Ulm University | 89069 Ulm | Germany

Faculty for Engineering, Computer Science and Psychology
Institute of Databases and Information Systems (DBIS)

Parallel Structural Graph Summarisation on Graphs of Billion Triples

Computing k-Bisimulation Summaries with the FLUID-based (B,R,S)-Algorithm

Bachelor's Thesis at Ulm University

Presented by:
Jannik Rau
jannik.rau@uni-ulm.de
981730

Supervisors:
Prof. Dr.-Ing. Ansgar Scherp
Dr. David Richerby

2021

Version January 18, 2022

Name: Jannik Rau Student Number: 981730

Statement of Originality

I hereby declare that I have written the thesis by myself, without contributions from any sources or aids other than those indicated. I confirm that this work has not been submitted or published elsewhere in any other form for the fulfillment of any other degree or qualification.

82024 Taufkirchen, 26.11.2021	
....	....
Place and Date	Jannik Rau

Contents

List of Figures

List of Tables

Parallel Structural Graph Summarisation on Graphs of Billion Triples

Computing k-Bisimulation Summaries with the FLUID-based (B,R,S)-Algorithm

Jannik Rau
jannik.rau@uni-ulm.de
Ulm University

ABSTRACT

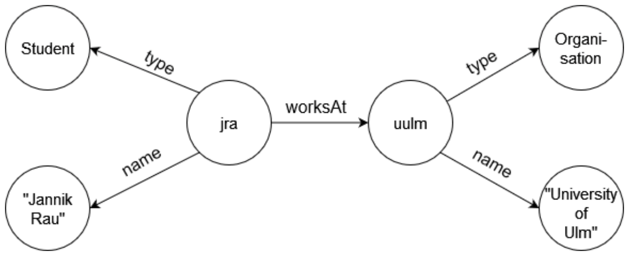
Summarising graphs w.r.t. structural features is an important concept in large graph applications in order to reduce the graph's size and make tasks like indexing, querying or visualisation feasible. The generic (B,R,S) algorithm enables summarising graphs w.r.t. a custom equivalence relation $\sim \subseteq V \times V$ defined on the graph's vertices V. Moreover, $\sim$ can be chained $k \in \mathbb{N}_{>1}$ times, such that the resulting equivalence classes are determined by global information of distance k. This mechanism enables the computation of stratified k-bisimulations, a popular concept in structural graph summarisation. However, so far the algorithm has only been evaluated for equivalence relations including local vertex information of distance $k = 1$. Therefore, a comparison of (B,R,S) against a parallel and a sequential bisimulation algorithm is provided. The performance is compared for a distance of $k = 10$ on four real-world graphs containing $100M$ to $15B$ triples and two synthetic graphs containing $100M$ and $1B$ triples. Moreover, an adjusted version of the (B,R,S) algorithm is provided, as the original implementation makes use of nested data structures, which cause heavy memory issues when using the chaining mechanism. All algorithms perform computations in-memory only. Results emphasise to use parallel algorithms for structural summarisation, as the sequential algorithm takes about 10 hours to compute the 10-bisimulation on a smaller graph of $100M$ triples. Furthermore, for extremely large graphs of $15B$ triples, none of the algorithms was capable of computing the 10-bisimulation due to running out of memory. This stresses the need for external memory solutions for such large graphs. On all other datasets, the generic (B,R,S) algorithm outperforms the respective bisimulation counterpart, indicating that it does not have any disadvantages, though its generic nature.

KEYWORDS

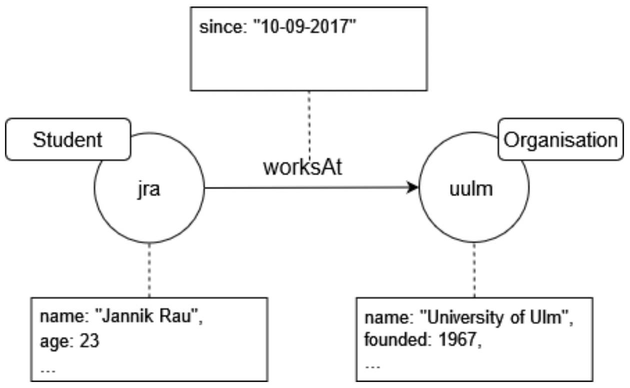
Semi structured data, k-Bisimulation, Structural Graph Summarisation, FLUID, Parallel Computation

1 INTRODUCTION

Graphs, for example in the form of a *Directed, Edge-Labelled Graph*, become an increasingly popular data-modelling paradigm [17]. A common example of a directed, edge-labelled graph is that of an RDF graph [12], which contains information about (real-word) entities – corresponding to the graph's vertices – and their relationships – corresponding to the graph's edges. An RDF graph can be expressed as a set of triples (s, p, o), of which each formulates a *subject-predicate-object* expression.[1] Translating this to a directed, edge-labelled graph, a triple (s, p, o) corresponds to an edge labelled with p going from vertex s to vertex o. Figure 1a shows an example RDF graph[2], which contains information about a student and the university, where he currently works at. Such



(a) Example RDF Graph of a Student-University relationship



(b) Example Property Graph of a Student-University relationship

Figure 1: Example Graphs containing a Student-University relationship.

RDF graphs, also termed as *Knowledge Graphs*, emerged and grew in both, the open-source domain, as well as the enterprise domain [17]. The applications of such graphs are numerous, ranging from search engines, recommendation systems of any kind, fraud detection and business intelligence, to simple information holders. An equivalent variant to that of a directed, edge-labelled graph is a *Property Graph* [17]. It additionally incorporates vertex labels, as well as property-value pairs for both, vertices and edges. Figure 1b shows an example property graph, which contains similar information as the graph in Figure 1a. Using either the concepts of RDF graphs or those of property graphs, one can for example model a university's information at a certain point in time, e.g. all information about students, employees, courses, exams etc. Going further, one could extend this model to include information about different universities of a certain state, country or even continent.

However, as the information contained in the graph increases, so does the graph's size, and potential problems like storing, indexing, querying the graph, or understanding/visualising the graph in general can arise [7]. One technique to approach these problems is *Graph Summarisation* [9]. Depending on the use case, one can summarise the graph with respect to structural features

[1] https://www.w3.org/TR/2014/REC-n-triples-20140225/Overview.html
[2] For simplification purposes, URIs do not follow any schemata in this example.

(e. g., incoming/outgoing paths), statistical measures (e. g., occurrences of specific nodes) or frequent patterns found in the graph [9]. This results in a usually smaller *Summary Graph*, which contains an approximation of or exactly the same information as the original graph. Therefore, the summary graph can be used to overcome the aforementioned problems.

This work focuses on summarisation w.r.t. structural features. Structural summaries, more precisely structural summaries based on equivalence relations, form a partition of the input graph's vertices. Consequently, the summary graph contains precise structural information about the input graph [9]. In contrast, summary graphs based on frequent patterns or statistical measures mostly contain an approximation of the input graph's information [9]. In the existing literature, one can observe many different structural summarisation approaches, which capture different aspects of the input graph [6, 9]. Consequently, many *single-purpose* algorithms exist, which are only designed for computing one specific summary. Motivated by this observation, a generic structural summarisation approach has been proposed [5, 6]. The proposed (B,R,S) algorithm summarises an input graph w.r.t. a custom equivalence relation defined in the formal language FLUID [6]. This mechanism enables summarising graphs w.r.t. structural features using one generic approach, allowing the user to select the specific structural features the resulting summary graph shall contain. Researchers and practitioners settled in the graph domain benefit from this mechanism, as they do not have to research for proper approaches, which summarise a graph according to their needs. Rather, they can define their custom equivalence relation and summarise a graph using the (B,R,S) algorithm.

When producing summaries, numerous structural features and combinations of them can be used. Hence, it is challenging to provide a generic approach, which incorporates all of them. The authors classified potential features according to existing structural summarisation approaches. The first group is comprised of a vertex's *local* information, e.g. the label set of a vertex, the direct neighbours of a vertex or the set of incoming/outgoing edge labels. The second group is comprised of a vertex's *global* information of distance $k > 1$. This includes, e. g., local information about reachable vertices up to distance k or information about incoming/outgoing paths up to distance k. Summarising a graph w.r.t. global information can be achieved by applying any variant of *stratified bisimulation* (Section 3.1.2) on the input graph. Informally, a stratified k-bisimulation groups together vertices, which have an equivalent structural neighbourhood up to distance k. Definitions 2 to 5 provide more formal definitions of stratified bisimulation variants. Several existing structural graph summarisation approaches utilise stratified bisimulation to incorporate global information into the resulting summary graph [6, 9]. Therefore, the FLUID-based (B,R,S) algorithm provides a mechanism of chaining the custom equivalence relation k times, such that the resulting equivalence classes are determined by global information up to distance $k > 1$, which essentially can be used to compute stratified k-bisimulations.

The authors have evaluated the performance of their generic approach concerning summaries that capture local information ($k = 1$) [5]. In this work, the algorithm's performance is evaluated concerning two summaries that capture information up to a distance of $k = 10$. More precisely, the algorithm is compared against two existing summarisation algorithms (1) *Schätzle et al.* [32] and (2) *Kaushik et al.* [19], which are specifically designed for computing stratified bisimulations. All algorithms are evaluated on four real-world and two synthetic datasets. Speaking in terms of RDF graphs, the real-world datasets' sizes reach up to $15B$ triples. Consequently, the main research questions can be formulated as follows:

RQ1 Is it possible to compute 10-bisimulations in an acceptable time on large real-world graphs?
RQ2 How well do the algorithms scale?
RQ3 Which algorithm performs fastest on which dataset?
RQ4 Do the specific bisimulation algorithms have any advantage over the generic algorithm?

To examine these questions, the two existing (native) algorithms have been reimplemented using the same framework, in which the (B,R,S) algorithm is implemented. Moreover, the respective *Graph Summary Models* (Section 3.1.4) have been defined in FLUID, such that the (B,R,S) algorithm summarises the input graph equivalently to the existing algorithms. The reimplemented native algorithms and the (B,R,S) algorithm applied on the two graph summary models are executed on three smaller and three larger datasets for a value of $k = 10$. Both, the set of smaller and the set of larger datasets are comprised of two real-world datasets and one synthetic dataset. The real-world datasets' sizes are $100M$, $150M$, $2B$ and $15B$ triples, whereas the synthetic datasets sizes are $100M$ and $1B$ triples. All algorithms perform computations in-memory only.

One key feature of the (B,R,S) algorithm is its capability of incrementally updating an existing graph summary, in case the input graph has changed (e. g., vertices/edges were added, removed or updated). Furthermore, the (B,R,S) algorithm can infer implicitly stated information (e.g. information described with the RDF Schema vocabulary) on the summary graph. Since the other two algorithms are neither designed for incremental updates nor for inference, these features are omitted.

The following sections are structured as follows. Section 2 provides related work in the domain of graph summarisation. Section 3 define and clarify any preliminaries as well as the methods. Section 4 outlines the experimental apparatus including datasets, experimental procedure, implementations and the applied measures. Section 5 describes the results obtained from the experiments, which are further discussed and interpreted in Section 6. Finally, Section 7 concludes the work.

