not necessary for calculating the signatures of other nodes, however their IDs need to be considered in the counting job to get the correct number of distinct IDs. Therefore, for every node its name and ID is stored in a separate file called “node IDs” in the distributed file system, while the updated RDF graph is stored in the file “graph”. The node IDs are used as input for the counting job, whereas the graph is used for consecutive iterations (cf. Figure 3).

Algorithm 5: update job - reduce(key, values)

```
input : key: composite key (node s, sortOrder)
values: a list of quads [(s,p,o, ID^k(o))]
1 sig^{k+1}(s) ← {} 
2 while value in values ∧ key.sortOrder = 0 do
3 (s,p,o, ID^k(o)) ← parseQuad(value)
4 sig^{k+1}(s) ← sig^{k+1}(s) ∪ {(p, ID^k(o))}
5 end
6 // new signature of s complete
7 ID^{k+1}(s) = hash(sign^{k+1}(s))
8 while value in values ∧ key.sortOrder = 1 do
9 (x,p,s, ID^k(s)) ← parseQuad(value)
10 emit(“graph”, (x,p,s, ID^{k+1}(s)))
11 end
12 // generate separate node ID list for counter job
13 emit(“node IDs”, s, ID^{k+1}(s))
```

4.2 MapReduce Evaluation

The evaluation was performed on a cluster of 10 Dell PowerEdge R320 servers equipped with a Xeon E5-2420 1.9 GHz 6-core CPU, 32 GB RAM, 4 TB disk space and connected via gigabit network. The software installation included Cloudera’s Distribution of Hadoop (CDH4). Again, we used the same benchmark generators as for the Oracle evaluation in Section 3 but with much bigger datasets. For SP²Bench [24] we considered datasets with sizes ranging from 1 million to 1 billion triples (i.e. edges). LUBM [11] is defined by the number of universities which in our evaluation ranges from 50 to 3000. For SIB [3] we generated datasets with the number of users ranging from 1000 to 12000. As these generated datasets are expected to be highly (SP²Bench and LUBM) and moderately (SIB) structured, we also applied our bisimulation implementation on the real-world DBpedia [5] dataset (version 3.7) whose structureness has been shown to be low according to the measures introduced in [9]. Table 3 shows the properties of all considered datasets as

![Figure 4: Workflow of a single update job.](image-url)
As visualized in Figure 5, the runtimes of the implementation are linear w.r.t. the size of the input dataset. Compared to the runtime of the non-parallel PL/SQL implementation we can observe that for small datasets (SP¹Bench 1) the database driven approach performs slightly better due to the initialization overhead of MapReduce for every single job. However, even for relatively moderate sized datasets with only 10 million RDF triples the situation changes and for 100 million triples the MapReduce implementation already outperforms the database approach by an order of magnitude. For small real-world datasets the performance of the database driven approach is sufficient but concerning its superlinear scaling behavior, it wouldn’t be suitable for web-scale datasets.