2 RELATED WORK

Summary graphs can be constructed in different ways. A classification of existing methods into four categories has been proposed by Čeberič et al. [9]. It classifies techniques into *Structural*, *Pattern-mining*, *Statistical*, and *Hybrid* approaches. Structural approaches can be used for improving query evaluation on a graph G, by using the resulting summary graph SG as and index for G [9]. Two common techniques can be distinguished in structural approaches. The first, producing quotient summaries, summarises a graph G w.r.t. an equivalence relation $\sim \subseteq V \times V$ defined on the vertices V of G [7, 9]. The resulting summary graph SG consists of vertices VS, which correspond to the equivalence classes A of the equivalence relation $\sim$. Tran et al. [37] construct a summary graph SG using k-bisimulation as the equivalence relation. Informally, k-bisimulation groups together vertices, that have an equivalent structural neighbourhood up to distance k. A more detailed definition is given in Section 3.1.2. Tran et al. use the bisimulation based summary graph to evaluate data partitioning and different query processing with SG serving as index for the original graph G. Another work that utilises bisimulation to construct a summary graph is the tool ExpLOD [22]. Here, the resulting summary graph SG finds a different application. The authors propose to use SG to understand and explore schemata of interlinked datasets. Hence, ExpLOD can help workers, who add or extract information to/from these datasets. Bisimulation is a popular concept for constructing structural graph summaries [6] and can be found in more works of that domain [11, 14, 26, 27]. However, there are also structural summarisation approaches that determine vertex equivalence only based on local information ($k \leq 1$). Campinas et al. [8] construct a summary graph by determining equivalence based on (1) outgoing edge labels and (2) vertex labels. With the resulting equivalence classes (1) *Attribute-based Collections* and (2) *Class-based Collections*, the authors implement a query recommendation system for SPARQL[3], which facilitate working with heterogeneous datasets in general and especially in case the dataset's schema structure is unknown to the user. SchemEX [23] constructs a three-layered schema-level index for an RDF graph. The third layer groups together vertices v, v', that have the same labels and for every edge (v, w) with label p (and vice versa), there exists a respective p-labelled edge (v', w'), with w and w' also having the same labels. The resulting equivalence classes $[v]_\sim$ are mapped to the sources containing their elements. This mapping can be used to aid query evaluation for certain type of queries, as the mapping provides an entry point and hence decreases the queried target's size. Several other works construct structural summaries by considering a vertex's local neighbourhood [2, 10, 29, 34].

The second structural summarisation technique, producing non-quotient summaries, do not make use of equivalence relations to summarise a graph. Rather, the summary graph is comprised of vertex summaries vs, which group together vertices v of the original graph G according to certain criteria/measures [9]. Hence, the main difference between the techniques' resulting summaries is that in non-quotient summaries a vertex v can belong to zero, one, or multiple vertex summaries vs, whereas in quotient summaries every vertex v has exactly one corresponding vertex summary VS, which is the equivalence class of v under $\sim$ [9]. Early work of non-quotient summarisation includes Goldman and Widom [15], who create a vertex summary vs for every labelled path in the original graph G. A vertex v of G is associated with a vertex summary vs, if it is reachable by the corresponding label path. The summary graph SG is used as a path-index, as well as a tool for understanding the schema structure in semi-structured databases and hence finds application in query formulation and query optimisation. Revisiting the summarisation tool SchemEX [23], its first layer – the *RDF class layer* – consists of vertex summaries vs_{c_j} representing all the classes c_j present in the input RDF graph G. A vertex v of G is associated with a vertex summary vs_{c_j}, iff v is of the corresponding type c_j. Since a vertex v can have multiple types $c_{j1}, c_{j2}, \ldots$, it is possible that v is associated with several vertex summaries $vs_{c_{j1}}, vs_{c_{j2}}, \ldots$ and therefore the index's RDF class layer is considered a non-quotient summary. Kellou-Menouer and Kedad [20] provide a schema extraction approach based on clustering. It first utilises density-based clustering to establish a partition of the vertices into *type sets* T_j. Afterwards, for each T_j a *type profile* $TP_j = \{(label_1, \alpha_1), (label_2, \alpha_2), \ldots\}$ is constructed, consisting of all the edge labels for edges (v, w) and (w, v) with $v \in T_j$ and an associated probability α_i, denoting how likely it is that a vertex $v \in T_j$ has an edge with the respective label. If a type profile TP_{j1} contains all entries $(label_i, \alpha)$ of another type profile TP_{j2} and every α is greater than a certain threshold θ (e.g., $\theta = 1$), then the vertices in T_{j2} are added to T_{j1} to create *overlapping classes*. Clustering can be found in more structural non-quotient approaches [21, 24, 28, 38].

Besides structural graph summarisation, the aforementioned classification is comprised of three more categories. Pattern-mining approaches utilise algorithms to identify frequent patterns in the original graph G, which are then used to construct the summary graph SG [9]. Song et al. [33] construct *d-summaries* to summarise a knowledge graph G. A summary P, which is a graph pattern found in G, is considered a d-summary, iff all the summary vertices $u \in P$ are d-similar (R_d) to all their respective original vertices $v \in V$. Informally, uR_dv iff (1) u and v share the same label and (2) for every neighbour $u' \in P$ of u connected over an edge with label p there exists a respective neighbour $v' \in V$ connected via the same edge label and $u'R_{d-1}v'$. Their definition of d-similarity is very alike to k-bisimulation (Section 3.1.2) and mainly differs in the domain on which it's defined, namely summary vertices and original vertices. Statistical approaches construct summary graphs by considering quantitative properties of the input graph G [9]. The summarisation operation k-SNAP [36] minimises a function based on occurrences of user selected edge labels to produce a summary graph SG, which contains exactly k vertex summaries. In its top-down approach, it starts by initially partitioning the graph based on user selected vertex attributes. Afterwards, the algorithm splits elements (vertex summaries) of the partition based on the aforementioned function, until the partition's size is equal to k. Combining the first step, partitioning vertices by label, and the second step, minimising a function which considers edge labels, k-SNAP can be considered a hybrid approach, combining structural and statistical concepts.

3 PRELIMINARIES AND METHODS

This section first defines the data structure (Section 3.1.1) and the main concepts of the algorithms (Sections 3.1.2 to 3.1.4). Afterwards, the general approach and properties of each algorithm are described and explained, starting with the (B,R,S) algorithm (Section 3.2.1), followed by the algorithms of Schätzle (Section 3.2.2) and Kaushik (Section 3.2.3).

3.1 Preliminaries

In this section, the data structure on which the algorithms operate, as well as the concepts they apply are defined.

3.1.1 Data Structures. The algorithms operate on a multi-relational, labelled graph G.[4]

Definition 1 (Multi-Relational, Labelled Graph). *A multi-relational labelled graph G is defined as* $G = (V, E, \Sigma_V, \Sigma_E)$*, where* $V = \{v_1, v_2, \ldots, v_n\}$ *is a set of vertices and* $E \subseteq V \times V$ *is a set of directed edges between the vertices in* V*. Furthermore, each vertex* $v \in V$ *and each edge* $(u, v) \in E$ *has zero or more associated labels from the finite set* Σ_V *and* Σ_E*, respectively. The mapping of vertex and edge labels is done via the functions* $l_V : V \rightarrow \mathcal{P}(\Sigma_V)$ *and* $l_E : E \rightarrow \mathcal{P}(\Sigma_E)$ *respectively.*

Henceforth, if not further specified every graph G is considered a multi-relational, labelled graph. Figure 2 depicts an example

[3]https://www.w3.org/TR/sparql11-overview/

[4]In fact, the algorithms could also operate on multi-relational property graphs, in which vertices and edges can have additional *property-value*-pairs. However, only edge or vertex labels are used by the algorithms to compute the output. None of the algorithms consider any properties of vertices or edges, which is why multi-relational, labelled graphs are used as the data structure.

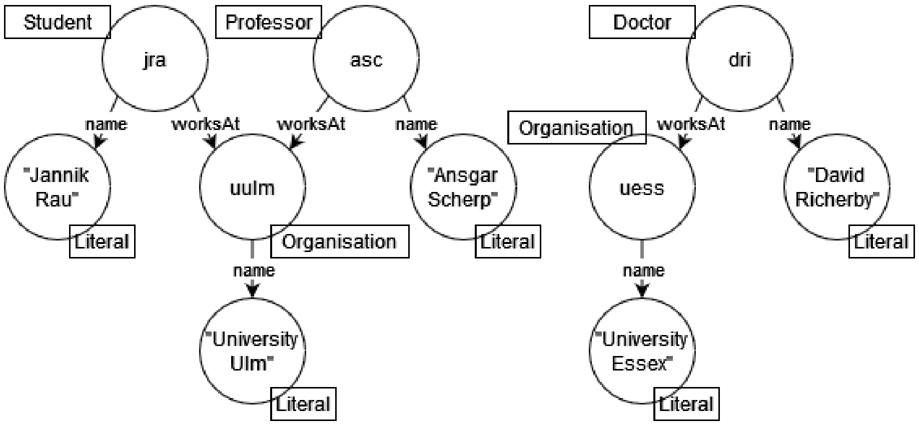


Figure 2: An example graph G displaying two universities and three employees.

graph $G = \{V, E, \Sigma_V, \Sigma_E\}$ with vertices

$$\begin{aligned} V = \{&\text{asc, dri, jra, uess, uulm},\\ &\text{"Ansgar Scherp", "David Richerby", "Jannik Rau"},\\ &\text{"University Essex", "University Ulm"}\},\end{aligned}$$

edges

$$\begin{aligned} E = \{&(\text{asc, "Ansgar Scherp"}), (\text{asc, uulm}),\\ &(\text{dri, "David Richerby"}), (\text{dri, uess}),\\ &(\text{jra, "Jannik Rau"}), (\text{jra, uulm}),\\ &(\text{uess, "University Essex}), (\text{uulm, "University Ulm})\}\end{aligned}$$

and respective vertex and edge label sets

$$\begin{aligned} \Sigma_V &= \{\text{Doctor, Literal, Organisation, Professor, Student}\},\\ \Sigma_E &= \{\text{name, worksFor}\}.\end{aligned}$$

The graph represents two universities and three employees. For example edges $(asc, uulm)$ and $(jra, uulm)$ labelled with *worksFor* together with edges $(asc, ”AnsgarScherp”)$, $(jra, ”JannikRau”)$ and $(uulm, ”UniversityUlm”)$ labelled with *name* and vertex labels *Professor* (asc), *Student* (jra) and *Organisation* ($uulm$) state that the professor *Ansgar Scherp* and the student *Jannik Rau* both work at the organisation *University Ulm*.

3.1.2 *Bisimulation.*

The algorithms of Schätzle et al. [32] and Kaushik et al. [19] compute a k-bisimulation partition of the input graph $G = (V, E, \Sigma_V, \Sigma_E)$. A k-bisimulation is an equivalence relation, that groups together vertices, which have an equivalent structural neighbourhood up to distance k. One can consider the outgoing edges to determine equivalence, which results in a *forward bisimulation*, or the incoming edges, which results in a *backward bisimulation.*

Definition 2 (forward k-bisimulation). *The forward k-bisimulation $\approx^k_{fw} \subseteq V \times V$ with $k \in \mathbb{N}$ is defined as follows:*

- $u \approx^0_{fw} v$ *for all* $u, v \in V$,
- $u \approx^{k+1}_{fw} v$ *iff for every successor* u' *of* u *there exists a successor* v' *of* v *such that* $u' \approx^k_{fw} v'$.

k	Partition
0	1: V
1	1: {asc, dri, jra, uess, uulm}, 2: literals
2	1: {asc, dri, jra}, 2: {uess, uulm}, 3: literals

Table 1: Forward 2-Bisimulation partition of the example graph according to Definition 2.

Table 1 shows the forward 2-bisimulation partitions of the example graph in Figure 2. Initially, every vertex is considered 0-bisimilar. The following 1-bisimulation partition is comprised of a block with vertices that have a successor and vertices who do not. Finally, the 2-bisimulation partition contains (1) a block with vertices that have *non-literal* and *literal* successors, (2) a block with vertices that only have *literal* successors and (3) a block with vertices that do not have any outgoing neighbour.

Definition 3 (backward k-bisimulation). *The backward k-bisimulation $\approx^k_{bw} \subseteq V \times V$ with $k \in \mathbb{N}$ is defined as follows:*

- $u \approx^0_{bw} v$ *for all* $u, v \in V$,
- $u \approx^{k+1}_{bw} v$ *iff for every predecessor* u' *of* u *there exists a predecessor* v' *of* v *such that* $u' \approx^k_{bw} v'$.

k	Partition
0	1: V
1	1: {asc, dri, jra}, 2: {uess, uulm, literals}
2	1: {asc, dri, jra}, 2: {uess, uulm, "Ansgar Scherp", "David Richerby", "Jannik Rau" }, 3: {"University Essex", "University Ulm"}

Table 2: Backward 2-Bisimulation partition of the example graph according to Definition 3.

Table 2 shows the backward 2-bisimulation partitions of the example graph in Figure 2. Again, initially every vertex is considered 0-bisimilar. This time, the following 1-bisimulation partition is comprised of a block with vertices that have a predecessor and vertices who do not. Finally, the 2-bisimulation partition contains (1) a block with vertices that do not have any incoming neighbour, (2) a block with vertices that have incoming neighbours, which on their own do not have any incoming neighbours and (3) a block with vertices that have incoming neighbours, which on their own have incoming neighbours.

3.1.3 Bisimulation variants. Schätzle et al. and Kaushik et al. use slightly adapted notions of forward and backward k-bisimulation. Schätzle et al. incorporate edge labels into their k-bisimulation definition.

Definition 4 (edge labelled forward k-bisimulation). *The edge labelled forward k-bisimulation $\approx^k_{fw} \subseteq V \times V$ with $k \in \mathbb{N}$ is defined as follows:*

- $u \approx^0_{fw} v$ *for all* $u, v \in V$,
- $u \approx^{k+1}_{fw} v$ *iff for every successor* u' *of* u *there exists a successor* v' *of* v *with* $l_E((u,u')) = l_E((v,v'))$ *and* $u' \approx^k_{fw} v'$.

k	Partition
0	1: V
1	1: {asc, dri, jra}, 2: {uess, uulm}, 3: literals
2	1: {asc, dri, jra}, 2: {uess, uulm}, 3: literals

Table 3: Edge labelled forward 2-Bisimulation partition of the example graph according to Definition 4.

Table 3 shows the edge labelled forward 2-bisimulation partitions of the example graph in Figure 2. They are nearly identical to the forward 2-bisimulation partitions depicted in Table 1. The only difference is the 1-bisimulation partition. According to Definition 4 vertices *uess* and *uulm* are not considered 1-bisimilar to vertices *asc*, *dri* and *jra*, because they do not have an outgoing edge with label *worksAt*.

One should note that originally, Schätzle et al. compute the full bisimulation in their approach, i.e. $k \rightarrow n_{full}$ with $n_{full} \in \mathbb{N}$ being the value for which the bisimilarity relation between any two vertices do not change anymore for a $n > n_{full}$. For the example graph G, the full bisimulation (w.r.t. Definition 4) is reached at $n_{full} = 1$.

The backward bisimulation provided by Kaushik et al. additionally takes the vertex labels into account.

Definition 5 (vertex labelled backward k-bisimulation). *The vertex labelled backward k-bisimulation $\approx^k_{bw} \subseteq V \times V$ with $k \in \mathbb{N}$ is defined as follows:*

- $u \approx^0 v \iff l_V(u) = l_V(v)$,
- $u \approx^{k+1} v \iff v \approx^k u$ *and for every predecessor* u' *of* u *there exists a predecessor* v' *of* v *such that* $u' \approx^k v'$ *and vice versa.*

Table 4 shows the vertex labelled backward 2-bisimulation partitions of the example graph in Figure 2. Initially, the vertices are partitioned by label, which makes up the 0-bisimulation partition. Afterwards, *uess, uulm* and the *literal* vertices are split according to the labels of their parents. In the final 2-bisimulation partition, no vertex is considered 2-bisimilar and hence the partition only contains singletons.

k	Partition
0	1: {asc}, 2: {dri}, 3: {jra}, 4: {uess, uulm}, 5: literals
1	1: {asc}, 2: {dri}, 3: {jra}, 4: {uess}, 5: {uulm}, 6: {"Ansgar Scherp"}, 7: {"David Richerby"}, 8: {"Jannik Rau"}, 9: {"University Essex", "University Ulm"}
2	1: {asc}, 2: {dri}, 3: {jra}, 4: {uess}, 5: {uulm}, 6: {"Ansgar Scherp"}, 7: {"David Richerby"}, 8: {"Jannik Rau"}, 9: {"University Essex"}, 10: {"University Ulm"}

Table 4: Vertex labelled backward 2-Bisimulation partition of the example graph according to Definition 5.

Henceforth, if not further specified, $\approx^k_{fw}$ and $\approx^k_{bw}$ will correspond to the edge labelled forward k-bisimulation and the vertex labelled backward k-bisimulation.

3.1.4 Graph Summary Model. The (B,R,S)-algorithm summarises a graph w.r.t. a *Graph Summary Model* (GSM) [6].

Definition 6 (Graph Summary Model). *A Graph Summary Model is an equivalence relation $\sim \subseteq V \times V$ on the vertices V of a graph G. Hence, the equivalence classes in $\sim$ define a partition of the graph G.*

A simple example of a GSM would be label equality, i. e. two vertices are considered equivalent iff they have the same label. Dependent on the application, one might want to summarise a graph w.r.t. different GSMs. Therefore, the algorithm works with GSMs defined in the formal language FLUID [6]. In FLUID, one can define its own custom GSM. The basic building blocks of a GSM are Simple Schema Elements (SSE) and Complex Schema Elements (CSE). Additionally, the user is able to incorporate six parameterisations. One parameterisation, the *Chaining parameterisation*, is of special interest, as it enables summarisation similarly to the concept of k-bisimulation. The other five parameterisations are out of scope and the reader is referred to the original paper [6].

Definition 7 (Simple Schema Element). *A Simple Schema Element groups vertices together by either comparing*

(1) their neighbours – Object Cluster (OC),
(2) their outgoing edge labels – Predicate Cluster (PC),
(3) or the combination of the former two, namely their neighbours and the edge labels which connect them – Predicate-Object Cluster (POC).

This means two vertices are considered equal under

OC *iff they have the same outgoing neighbours,*
PC *iff they have the same outgoing edge labels,*
POC *iff they have the same outgoing neighbours connected over the same edge labels.*

One observation for SSEs is that they only take local information of a vertex into account, e. g., the neighbours, the edge labels or the combination of the two. To be able to summarise vertices based on information of their neighbours local information, e. g., the neighbours of their neighbours, the edge labels of their neighbours or the combination of the two, FLUID provides a Complex Schema Element, which enables this feature [6].

Definition 8 (Complex Schema Element). *A Complex Schema Element consists of three equivalence relations and is defined as $CSE := (\sim^s, \sim^p, \sim^o)$. Two vertices v, v' are considered equal, iff*

$$v \sim^s v', \tag{1}$$

$$\forall (v,w) \in E: \ \exists (v',w') \in E \text{ with } l_E(v,w) \sim^p l_E(v',w') \text{ and vice versa,} \tag{2}$$

$$\forall w \in \Gamma^+(v): \ \exists w' \in \Gamma^+(v') \text{ with } w \sim^o w' \text{ and vice versa.} \tag{3}$$

Here, $N^+(v)$ denotes the set of outgoing neighbours of v.

Introducing the identity relation $id = \{(v,v) | v \in V\}$ and tautology relation $T = V \times V$, one is able to represent the three SSEs as CSEs.

$$OC = (T, T, id),$$
$$PC = (T, id, T),$$
$$POC = (T, id, id).$$

An example of a CSE, which not only takes local information into account, is given by $CSE-1 = (T, id, PC) = (T, id, (T, id, T))$. It considers vertices as equal, iff they have the same outgoing edge labels plus neighbours with the same outgoing edge labels. As mentioned before, the chaining parameterisation is of special interest, as it enables computing k-bisimulations of a graph G and hence increases the considered neighbourhood for determining vertex equality [6].

Definition 9 (Chaining parameterisation). *The chaining parameterisation $cp(CSE, k)$ takes a complex schema element $CSE := (\sim^s, \sim^p, \sim^o)$ and a chaining parameter $k \in \mathbb{N}_{>0}$ and returns an equivalence relation CSE^k that corresponds to recursively applying CSE to a distance of k hops. CSE^k is defined inductively as follows:*

$$CSE^1 = (\sim^s, \sim^p, \sim^o),$$
$$CSE^{k+1} = (\sim^s, \sim^p, CSE^k).$$

Using the chaining parameterisation, the edge labelled forward k-bisimulation (Definition 4) can be defined as

$$CSE_{\text{Schätzle}} = cp((T, id, T), k). \tag{4}$$

$CSE_{\text{Schätzle}}$ can be written short as $(T, id, T)^k$.

The vertex labelled backward k-bisimulation (Definition 5) can be defined as

$$CSE_{\text{Kaushik}} = dp(cp((OC_{type}, T, OC_{type}), k), i). \tag{5}$$

CSE_{Kaushik} can be written short as $i-(OC_{type}, T, OC_{type})^k$. Here, $i - SE$ modifies the schema element SE to only use incoming predicates. This results in v and v' considered equal, iff

$$v \sim^o v' \tag{6}$$

$$\forall (w,v) \in E: \ \exists (w',v') \in E \text{ with } l_E(w,v) \sim^p l_E(w',v') \text{ and vice versa} \tag{7}$$

$$\forall w \in \Gamma^-(v): \ \exists w' \in \Gamma^-(v') \text{ with } w \sim^s w' \text{ and vice versa,} \tag{8}$$

where $N^-(v)$ denotes the set of incoming neighbours of v. Additionally, OC_{type} considers two vertices to be equal if they have the same label(s).[5] For clarification purpose, this means two vertices are considered equal under $i-(OC_{type}, T, OC_{type})^k$, iff they have the same label(s) plus have respective incoming neighbours with same label(s), which have respective incoming neighbours with same label(s), and so forth up to a distance of k. [6]

3.2 Methods

In Table 5 the considered methods are compared concerning their *Paradigm*, *Origin*, *Worst-Case Complexity* and the respective computed *Graph Summary Model*. A fundamental difference between (B,R,S) and the two algorithms lies in the approach to construct the graph summary. At the beginning of the execution, (B,R,S) considers every vertex to be in its own block, hence not equal to any other vertex. At the end of execution, vertices with the same vertex summary are merged together. Therefore, (B,R,S) can be seen as a *Bottom-Up* approach. In contrast, Schätzle et al. consider every vertex to be in the same block at the beginning, hence equal to every other vertex. And Kaushik et al. consider vertices with the same label to be in the same block at the beginning, hence equal to some other vertex. During execution, the partition is successively refined. Therefore, the two algorithms specifically designed for bisimulation can be seen as *Top-Down* approaches. The following subsections describe the algorithms' origins and procedure in more detail.

3.2.1 *(B,R,S).*

The (B,R,S)-algorithm is not a specific implementation for k-bisimulation reduction. Rather, one can define a custom graph summary model $\sim$ in FLUID (Section 3.1.4) and the algorithm summarises a graph w.r.t. $\sim$. However, the algorithm is capable of computing k-bisimulation partitions through the *Chaining parameterisation* $cp(CSE, k)$ (Definition 9), which extends the considered neighbourhood for constructing vertex summaries as defined by $\sim$ up to $k > 1$ hops [6]. In other words, this enables one to incorporate k-bisimulation into any graph summary model defined in FLUID. The algorithm is executed in parallel and makes use of the Signal/Collect paradigm [5]. In Signal/Collect, vertices collect information from their neighbours, sent over the connected edges as signals. Using this paradigm, the algorithm is implemented as a set union approach, starting with

[5]This might seem confusing with Definition 7 as it states that *OC* compares neighbours. However, FLUID is defined on RDF triples, in which vertex label(s) are expressed through (v, rdf:type, label), where label is considered a vertex as well, and hence a neighbour of v.

[6]The original paper [6] defines GSMs for Schätzle et al. and Kaushik et al. as well, however they differ from the definitions provided here. Reasons for this might be, that different GSMs result in the same Graph Summary or that the authors interpreted the data structures and bisimulation definitions in a different way.