| dataset    | #triples | #nodes | #int. nodes | time (s) | \(|P_f|\) | %i.n. | %i.n. | f   |
|------------|----------|--------|-------------|----------|----------|-------|-------|-----|
| LUBM       |          |        |             |          |          |       |       |     |
| 50         | 6.89 M   | 1.64 M | 1.08 M      | 399      | 4457     | 0.27  | 0.41  | 7   |
| 100        | 13.88 M  | 3.30 M | 2.18 M      | 598      | 13327    | 0.40  | 0.61  | 7   |
| 200        | 27.64 M  | 6.58 M | 4.34 M      | 919      | 44740    | 0.68  | 1.03  | 7   |
| 300        | 41.29 M  | 9.82 M | 6.48 M      | 1271     | 88587    | 0.90  | 1.37  | 7   |
| 400        | 55.26 M  | 13.14 M| 8.68 M      | 1604     | 135451   | 1.03  | 1.56  | 7   |
| 500        | 69.10 M  | 16.43 M| 10.85 M     | 1925     | 171045   | 1.04  | 1.58  | 7   |
| 600        | 82.95 M  | 19.72 M| 13.02 M     | 2299     | 175728   | 0.89  | 1.35  | 7   |
| 700        | 96.64 M  | 22.98 M| 15.17 M     | 2621     | 151292   | 0.66  | 1.00  | 7   |
| 800        | 110.57 M | 26.29 M| 17.36 M     | 2970     | 102966   | 0.39  | 0.59  | 7   |
| 900        | 124.42 M | 29.59 M| 19.54 M     | 3300     | 144389   | 0.15  | 0.23  | 7   |
| 1000       | 138.32 M | 32.89 M| 21.72 M     | 3608     | 229      | <0.01 | <0.01 | 7   |
| 1500       | 207.33 M | 49.30 M| 32.55 M     | 5302     | 234      | <0.01 | <0.01 | 7   |
| 2000       | 276.43 M | 65.72 M| 43.40 M     | 7136     | 234      | <0.01 | <0.01 | 7   |
| 2500       | 345.66 M | 82.19 M| 54.27 M     | 8990     | 235      | <0.01 | <0.01 | 7   |
| 3000       | 414.61 M | 98.58 M| 65.10 M     | 10934    | 236      | <0.01 | <0.01 | 7   |
| SIB         |          |        |             |          |          |       |       |     |
| 1000       | 12.86 M  | 2.60 M | 1.16 M      | 849      | 177751   | 6.83  | 15.38 | 12  |
| 2000       | 25.39 M  | 4.88 M | 2.27 M      | 1164     | 318809   | 6.54  | 14.02 | 12  |
| 3000       | 37.37 M  | 6.92 M | 3.35 M      | 1731     | 443671   | 6.41  | 13.24 | 13  |
| 4000       | 50.37 M  | 9.03 M | 4.52 M      | 2043     | 566509   | 6.27  | 12.53 | 13  |
| 6000       | 75.33 M  | 12.78 M| 6.76 M      | 2966     | 812312   | 6.35  | 12.03 | 14  |
| 8000       | 100.44 M | 16.29 M| 9.02 M      | 3816     | 1045199  | 6.42  | 11.59 | 14  |
| 12000      | 150.46 M | 22.72 M| 13.50 M     | 4927     | 1495640  | 6.59  | 11.08 | 14  |
| DBPedia     | 198.09 M | 61.06 M| 21.39 M     | 27561    | 5021426  | 8.22  | 23.47 | 34  |

| Table 3: Dataset properties and properties of the final bisimulation partition \(P_f\) for every dataset. Internal nodes (int. nodes) have at least one outgoing edge, i.e. a non-empty signature. %i.n. is the relative size (in %) of \(P_f\) compared to the number of internal nodes, whereas %i.a.n. compares it to the number of all nodes. f indicates the final iteration of the algorithm.

well as an overview of the resulting bisimulation partition sizes and runtimes.

As visualized in Figure 5, the runtimes of the implementation are linear w.r.t. the size of the input dataset. Compared to the runtime of the non-parallel PL/SQL implementation we can observe that for small datasets (SP¹Bench 1) the database driven approach performs slightly better due to the initialization overhead of MapReduce for every single job. However, even for relatively moderate sized datasets with only 10 million RDF triples the situation changes and for 100 million triples the MapReduce implementation already outperforms the database approach by an order of magnitude. For small real-world datasets the performance of the database approach is sufficient but concerning its superlinear scaling behavior, it wouldn’t be suitable for web-scale datasets.

There is a surprising observation regarding the LUBM evaluation. At first the size of the final bisimulation partition (i.e. number of blocks, \(|P_f|\)) rises continuously for datasets with up to 600 universities but then the size begins to decrease and finally stabilizes around 230 blocks. We verified the outcome but didn’t find any entities of any class put together in a block with entities of another class. In fact, we get some very large blocks, e.g. for publications, (under)graduate students, research groups, (full/assistant) professors etc. This can only be explained by a change in the link structure of the generated datasets for higher university numbers allowing the algorithm to put more nodes in a common block without violating the strong bisimulation properties. This also yields a strong reduction of the graph size to less than 0.01% compared to the input graph.

Also interesting is the correlation between the structuredness of an RDF graph and the outcome of its bisimulation. Considering the highly structured datasets, the size of the final bisimulation partition and hence the size of the reduced graph is less than 1% for SP¹Bench and even less than 0.01% for LUBM (cf. Table 3). Furthermore, the number of iterations needed to compute the bisimulation are also rather small (SP¹Bench: 4, LUBM: 7) and remain constant with increasing dataset size. The reduction ratio increases with the input graph being more unstructured. For the moderately structured SIB benchmark the ratio is significantly higher ranging from 11 to 15% (regarding internal nodes only). However, also for SIB the ratio is constantly drop-
Figure 5: Runtimes for benchmark datasets.

ping with increasing number of users. Hence, the nodes of the graph become more similar with increasing dataset size. Likewise, the algorithm also needs more iterations than for SP²Bench or LUBM and the number slightly increases with the dataset size which is not surprising as more iterations yield a more fine-grained partition structure. Finally, the reduction ratio is worst for the loosely structured DBPedia dataset, especially if we only consider the internal nodes since nodes with no outgoing edges have an empty signature, thus lying in the same bisimulation block. This observation directly coincides with the results in [9], thus we conclude that the relative size of the largest bisimulation partition of an RDF graph \( G \) (i.e. the number of blocks) compared to the size of \( G \) (i.e. the number of nodes) is a measure for the degree of structuredness of \( G \).