Algorithm	Paradigm	Based on	Worst-Case Complexity	Graph Summary Model
Set Union Approaches (Bottom-Up)				
(B,R,S) [5]	Parallel – Signal/Collect	Set Union Problem [35]	$O(n + m \cdot \alpha(m + n, n))$	-
Set Refinement Approaches (Top-Down)				
Schätzle et al. [32]	Distributed – MapReduce	Blom and Orzan [4], a distributed version of Kanellakis and Smolka [18]	$O(m \cdot n + n^2)$	Definition 4, Equation (4)
Kaushik et al. [19]	Sequential	Paige and Tarjan [30]	$O(k \cdot m)$	Definition 5, Equation (5)

Table 5: Comparison of the considered algorithms.

every vertex in its own singleton set. Before outlining the algorithm's procedure, three important notions, based on Definition 6, Definition 7 and Definition 8, are introduced.

Definition 10 (Vertex Schema). *The (Simple) Vertex Schema of a vertex $v \in V$ is the local information of v, which is relevant for the equivalence relations $\sim^s$ and $\sim^o$ as defined by the GSM $\sim := (\sim^s, \sim^p, \sim^o)$.*

Definition 11 (Edge Schema). *The (Simple) Edge Schema of an edge $(v, w) \in E$ is the information of (v, w), which is relevant for the equivalence relation $\sim^p$ as defined by the GSM $\sim := (\sim^s, \sim^p, \sim^o)$.*

Definition 12 (Vertex Summary). *The Vertex Summary vs of a vertex $v \in V$ is the relevant information for the GSM $\sim := (\sim^s, \sim^p, \sim^o)$.*

Hence, the vertex summary vs of a vertex v can be constructed through its vertex schema w.r.t. $\sim^s$, its edge schema w.r.t. $\sim^p$ and the vertex schemata of its neighbours w.r.t. $\sim^o$.

The (B,R,S)-algorithm computes the SG of a Graph G w.r.t. a GSM $\sim$ as outlined in Algorithm 1. In parallel, for all vertices $v \in V$, the algorithm computes their vertex schema w.r.t. $\sim^s$ (Line 4) and $\sim^o$ (Line 5). The latter is then signaled to the neighbours w of v (Line 6). Afterwards, for every neighbour w, the edge schema w.r.t. $\sim^p$ is constructed (Line 9), which is then used together with the two vertex schemata to construct the vertex summary vs for v w.r.t. $\sim$ (Line 10). If the GSM incorporates chaining parameterisation, then vs is signaled to neighbours w (Line 14), which update their vs accordingly. This procedure is repeated k times. At the end of execution, vertices with the same vertex summary are merged together to vertices VS_{vs}, which make up the resulting summary graph SG (Line 15).

Algorithm 1: Parallel (B,R,S)-Algorithm.

```
1  function ParallelSummarise($G, SG, (\sim_s, \sim_p, \sim_o)_k$)
2      returns graph summary $SG$
3      forall $v \in V$ do in parallel
4          $vs \leftarrow$ ExtractSimpleVertexSchema($v, E, \sim_s$);
5          $tmp \leftarrow$ ExtractSimpleVertexSchema($v, E, \sim_o$);
           /* signal information relevant for $\sim_o$ to incoming
              neighbours */
6          forall $w \in V : (w, v) \in E$ do
7              $w$.Inbox $\leftarrow (tmp, 1)$;
           /* collect information of neighbours and construct
              complex vertex summaries */
8          forall $(tmp\_vs', r) \in v$.Inbox do
9              $t \leftarrow$ ExtractSimpleEdgeSchema($(v, w), \sim_p$);
10             $vs$.neighbour$_w \leftarrow (t, tmp\_vs')$
11             ; /* $k$-chaining repeats signal and collect
                  $k$-times */
12             if $r < k$ then
13                 forall $w \in V : (w, v) \in E$ do
14                     $w$.Inbox $\leftarrow (vs, r + 1)$;
15         $SG \leftarrow$ FindAndMerge($SG, vs, v$);
16     return $SG$;
```

3.2.2 *Schätzle et al. Algorithm.*

The approach taken by Schätzle et al. is distributed and makes use of the *MapReduce* paradigm. It is based on an implementation of Blom and Orzan [4] for reducing labelled transition systems (LTS), a form of a labelled directed graph (LDG), modulo strong bisimulation. For the sake of completeness, one should mention that Blom and Orzan's implementation is a distributed version of the *naive method* established by Kanellakis and Smolka [18]. They examine the problem of testing *observational equivalence* on *Calculus of Communicating Systems* (CCS) expressions, which can be represented as labelled directed graphs [18]. Their definition of *k-limited observational equivalence* corresponds to a stratified k-bisimulation. Putting focus back on the work of Schätzle et al., one difference in their approach to Blom and Orzan's is the data structure on which the algorithm is applied. Schätzle et al. consider RDF Graphs for their algorithm. Moreover, another difference between Blom and Orzan's implementation and the one of Schätzle et al. is the distributed framework they make use of. Blom and Orzan implement their own distributed framework using the *Message Passing Interface*[7], whereas Schätzle et al. make use of the Apache Hadoop Framework [8], an open-source implementation of *MapReduce* [13]. The reason why Blom and Orzan use their own distributed framework may be that there were no open-source frameworks at the time they were developing. Hence, Schätzle et al. essentially adapt Blom and Orzan's method to RDF Graphs and the Apache Hadoop Framework in their approach.

The rest of this section will refer to the approach of Schätzle et al., however the concepts are nearly the same for Blom and Orzan's work. Furthermore, definitions and pseudo-code are adapted to Multi-Relational labelled Property Graphs (Definition 1) and therefore they slightly differ from the original ones.

Two fundamental concepts are the *signature* and the *ID* of a vertex v with respect to the current iterations partition P_i.

Definition 13 (signature). *The signature of a vertex v with respect to the current iterations partition $P_i = \{B_{i1}, B_{i2}, \ldots\}$ is*

[7]https://www.mpi-forum.org/index.html
[8]https://hadoop.apache.org/

given by

$$sig_{P_i}(v) = \{(p, B_{ij}) | (v, w) \in E \text{ with } l_E((v, w)) = p \text{ and } w \in B_{ij} \in P_i\},$$

where B_{ij} corresponds to a block (set of vertices v) of P_i.

Hence, the signature of a vertex v with respect to the current iteration's partition P_i, denotes the set of outgoing edges to blocks of P_i [32]. With the corresponding bisimulation definition (Definition 4), this means that two vertices u, v are considered k-bisimilar, $u \approx^k_{fw} v$, iff $sig_{P_k}(u) = sig_{P_k}(v)$. Therefore the signature can be used to identify the block of a vertex and consequently represent the current bisimulation partition [32]. Now the signature of a vertex v w.r.t. P_i can be represented as

$$sig_{P_i}(v) = \{(p, sig_{P_i}(w)) | (v, w) \in E \text{ with } l_E((v, w)) = p\}. \quad (9)$$

At this point, the ID function has to be introduced. Considering the potential size of a vertex signature, it is more applicable to use an identifier function for the mapping of a vertex to its block [32].

Definition 14 (ID). *The ID of a vertex v with respect to the current partition P_i is given by the function*

$$ID_{P_i} : V \longrightarrow \mathbb{N},$$
$$ID_{P_i}(v) = hash(sig_{P_i}(v)),$$

which computes a hash value for $sig_{P_i}(v)$.

Now the signature of a vertex v w.r.t. the current partition P_i can be represented as

$$sig_{P_i}(v) = \{(p, ID_{P_i}(w)) | (v, w) \in E \text{ with } l_E((v, w)) = p\} \quad (10)$$

With $sig_{P_i}(v)$ and $ID_{P_i}(v)$, the procedure for computing a k-bisimulation partition (using Definition 4) is sketched in Algorithm 2. The initial partition is trivial, as every vertex is considered to be bisimilar to every other vertex. Hence, the initial partition is equal to V (Line 2). Afterwards, the algorithm performs k iterations (Line 3 to Line 10), where in each iteration $i \in \{1, 2, \ldots, k\}$, first the number of distinct vertex ID values w.r.t. iteration $i - 1$ is computed (Line 4). Afterwards, the necessary information for constructing a vertex's signature $sig_{P_i}(v)$ is send to every vertex v (Line 5). This information consists of the edge label $l_E((v, w)) = p \in \Sigma_E$ and the block identifier $ID_{P_{i-1}}(w)$ for every outgoing edge (v, w) from v. Subsequently, the signature $sig_{P_i}(v)$ is constructed according to the received information and the identifiers $ID_{P_i}(v)$ are updated for all v (Line 6). Finally, at the end of an iteration, the algorithm checks if any vertex ID was updated by comparing the number of distinct values in ID_i with the ones in ID_{i-1} (Line 7). If no vertex ID was updated, we have reached full bisimulation [4, 32] and hence can stop execution early (Line 8). At the end, the resulting k-bisimulation partition P_k is constructed through putting vertices v in one block, if they share the same identifier value $ID_{P_k}(v)$.

3.2.3 *Kaushik et al.*

In contrast to the other two approaches, Kaushik et al. provide a sequential implementation of k-bisimulation reduction. It is an adapted version of Paige and Tarjans *naive method* [30] for solving the *relational coarsest partition problem.* Table 11 in Appendix A clarifies the notations used by Kaushik et al. and Paige and Tarjan, such that they are uniform for both, which is not the case in the original papers. Furthermore, the definitions, which are necessary to follow the two approaches, are provided as well. All definitions are taken from Paige and Tarjan [30]. Moreover, in the following definitions

$$N^-(S) = \{v | \exists w \in S \text{ with } vEw \text{ and } v \in V\},$$

corresponds to the set of elements $v \in V$, which are related to any element $w \in S$ for a relation $E \subseteq V \times V$. Likewise

$$N^+(S) = \{w | \exists v \in S \text{ with } vEw \text{ and } y \in V\},$$

corresponds to the set of elements $w \in S$, which are related to any element $v \in V$ for a relation $E \subseteq V \times V$. Speaking in terms of vertices and edges, $N^-(S)$ corresponds to the predecessors (incoming neighbours) of the vertices in S and $N^+(S)$ corresponds to the successors (outgoing neighbours) of the vertices in S.

Algorithm 2: Native Schätzle Algorithm

```
1  function ComputeBisim($G = (V, E, \Sigma_V, \Sigma_E), k \in \mathbb{N}$)
       /* Initially, all vertices $v$ are in the same block with
          $ID_{P_0}(v)$ = 0 */
2      $P_i \leftarrow V$;
3      for $i = 1$ to $k$ do
4          $count_i \leftarrow |ID_{i-1}|$;
           /* Map Job */
5          Send $(l_E((v, w)), ID_{P_{i-1}}(w))$ to $v$ $\forall (v, w)$;
           /* Reduce Job */
6          Construct $sig_{P_i}(v)$ and update $ID_{P_i}(v)$;
           /* Check if full bisimulation */
7          $count_i \leftarrow |ID_i| - count_i$;
8          if $count_i == 0$ then
9              $ID_{P_k}(v) \leftarrow ID_{P_i}(v)$ $\forall v \in V$;
10             break;
11     Adjust $P_k$ via $ID_{P_k}(v)$ for all $v$;
12     return $P_k$;
```

Definition 15 (subset stable with respect to another subset). *A subset $B \subseteq V$ is stable with respect to another subset $S \subseteq V$ if either*

(1) $B \subseteq N^-(S)$ or
(2) $B \cap N^-(S) = \emptyset$.

Speaking in terms of vertices and edges, this means that either

(1) All elements in B are predecessors of elements in S or
(2) No element in B is a predecessor of an element in S.

Definition 16 (stable partition with respect to a subset). *A partition P_i of V is stable with respect to a subset $S \subseteq V$ if every block $B_{ij} \in P_i$ is stable with respect to S.*

Definition 17 (stable partition). *A partition P_i of V is stable, if it is stable with respect to each of its own blocks B_{ij}.*

Definition 18 (partition refinement). *For two partitions $P_i = \{B_{i1}, B_{i2}, \ldots\}$ and $P_j = \{B_{j1}, B_{j2}, \ldots\}$ of a finite set V. P_i is a refinement of P_j, if every block B_{ik} of P_i is contained in a block B_{jl} of P_j.*

Definition 19 (relational coarsest partition problem). *The relational coarsest partition problem is that of finding, for a given relation E and initial partition P_0 over a set V, the coarsest stable refinement of P_0, i.e., the partition such that every other stable partition is a refinement of it, so that it has the fewest blocks.*

The naive method of Paige and Tarjan for solving the relational coarsest partition problem is defined in Algorithm 3. It makes use of the *split* function, which is defined as follows:

Definition 20 (*split*). *For any partition P_i and subset $S \subseteq V$, let $split(S, P_i)$ be the refinement of P_i obtained by replacing each* *block $B_{ij} \in P_i$, for which $B_{ij} \cap N^-(S) \neq \emptyset$ and $B_{ij} - N^-(S) \neq \emptyset$ by the blocks $B'_{ij} = B_{ij} \cap N^-(S)$ and $B''_{ij} = B_{ij} - N^-(S)$.*

The idea of the algorithm is to gradually refine the partition by using its own blocks or unions of them as splitters, until the partition is stable and hence no such splitters exist anymore [30].

The resulting stable partition is equivalent to the full forward bisimulation of the initial partition P_0 [18].[9]

As mentioned before, Algorithm 4 is an adaption of Algorithm 3. The first difference is that in each iteration $i \in \{1, 2, \ldots, k\}$ the partition is stabilised with respect to each of its own blocks (line 7 to line 14). This ensures that after iteration i, the algorithm has computed the i-bisimulation [19], which is not the case in Algorithm 3. Second, the $N^+(\cdot)$ function is used to stabilise the partition (line 9 to line 13), instead of $N^-(\cdot)$ (see Definition 20). As a result, Algorithm 4 computes the k-backward bisimulation.[10] To check if P_i is the relational coarsest partition of P_0 (i.e., if full bisimulation is reached), the algorithm makes use of the boolean *wasSplit* (lines 6 and 13). If the value is equal to $false$ at the end of an iteration i, the algorithm stops execution early (line 15). Moreover, Algorithm 4 incorporates a *usedSplitters* Map (line 3), which stores blocks that where used in a split. The algorithm provided by Kaushik et al. does not include such a Map. Its purpose is to remove unnecessary stability checks. As stated by Paige and Tarjan [30], for a partition P that is stable w.r.t. a block B it holds true that each refinement R of P is stable w.r.t. B as well. So after the partition P_i was stabilised w.r.t. a block copy B^{copy} (line 7 to line 14), we can add B^{copy} to the *usedSplitters* map and stop caring about it in subsequent iterations.

4 EXPERIMENTAL APPARATUS

4.1 Datasets

The use of real-world datasets when evaluating graph summarisation algorithms is of great importance, as synthetic datasets lack in capturing the characteristics of real-world graphs [5, 25]. Therefore, four real-world datasets and two synthetic datasets were chosen for the experiments.

We experiment with smaller and larger sized graphs. Table 6 and Table 7 list statistics of the smaller and larger datasets respectively, with $|l_v|$ corresponding to the number of different label sets, $|l_v(\cdot)|$ corresponding to the number of labels of a vertex $v \in V$, $d(G)$ corresponding to the average degree of vertices in G, $d_G(v)$ corresponding to the degree of a vertex $v \in V$ and $\Delta(G)$ corresponding to the maximum degree in G.

4.1.1 Real-World Datasets. Concerning the real-world datasets, four datasets were chosen.

First, the *Laundromat100M* and *Laundromat15B* datasets contain 100 million and 15 billion quads of the *LOD Laundromat* service.[11] LOD Laundromat automatically cleans existing Linked Datasets and provides the cleaned version on a publicly accessible Website [1]. The smaller dataset consists of about 29.87 million vertices and 87.85 million edges. Compared to the other two smaller datasets (BTC150M and BSBM100M), Laundromat100M contains a huge amount of different labels (= 33 431) and label sets (= 7 373). However, on average a vertex's label set $l_v(v)$ only contains 0.93 labels. Concerning vertex degrees, Laundromat100M has the smallest average and maximum degree values concerning all (incoming and outgoing) and incoming edges. Laundromat15B is the largest dataset considered here. The graph contains about 1.46 billion vertices and 13.52 billion edges. Further, about 0.71 million different label sets and 1.02 million edge labels can be observed in the dataset.

Second, the *BTC150M* and *BTC2B* datasets contain 150 million quads and nearly[12] all quads of the *Billion Triple Challenge 2019* (BTC2019) [16] dataset, which is comprised of approximately 2.15 billion unique quads. The authors conducted a statistical analysis of the entire BTC2019 dataset. They found out, that 93% (~2 billion) of the total quads originate from *Wikidata*.[13] BTC150M is a chunk of those 2 billion quads. Moreover, investigation of used vocabularies, predicates and classes, lead the authors to the conclusion that BTC2019 is a 'highly diverse dataset' [16]. The BTC150M dataset consists of about 4.98 million vertices and 145.48 million edges. Compared to the other two smaller datasets, it contains the least amount of different labels (= 69) and label sets (= 137). However, on average a vertex has 12.38 different labels, which is about 12 times more than the other two. Furthermore, BTC150M contains the most amount of different edge labels (= 10 750) and has the highest average and maximum degree values concerning all and outgoing edges. For incoming edges it only has the highest maximum degree value. BTC2B consists of about 79.65 million vertices and 1.92 billion edges. It contains 113 365 different label sets and 38 136 different edge labels. On average, a vertex has 10.40 different labels, which is the largest value among the larger datasets.

4.1.2 Synthetic Dataset. Concerning the synthetic datasets, two datasets were chosen.

BSBM100M and *BSBM1B* are two versions of the *Berlin SPARQL Benchmark* (BSBM) [3]. Originally developed for 'comparing the SPAQRL query performance of native RDF stores with that of SPARQL-to-SQL rewriters'[3], the benchmark consists of a data generator and a test driver. The benchmarks test driver is further omitted here, because it functions as query executor, which is out of scope. The benchmarks data generator is capable of producing an RDF dataset, which consists of products, vendors, offers and reviews. It simulates an e-commerce use case comprised of the aforementioned classes. Moreover, datasets can be scaled to arbitrary sizes with a scale factor n, which represents the number of products present in the resulting dataset. BSBM100M and BSBM1B consist of 100 million and 1 billion triples and were generated by a scale factor of $n_{100M} = 284\,826$ and $n_{1B} = 2\,850\,000$ respectively. More concrete, BSBM100M consists of about 17.77 million vertices and 89.54 million edges. Compared to the other smaller datasets, it contains the least amount of edge labels (= 39). Furthermore, the graph has the largest maximum incoming degree and smallest maximum outgoing degree. BSBM1B consists of about 172.25 million vertices and 941.21 million edges. 6 153 different label sets and 39 different edge labels can be observed in this dataset.

Other notable synthetic benchmark datasets are LUBM[14] and SP^{2}Bench[15], however they will not be used for experiments.

[9]Kanellakis and Smolka refer to this as *observationally equivalent or strongly equivalent.*

[10]As a note, one could achieve the same result with $N^-(\cdot)$ by reversing all the edges in E

[11]https://lodlaundromat.org

[12]The dataset was provided in several individual files. One file containing about 150M quads was corrupted and could not be parsed, hence it was omitted from experiments

[13]https://www.wikidata.org/

[14]http://swat.cse.lehigh.edu/projects/lubm/

[15]http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B

Algorithm 3: Naive Algorithm for relational coarsest partition of initial partition P_0 of V

```
function NaiveApproach(Initial Partition $P_0$ of $V$)
    $P_i \leftarrow P_0$;
    while $\exists$ splitter $S$, which is a block or a union of some blocks $B_{ij}$ of $P_i$ do
        $P_i \leftarrow split(S, P_i)$;
    return $P_i$;
```

Algorithm 4: Native Kaushik Algorithm

```
function ComputeBisim($G = (V, E, \Sigma_V, \Sigma_E)$, $k \in \mathbb{N}$)
    /* $P_i := \{B_{i1}, B_{i2}, \ldots, B_{in}\}$ */
    $P_i \leftarrow$ partition $V$ by label;
    $usedSplitters \leftarrow \{\}$;
    for $i = 1$ to $k$ do
        $P_i^{copy} \leftarrow$ copy of $P_i$;
        $wasSplit \leftarrow false$;
        for $B_{ij}^{copy} \in P_i^{copy}$ with $B_{ij}^{copy} \notin usedSplitter$ do
            /* Use blocks of copy partition to stabilize blocks of original partition */
            for $B_{ij} \in P_i$ do
                /* Only consider a block in case of instability */
                if $B_{ij} \cap N^+(B_{ij}^{copy}) \neq \emptyset$ and $B_{ij} - N^+(B_{ij}^{copy}) \neq \emptyset$ then
                    /* Replace the block in case of instability */
                    $P_i.add(B_{ij} \cap N^+(B_{ij}^{copy}))$;
                    $P_i.add(B_{ij} - N^+(B_{ij}^{copy}))$;
                    $P_i.delete(B_{ij})$;
                    $wasSplit \leftarrow true$;
            $usedSplitter.add(B_{ij}^{copy})$;
        if $wasSplit == false$ then
            break;
    return $P_i$;
```

(a) General Statistics

Graph	|V|	|E|	$\|\Sigma_V\|$	$\|l_v\|$	$\mu(\|l_v(\cdot)\|)$	$\sigma(\|l_v(\cdot)\|)$	$\|\Sigma_E\|$
Laundromat100M	29 873 739	87 848 315	33 431	7 373	0.93	43.67	5 630
BTC150M	4 979 259	145 476 874	69	137	12.38	183.98	10 750
BSBM100M	17 769 291	89 542 510	1 289	2 274	1.02	0.13	39

(b) Degree Statistics

Graph	$d(G)$	$\sigma\ d_G(v)$	$\Delta(G)$	$d^-(G)$	$\sigma\ d_G^-(v)$	$\Delta^-(G)$	$d^+(G)$	$\sigma\ d_G^+(G)$	$\Delta^+(G)$
Laundromat100M	5.89	569.31	1 570 748	3.37	597.34	1 570 748	10.08	198.81	545 688
BTC150M	58.43	5655.05	5 629 275	30.86	518.16	283 686	151.11	12 800.63	5 628 254
BSBM100M	10.08	1143.83	2273014	5.99	1247.83	2273014	9.92	4.35	76

Table 6: Statistics of the smaller datasets.

4.2 Procedure

An experiment consists of the algorithm to run, the dataset to summarise and the bisimulation degree k. In case of the (B,R,S) algorithm, it additionally consists of the graph summary model to use. Thus, we have two different graph summary models, each with a native implementation and an implementation through the (B,R,S) algorithm. This makes four algorithms, which are applied on each of the six datasets with a bisimulation degree of $k = 10$, making up a total number of 24 experiments. Each experiment is executed according to the following procedure.

We run the algorithm six times in a row with the specific configuration. We use the first run (run 0) as a warm up run and hence do not account it for measurement of the dependent variable. Afterwards, the following five runs (runs 1–5) are used to measure the variable. In such a way, the last five runs of all experiments follow the same environment, i. e., none of the experimental results are influenced by any processes that ran before the experiment started.