If we take a more detailed look at the iterations of the algorithm, we can always observe a major increase in partition size, \(|P_k|\), for small numbers of \( k \) and only minor changes with \( k \) close to \( f \), i.e. the final iteration. This is an expected behavior as it is more likely that two non-similar nodes, with respect to bisimulation, differ in their paths with a relatively small length. In other words, it is more unlikely that two nodes are similar up to a path of length \( k \) where \( k \) is close to \( f \), especially if \( f \) is rather large. This is underpinned by Table 4 that illustrates the absolute and relative (compared to the final partition) partition size with increasing number of iterations.

| dataset       | \(|P_k|\) | \(|P_k|/|P_f|\) (in %) | \( k \) |
|---------------|---------|----------------------|-------|
| SP²Bench 1000 | 814     | 0.09                 | 1     |
|               | 412632  | 45.61                | 2     |
|               | 887330  | 98.09                | 3     |
|               | 904604  | 100                  | 4\*   |
| SIB 12000     | 106     | < 0.01               | 1     |
|               | 7122    | 0.48                 | 2     |
|               | 971541  | 64.92                | 3     |
|               | 1103691 | 73.75                | 4     |
|               | 1409641 | 94.19                | 8     |
|               | 1474396 | 98.52                | 9     |
|               | 1496274 | 99.98                | 12    |
|               | 1496527 | 99.99                | 13    |
|               | 1496540 | 100                  | 14\*  |
| DBPedia       | 999700  | 18.12                | 1     |
|               | 3827405 | 76.22                | 2     |
|               | 4908547 | 98.95                | 3     |
|               | 5018873 | 99.95                | 4     |
|               | 5020485 | 99.98                | 5     |
|               | 5020839 | 99.99                | 6     |
|               | 5021242 | ~100                 | 33    |
|               | 5021426 | 100                  | 34\*  |

Table 4: Absolute and relative partition sizes; the final bisimulation partition, \( P_f \), is indicated by \( \ast \).

5. RELATED WORK

Independently and parallel to our work, [18] have proposed a MapReduce-based implementation of bisimulation. While they seem to be mostly interested in a clever handling of skew data, we aim at reducing the overall number of MapReduce tasks required for computing a bisimulation reduction, i.e. our algorithm needs two MapReduce tasks for each iteration compared to three in theirs. To give a rough comparison of the evaluation results of the different implementations, we first have to comment on the Hadoop clusters used. While our cluster consists of 10 nodes each having a 1.9 GHz 6-Core CPU with 32 GB RAM, the cluster described in [18] has a size of 72 nodes each being equipped...
with a 2.0 GHz 8-Core CPU with 64 GB RAM. Our DBPedia dataset consists of 61.06M nodes and 198.09M triples, while theirs consists of only 38.62M nodes and 115.3M triples. Furthermore, they only perform a 10-bisimulation. Our algorithm requires approx. 8000 seconds for a 10-bisimulation of our DBPedia dataset, while the algorithms in [18] have a running time of approx. 7000 seconds for the Base Algorithm and approx. 2000 seconds for the optimized versions on their smaller DBPedia dataset. Due to strongly different cluster sizes and also different dataset sizes, it is certainly not possible to make any definite statements characterizing the efficiency of both approaches. However, we like to conjecture that a reduction of the number of MapReduce tasks seems to have a higher impact on efficiency gains compared to an optimization targeted on skew in data.

6. CONCLUSION

In this paper we demonstrated that the bisimulation reduction of large-scale RDF graphs can be done efficiently using MapReduce. Our algorithm scales linearly with the number of triples and in our experiments it was able to achieve a reduction of more than 90% of the size of the input RDF graph in terms of the number of nodes to the number of blocks in the resulting bisimulation partition. However, for huge graphs even such a reduction might be not sufficient. While bisimulation preserves the structure of the RDF graph, its definition is rather strict and does not consider the specific properties of RDF that can be used to further reduce the size (e.g., publications can have varying numbers of authors). We are currently considering agglomerative clustering to further reduce the number of blocks in a bisimulation partition [4] and plan to develop a distributed MapReduce-based implementation of this approach.

7. REFERENCES