4.3 Implementation

All algorithms are implemented in Scala upon the Apache Spark Framework. Apache Spark offers an API for parallel computation

(a) General Statistics

Graph	|V|	|E|	$\|\Sigma_V\|$	$\|l_v\|$	$\mu(\|l_v(\cdot)\|)$	$\sigma(\|l_v(\cdot)\|)$	$\|\Sigma_E\|$
Laundromat15B	1 462 577 896	13 515 336 321	714 309	4 078 918	2.13	428.51	1 025 510
BTC2B	79 653 641	1 918 545 492	113 365	576 265	10.40	559.42	38 136
BSBM1B	172 253 440	941 206 409	6 153	27 306	1.03	0.18	39

(b) Degree Statistics

Graph	$d(G)$	$\sigma\ d_G(v)$	$\Delta(G)$	$d^-(G)$	$\sigma\ d^-_G(v)$	$\Delta^-(G)$	$d^+(G)$	$\sigma\ d^+_G(G)$	$\Delta^+(G)$
Laundromat15B	18.48	9 791.92	215 181 452	10.61	10 460.35	215 153 114	18.04	779.03	19 188 279
BTC2B	48.40	17 119.01	65 879 409	31.57	1 777.21	3 856 778	55.15	25 719.37	65 878 298
BSBM1B	10.93	3 862.48	23 924 441	6.57	4 234.02	23 924 441	10.04	4.45	85

Table 7: Statistics of the larger datasets.

which enables implementation of Map-Reduce and Signal-Collect routines. For the sequential Kaushik algorithm (Algorithm 4) this API was used for graph and partition data structures, as well as in parsing and initialising these. However, it must be noted that the stabilisation routine of the algorithm (line 4 – line 16) is implemented sequentially. The stabilisation routine is limited concerning opportunities to parallelise it. The first inner loop (line 7) has the following problem. In each iteration of the second inner loop at line 8, there is a chance that new blocks are added to the current partition P_i. This leads to two new blocks which must be checked for instability with respect to all the other block-copies in the copy partition. Hence, as a solution one could do both loops in parallel and after the first inner loop is finished for all the block-copies B^{copy} in the copy partition, the newly created blocks are exchanged between all of them. This is repeated until no blocks are created anymore. Moreover, to make this work, a worker which is for example assigned to stabilise block B_{i1} with respect to block-copy B^{copy}_{i5} needs to know, or need to be able to compute the neighbours of the vertices in B^{copy}_{i5}. This either results in more communication (e.g., send the neighbours to the worker who is assigned for the respective block-copy) or in more memory consumption (e.g. broadcast a read-only map containing (vertex $\longrightarrow$ list of neighbours)-entries of all vertices to every worker).

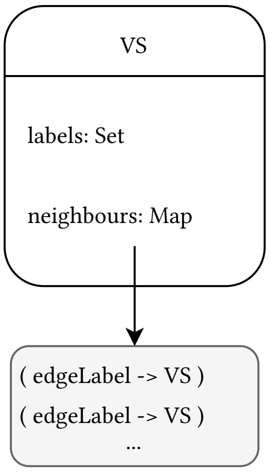


Figure 3: Vertex summary of GSM Schätzle.

Furthermore, experiments on the smaller datasets revealed that the (B,R,S) algorithm had extreme memory consumption issues.[16] The cause of these issues was the data structure of a vertex summary and how the algorithm operated on it. Figure 3 depicts an example vertex summary of a vertex v with respect to the GSM of Schätzle (eq. (4)). It first contains a label set, which is further omitted, as the GSM of Schätzle does not incorporate any vertex labels. Second, it contains a map of neighbours, which stores $(key, value)$ pairs, where key is the edge label connecting v to its neighbour w, and $value$ is the vertex summary of the neighbour w. The memory issues arise from the neighbours map, which potentially can become heavily nested. For each neighbour of v, the map stores the vertex summary of a neighbour w. Furthermore, the stored vertex summary of w contains a respective map of neighbours of w, which potentially contains additional vertex summaries of the neighbours neighbours of w, and so on. Storing and signalling such nested structures consumes a lot of memory even for rather small graphs. Furthermore, the Apache Spark tuning guide recommends avoiding nested structures and collection classes like maps.[17] Hence, the algorithm had to be adjusted to overcome this issue. The source code for the adjusted algorithm is provided in a git repository.[18] It is now designed similarly to the Schätzle algorithm, making use of identifiers derived from hashes. More concrete, every vertex contains an identifier $id_{\sim_s}$ and $id_{\sim_o}$. At the end of execution, two vertices v, v' are considered equal w.r.t. $(\sim_s, \sim_p, \sim_o)_k$, iff $v.id_{\sim_s} = v'.id_{\sim_s}$. Algorithm 5 shows the pseudo code of the adapted version. The algorithm now only operates on vertex summaries in the initialisation step (line 3 to line 7), which computes the local information of a vertex w.r.t. $\sim_s$ (line 4) and $\sim_o$ (line 5) of the respective GSM. After computing the local information, an ID function is used to derive a numerical value from the initial vertex summaries, resulting in identifier values $id_{\sim_s}$ (line 6) and $id_{\sim_o}$ (line 7). The ID function calculates a numerical value based on the vertex summaries content, such that if two vertex summaries vs_1 and vs_2 have the same content, the ID function outputs the same values. This can be seen as iteration $i = 0$. Afterwards, if $k = 1$ (line 26 to line 30), every vertex v sends its $id_{\sim_o}$ value along with the simple edge schema t_w to all its neighbours w (line 28). Subsequently, each vertex merges the incoming messages into an array containing tuples with the received information $(t_w, id_{\sim_o})$, eliminating any duplicates (line 29). Finally, the vertex's $id_{\sim_s}$ value

[16]The old implementation can be found in the *bisimulation* branch https://gitlab.informatik.uni-ulm.de/dbis/data-science-and-big-data-analytics/teaching/2021ss-thesis-jannik/-/tree/bisimulation

[17]https://spark.apache.org/docs/latest/tuning.html

[18]https://gitlab.informatik.uni-ulm.de/dbis/data-science-and-big-data-analytics/teaching/2021ss-thesis-jannik

Algorithm 5: Adjusted Parallel (B,R,S)-Algorithm

```
1  function ParallelSummariseAdjusted($G, SG, (\sim_s, \sim_p, \sim_o)_k$)
2      returns graph summary SG
       /* Initialisation */
3      forall $v \in V$ do in parallel
4          $vs_{\sim_s} \leftarrow$ VertexSchema($v, E, \sim_s$);
5          $vs_{\sim_o} \leftarrow$ VertexSchema($v, E, \sim_o$);
6          $v.id_{\sim_s} \leftarrow vs_{\sim_s}.ID$;
7          $v.id_{\sim_o} \leftarrow vs_{\sim_o}.ID$;
8      if $k > 1$ then
           /* Signal initial messages. Update $v.id_{\sim_s}$ and $v.id_{\sim_o}$ */
9          forall $v \in V$ do in parallel
10             SignalMessages($t_w, v.id_{\sim_s}, v.id_{\sim_o}$);
11             $arr_{id_{\sim_s}} \leftarrow$ MergeMessages($(t_w, id_{\sim_s})$);
12             $arr_{id_{\sim_o}} \leftarrow$ MergeMessages($(t_w, id_{\sim_o})$);
13             $v.id_{\sim_s} \leftarrow$ Hash($arr_{id_{\sim_s}}$);
14             $v.id_{\sim_o} \leftarrow$ Hash($arr_{id_{\sim_o}}$);
           /* Signal $k-2$ times. Do not include $t_w$ when updating $v.id_{\sim_o}$. */
15         for $i = 2$ to $k - 1$ do
16             forall $v \in V$ do in parallel
17                 SignalMessages($t_w, v.id_{\sim_s}, v.id_{\sim_o}$);
18                 $arr_{id_{\sim_s}} \leftarrow$ MergeMessages($(t_w, id_{\sim_s})$);
19                 $arr_{id_{\sim_o}} \leftarrow$ MergeMessages($id_{\sim_o}$);
20                 $v.id_{\sim_s} \leftarrow$ Hash($arr_{id_{\sim_s}}$);
21                 $v.id_{\sim_o} \leftarrow$ Hash($arr_{id_{\sim_o}}$);
           /* Signal final messages. Update $v.id_{\sim_s}$ */
22         forall $v \in V$ do in parallel
23             SignalMessages($v.id_{\sim_o}$);
24             $arr_{id_{\sim_o}} \leftarrow$ MergeMessages($id_{\sim_o}$);
25             $v.id_{\sim_s} \leftarrow$ Hash($arr_{id_{\sim_o}}$);
       /* If $k = 1$: Only signal $(t_w, v.id_{\sim_o})$ and return */
26     else
27         forall $v \in V$ do in parallel
28             SignalMessages($t_w, v.id_{\sim_o}$);
29             $arr_{id_{\sim_o}} \leftarrow$ MergeMessages($(t_w, id_{\sim_o})$);
30             $v.id_{\sim_s} \leftarrow$ Hash($arr_{id_{\sim_o}}$);
31     $SG \leftarrow$ FindAndMerge($SG, V$);
32     return $SG$;
```

is updated by hashing the array (line 30). As a note, *HASH* is an order independent hash function, i.e. it computes the same value for the arrays $[1, 2]$ and $[2, 1]$. Moreover, whenever the algorithm updates the $v.id_{\sim_s}$ value for a vertex v, it first adds the old $v.id_{\sim_s}$ value of v so that this information is included in the new value. If $k > 1$, the algorithm operates slightly different (line 8 to line 25). In the first iteration (line 9 to line 14) every vertex v sends both, its $id_{\sim_s}$ and $id_{\sim_o}$ value along with the simple edge schema t_w to all its neighbours w (line 10). Subsequently, each vertex merges the incoming messages into (1) an array containing tuples with the received information $(t_w, id_{\sim_s})$ (line 11) and (2) an array containing tuples with the received information $(t_w, id_{\sim_o})$ (line 12). Finally, the $id_{\sim_s}$ value (line 13) and the $id_{\sim_o}$ value (line 14) of v are updated by hashing the corresponding array. In the next block (line 15 to line 21), the algorithm performs the same steps but excludes t_w when merging messages for $id_{\sim_o}$ (line 19). Including t_w is not necessary, as in iterations 2 to $k - 1$ the algorithm only compresses the received $id_{\sim}o$ values into one value using the hash function, so this information can be signaled further in the next iteration. In fact, the $id_{\sim}o$ values already contain the corresponding simple edge schema. This information was added in the first iteration. As a result, in the final iteration (line 22 to line 25), the only missing information to determine equivalence between two vertices v and v' is stored in their neighbours' $id_{\sim_o}$ values. Hence, each vertex first signals this information to its neighbours (line 23), then merge the incoming information (line 24) and ultimately compute their final $id_{\sim_s}$ value (line 25). At the end of execution, vertices with the same $id_{\sim_s}$ value are merged together (line 31).

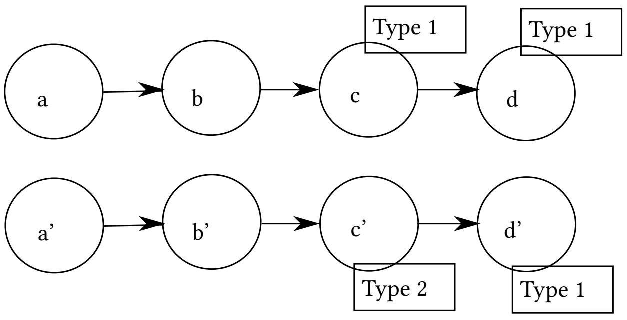


Figure 4: Chaining parameterisation example.

Example. The following example shall make clear, why the algorithm only incorporates the incoming $id_{\sim_o}$ values into a vertex's $id_{\sim_s}$ value in the final iteration, Recalling Definition 9, the chaining parameterisation nests the $CSE = (\sim_s, \sim_p, \sim_o)$ $(k - 1)$-times into the object relation $\sim_o$. For example, for $k = 3$ and $CSE = (T, T, OC_{type})$ this results in $CSE - 3 = cp(CSE, 3) = (T, T, (T, T, (T, T, OC_{type})))$. Two vertices v and v' are considered equal w.r.t. $CSE - 3$ iff for each neighbour w of v, there exists a respective neighbour w' of v', which have respective neighbours x and x', which have respective neighbours y and y' with the same type set. Consider the graph in Figure 4. Edge labels are omitted in the graph, as $CSE - 3$ defines $\sim_p$ to be the tautology. Vertices a and a' are considered equal under $CSE - 3$, because d and d' have an equal type set. The fact that c and c' have different type sets does not matter, as the OC_{type} relation applies to d and d' for the relation between a and a'. If now in iteration 1 the algorithm would incorporate the signalled values $c.id_{\sim_o}$ and $c'.id_{\sim_o}$ into the values $b.id_{\sim_s}$ and $b'.id_{\sim_s}$ respectively, the values of b and b' would differ. Consequently, in iteration 2, vertices a and a' would incorporate these different values into their respective values $a.id_{\sim_s}$ and $a'.id_{\sim_s}$, which results in inequality between a and a', which is wrong w.r.t. $CSE - 3$. The same holds true, if b and b' do not incorporate $c.id_{\sim_o}$ and $c'.id_{\sim_o}$ into $b.id_{\sim_s}$ and $b'.id_{\sim_s}$ in iteration 1, but a and a' incorporate $b.id_{\sim_o}$ and $b'.id_{\sim_o}$ (which contain the information of $c.id_{\sim_o}$ and $c'.id_{\sim_o}$) in iteration 2. Again, the final values would differ and hence a and a' would be considered unequal.

The experiments are executed on an Ubuntu 20 OS with 32 cores and 2 TB RAM. For execution, all 32 cores and 1.94 TB

Heap Space were given to the Apache Spark Context. Instrumentation was done using the Apache Spark Monitoring API, which provides several time and memory metrics.

4.4 Measures and Metrics

The scope is to evaluate the algorithms' performance. Hence, the two main performance indicators are the algorithms run time and memory consumption. More concrete, for every run of an experiment we report

(1) Total run time
(2) Run time of each of the k iterations
(3) Maximum JVM Memory OnHeap during execution

5 RESULTS

Table 8 lists the average total run time (minutes) for each experiment. As the (B,R,S) algorithm takes an additional initialisation step (Algorithm 5, line 9 to line 14), the composition of the algorithms total run time is reported in Table 9. It is separated into average run time for initialisation and computation of all ten iterations.[19] Furthermore, Table 10 lists the maximum JVM OnHeap Memory (GB) during execution for each experiment. Further, results are reported separately for smaller and larger datasets.

5.1 Smaller Datasets

Figure 5 shows plots of the average run time (minutes) for each of the ten iterations on the smaller datasets. As the iterations in the native Kaushik experiments take much longer than in the other experiments (Figures 5a, 5c and 5e), additional plots without the native Kaushik results are provided (Figures 5b, 5d and 5f). Such, differences between native Schätzle and (B,R,S) can be seen in more detail.

Starting with the (B,R,S) algorithm executed for the GSM of Schätzle, the algorithm shows a relatively constant iteration time on all smaller datasets. On Laundromat100M (Figure 5b), the algorithm computes an iteration in about 0.4 to 0.6 minutes. On BTC150M (Figure 5d) an iteration takes approximately 0.2 to 0.4 minutes and on BSBM100M (Figure 5f) 0.5 to 0.8 minutes. Concerning total run time (Table 8), the algorithm runs fastest on BTC150M taking about 4.08 minutes on average. On Laundromat100M the average run time is about 5.56 minutes and on BSBM100M the algorithm needs 6.46 minutes on average. On all smaller datasets the initialisation step takes about 1 minute on average (Table 9). Its counterpart for this GSM, the native Schätzle algorithm shows a similar behaviour across the datasets. The algorithm has a relatively constant iteration time on all datasets as well. On Laundromat100M (Figure 5b), the algorithm computes an iteration in about 0.7 to 0.95 minutes. On BTC150M (Figure 5d) an iteration takes approximately 0.55 to 0.8 minutes. Finally, on BSBM100M (Figure 5f) the average iteration time ranges from 0.9 to 1.15 minutes. Concerning average run time (Table 8), the algorithm performs fastest on BTC150M with 6.14 minutes on average. This is followed by an average run time of 7.72 minutes on Laundromat100M and 9.40 minutes on BSBM100M. Comparing the run times of the two algorithms, (B,R,S) is slightly faster on all smaller datasets. However, the native algorithm consumes less memory during execution on all smaller datasets (Table 10).

[19]For the native Kaushik algorithm, this distinction is not reported, as for its completed experiments, the initialisation step of partitioning the vertices based on their label (which is done in parallel, see discussion in Section 4.3) took less than 0.01 minutes.

The (B,R,S) algorithm executed for the GSM of Kaushik shows similar results as for the GSM of Schätzle. Again the algorithm has a relatively constant iteration time on all datasets taking about 0.3 to 0.6 minutes on Laundromat100M (Figure 5b) and 0.3 to 0.5 minutes on BTC150M and BSBM100M. Likewise to the GSM of Schätzle, the algorithm runs fastest on $BTC150M$ with an average run time of 4.54 minutes. On BSBM100M the average run time is about 5.20 minutes and on Laundromat100M the algorithm needs 5.60 minutes on average. Again, the initialisation step takes approximately 1 minutes on all smaller datasets (Table 9). Its counterpart for this GSM, the native Kaushik algorithm shows a different behaviour across the datasets. It does not show any constant iteration time on any dataset. Rather, the iteration time increases in early iterations until it reaches a maximum value, from which on the iteration time decreases again. The only exception of this behaviour occurs on BTC150M, where the iteration time decreases from iteration one to two. Afterwards, from iterations two to ten, it follows the described pattern. On Laundromat100M (Figure 5a) an iteration takes about 10 to 100 minutes. The maximum iteration time is reached in iteration six. On BTC150M (Figure 5c) the iteration time ranges from approximately 1 to 17 minutes. Here, the maximum iteration time is reached in iteration five. Finally, on BSBM100M (Figure 5e) the algorithm needs about 0.6 to 30 minutes to compute an iteration. The maximum iteration time is reached in iteration three on this dataset. Furthermore, the full bisimulation w.r.t. Definition 5 is reached and detected by the algorithm in iteration seven. The native Kaushik algorithm runs fastest on BSBM100M taking about 77.84 minutes on average. This is followed by an average run time of 78.02 minutes on BTC150M and 586.66 minutes on Laundromat100M. Comparing the run times of the two algorithms, (B,R,S) is significantly faster on all smaller datasets (Table 8). Moreover, (B,R,S) consumes slightly less memory than native Kaushik on all smaller datasets (Table 10).

In conclusion, (B,R,S) with GSM Schätzle computes the 10-bisimulation the fastest on BTC150M and Laundromat100M taking 4.08 and 5.56 minutes on average. On BSBM100M, the algorithm executed on GSM Kaushik runs the fastest taking 5.20 minutes on average. Further, the native Schätzle algorithm consumes the least memory on all smaller datasets.

5.2 Larger datasets

Figure 6 shows plots of the average run time (minutes) for each of the ten iterations on the larger datasets. The native Kaushik algorithm did not complete one iteration on BTC2B in 24 hours. Therefore, the execution on BTC2B was cancelled for this algorithm , as the completion was not feasible in terms of available time. Moreover, the algorithm ran out of memory on Laundromat15B and BSBM1B (Table 10). Hence, the plots in Figure 6 only provide results for (B,R,S) and native Schätzle.

The (B,R,S) algorithm executed for the GSM of Schätzle is capable of computing the 10-bisimulation on BTC2B and BSBM1B. On Laundromat15B, the algorithm runs out of memory in the initialisation step. For BTC2B and BSBM1B, the algorithm again shows a relatively constant iteration time. On BTC2B (Figure 6a) an iteration takes approximately 4 to 6 minutes and on BSBM100M (Figure 6b) 4 to 5 minutes. Concerning total run time (Tables 8 and 9), the algorithm runs fastest on BSBM1B taking 54.44 minutes on average, with 10.20 minutes for initialisation and 44.24 minutes for the 10 iterations. On BTC2B the algorithm needs

	(B,R,S)-Schätzle	Schätzle	(B,R,S)-Kaushik	Kaushik
Laundromat100M	**5.56**	7.72	5.60	586.66
BTC150M	**4.08**	6.14	4.54	78.02
BSBM100M	6.46	9.40	**5.20**	77.84
Laundromat15B	OOM	OOM	OOM	OOM
BTC2B	**61.96**	83.74	85.92	cancelled
BSBM1B	**54.44**	85.98	64.00	OOM

Table 8: Average total run time (minutes) for the k=10 bisimulation over 5 runs.

	(B,R,S)-Schätzle		(B,R,S)-Kaushik	
	Initialisation	Iterations	Initialisation	Iterations
Laundromat100M	1.10	4.46	1.10	4.50
BTC150M	1.47	2.61	1.44	3.10
BSBM100M	1.00	5.46	1.05	4.15
Laundromat15B	OOM	-	OOM	-
BTC2B	16.22	45.74	16.29	69.63
BSBM1B	10.20	44.24	10.43	53.57

Table 9: Composition (minutes) of average total run time for (B,R,S).

	(B,R,S)-Schätzle	Schätzle	(B,R,S)-Kaushik	Kaushik
Laundromat100M	211.5	147.9	210.5	335.1
BTC150M	140.6	107.7	130.1	181.7
BSBM100M	248.6	113.1	172.0	327.0
Laundromat15B	OOM	OOM	OOM	OOM
BTC2B	1249.1	1249.7	1249.3	cancelled
BSBM1B	1248.2	1335.4	1249.2	OOM

Table 10: Maximum JVM OnHeap Memory (GB) for the k=10 bisimulation over 5 runs.

16.22 minutes to initialise the graph and 45.74 minutes to compute the 10 iterations. This results in a total run time of 61.96 minutes on average. Native Schätzle is neither capable of computing the 10-bisimulation on Laundromat15B. The algorithm runs out of memory in the first iteration. For the other two datasets, again a nearly constant run time across iterations can be observed. On BTC2B (Figure 6a) an iteration takes approximately 8 minutes. On BSBM1B (Figure 5f) the average iteration time ranges from 8 to 9.5 minutes. Concerning the average run time for all iterations, the algorithm performs fastest on BTC2B with 83.74 minutes on average. This is followed by an average run time of 85.98 minutes on BTC2B. Comparing the run times of the two algorithms, (B,R,S) is slightly faster on the BSBM1B dataset and significantly faster on the BTC2B dataset. Both algorithms consume about the same amount of memory during execution on BTC2B. On BSBM1B native Schätzle consumes slightly more memory than (B,R,S).

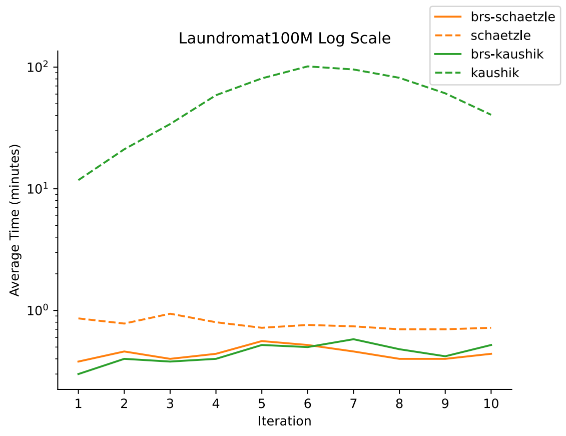
Finally, (B,R,S) executed for the GSM of Kaushik is also not able to compute the 10-bisimulation on Laundromat15B. Again, it runs out of memory in the initialisation step. Further, the algorithm only has a relatively constant iteration time on BSBM1B (Figure 6b). Here, the iteration time ranges from about 4 to 6 minutes. On BTC2B it shows a slightly different behaviour (Figure 6a). In iterations one to three the time ranges from 2 to 4 minutes. Afterwards, from iterations four to nine the algorithm constantly computes an iteration in about 8 minutes. Then in the final iteration ten, the running time slightly increases to about 9.5 minutes. Concerning total run time, (B,R,S)-Kaushik runs fastest on *BSBM1B* with 64 minutes on average taking 10.43 minutes for initialisation and 53.57 minutes for the 10 iterations. This is followed by 85.92 minutes on BTC2B with 16.29 minutes for initialisation and 69.93 minutes for the 10 iterations.

In conclusion, on BTC2B and BSBM1B (B,R,S) with GSM Schätzle computes the ten iterations the fastest taking 61.96 and 54.44 minutes on average respectively. Further, native Schätzle consumes slightly more memory on BSBM1B, whereas on BTC2B (B,R,S) and native Schätzle consume about the same amount of memory.

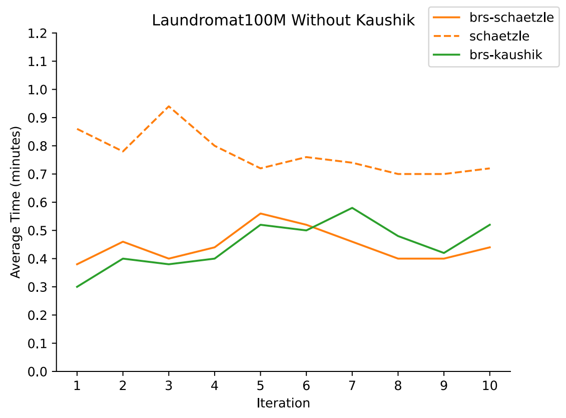
6 DISCUSSION

6.1 Main Results

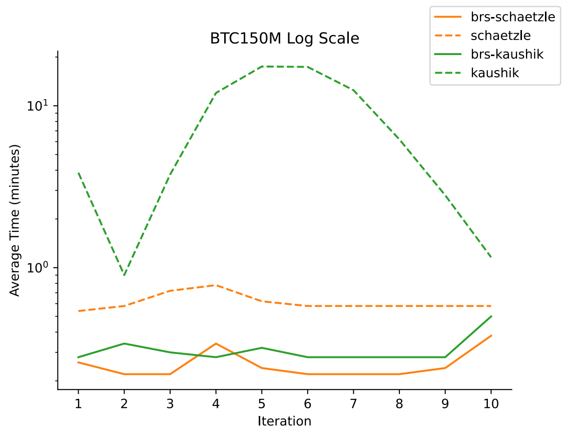
All algorithms are capable of computing 10-bisimulations on small synthetic and real-world graphs containing up to 150M triples. The parallel algorithms (B,R,S) and native Schätzle compute the 10-bisimulations in about 4.08 to 9.40 minutes on the smaller datasets. The sequential Kaushik algorithm needs far more time and only has an applicable run time on BTC150M and BSBM100M. There, the algorithm needs about 78 minutes on average. On Laundromat100M, the algorithm needs about 10 hours for the 10-bisimulation, which can not be considered an acceptable time for practical applications. A reason for the long running time on Laundromat100M could be the size of the initial partition P_0. It depends on the number of different label sets present in the graph's vertices (Definition 5). Laundromat100M contains 33 431 different label sets, which is much higher than for BTC150M with 69 and BSBM100M with 1 289 (Table 6). As a consequence, the algorithm has to perform stability checks and (potential) splits



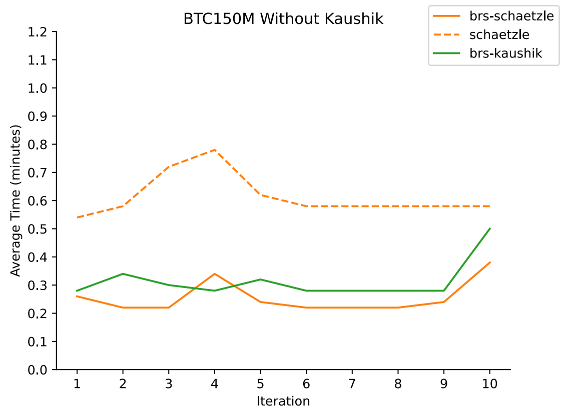
(a) Laundromat100 Log Scale



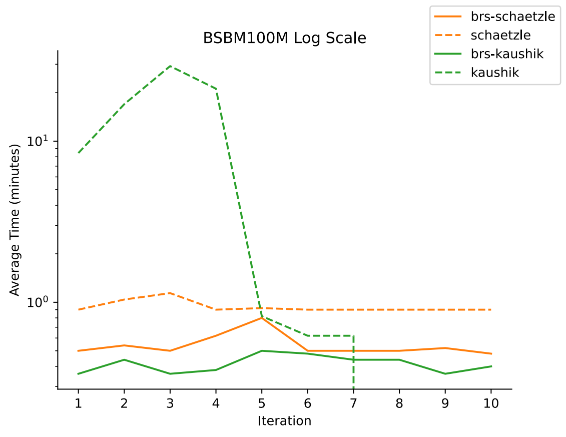
(b) Laundromat100 without Kaushik



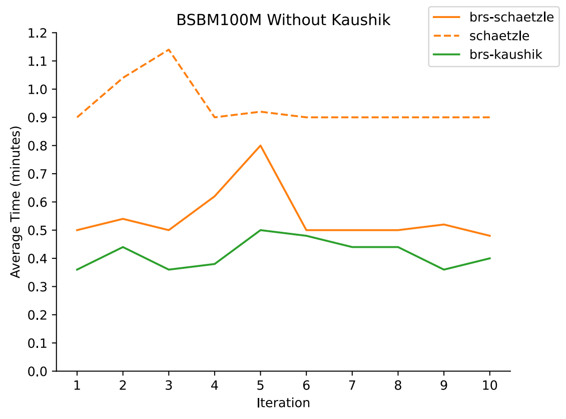
(c) BTC150M Log Scale



(d) BTC150M without Kaushik



(e) BSBM100M Log Scale

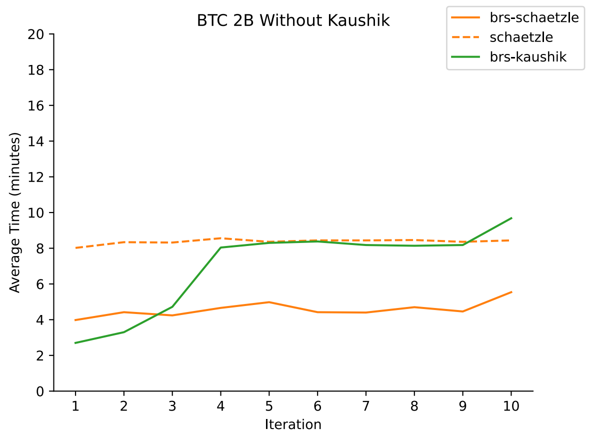


(f) BSBM100M without Kaushik

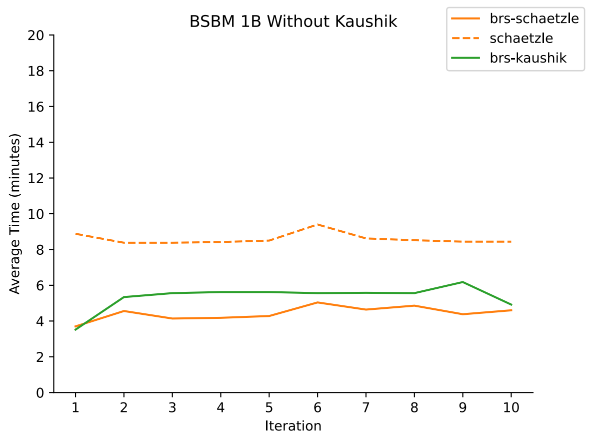
Figure 5: Average Iteration Times (minutes) of Experiments on Smaller Datasets.

on blocks more often than on the other datasets. This is not contradicting to the similar run times on BTC150M and BSBM100M (69 label sets vs. 1 289 label sets). First, BSBM100M reaches full bisimulation w.r.t. Definition 5 in iteration seven and hence execution is stopped early. Second, as it can be seen in Figure 5e, the iteration time starts to decrease rapidly from iteration four to five on BSBM100M, going down from about 20 minutes to about 1 minute. This indicates that the partition hardly changes any further and therefore only few stability checks and splits are performed in iterations five to seven.

The native Kaushik algorithm operates on the blocks of the graph's current partition. In each iteration i, the algorithm produces a partition P_i, which is stable w.r.t. all the blocks in P_{i-1} [19]. Consequently, if a block is split into two new blocks, the



(a) BTC2B without Kaushik



(b) BSBM1B without Kaushik

Figure 6: Average Iteration Times (minutes) of Experiments on Larger Datasets.

two new blocks have to be checked for stability in that iteration as well (see also discussion in Section 4.3). Hence, the more a partition increases in an iteration, the more steps have to be performed by the algorithm. In addition, bisimulation relationships between the vertices of a graph G change far more often in early iterations [32]. This explains the specific shape of the run time curve (Figures 5a, 5c and 5e) for the native Kaushik algorithm. The exception on BTC150M (Figure 5c), where the iteration time first decreases from iteration one to two, indicates that the 1- and 2-bisimulations of this graph are highly similar.

The parallel algorithms (B,R,S) and native Schätzle operate on the graph's vertices and edges. As the number of vertices and edges do not change during execution, the algorithms perform the same amount of steps in each iteration. Hence, they both show a more or less constant iteration time curve. The peaks in certain iterations could be due to the elimination of duplicates when merging incoming messages, which can vary in each iteration. Furthermore, both parallel algorithms perform significantly faster than the sequential Kaushik algorithm. (B,R,S) needs 4.08 to 5.60 and native Schätzle 6.14 to 9.40 minutes on the smaller datasets, whereas the sequential algorithm needs 77.84 to 586.66 minutes. Hence, even for smaller graphs of $100M$ to $150M$ triples, it is far more applicable to use a parallel algorithm for computing bisimulation reductions. For larger graphs of $1B$ to $2B$ triples, it is infeasible to use a sequential algorithm for practical applications.

Beyond, the parallel algorithms perform well on both the synthetic BSBM1B graph and the real-world BTC2B graph. Results indicate that the native algorithm does not have any advantage over the generic one, as (B,R,S) outperforms native Schätzle on every dataset. However, the comparison between (B,R,S) and native Schätzle could potentially have a bias, which was introduced in the adjustment of the (B,R,S) algorithm (Section 4.3), see Section 6.2 for further discussion. Moreover, native Schätzle performs a *full-bisimulation-check* in each iteration (line 7 in Algorithm 2). To check for full bisimulation, the algorithm counts the distinct block identifiers of the current iteration. This step was necessary in the original implementation, as they computed full bisimulations with their algorithm. In the experiments [32], their algorithm was evaluated on different versions of three benchmarks and on one real world graph. On the synthetic datasets, full bisimulation was reached in iterations four (SP^{2}Bench), seven (LUBM) and twelve to fourteen (SIB). The real-world graph (DBPedia version 3.7) reached full bisimulation in iteration 37. Hence, together with the results provided here, where no graph reaches full bisimulation w.r.t. Definition 4 in the first 10 iterations, this step can be considered as unnecessary for computing stratified bisimulations, especially on real-world graphs.

Both, (B,R,S) and native Schätzle seem to scale linearly with the input graph's size. BTC150M contains approximately $5M$ vertices and $145M$ edges. BTC2B is comprised of approximately $80M$ vertices and $2B$ edges, which is a factor of about 18 and 14 respectively. (B,R,S)-Schätzle takes 4.08 minutes, (B,R,S)-Kaushik 4.54 minutes and native Schätzle 6.14 minutes on the smaller version. On the larger version, they take 15 times, 19 times and 14 times longer. The size factor of BSBM1B to BSBM100M is about 10 for the number of vertices and edges. The experiments on BSBM1B took about 10 times longer for (B,R,S)-Schätzle, 12 times for (B,R,S)-Kaushik and 9 times for native Schätzle. Due to the OOM on Laundromat15B, a comparison of the run times with Laundromat100M can not be given. Experimental results for native Kaushik indicate that the algorithm does not scale linearly with the input graph's size. In fact, the worst-case complexity provided by the authors, which is said to be $O(k \cdot m)$, where m corresponds to the number of edges in the input graph, seems to be incorrect. As the algorithm operates on blocks, one should rather express the worst-case complexity in terms of the maximum number of blocks. In the worst case, the initial partition P_0 contains $n = |V|$ blocks, which corresponds to the scenario where each vertex has a different label set. Then, the algorithm checks for each of the n blocks, if it is stable to each of the n block-copies. This would be the only iteration, as in a partition of size n no block can be split. Hence, the worst-case complexity of Algorithm 4 is given by $O(n^2)$, where n is the number of vertices in the input graph. Note that in practice, this is very unusual. Rather, in practice, the worst-case complexity may be given by $O(b_{max} \cdot |P_i| \cdot k)$, where b_{max} corresponds to the maximum number of blocks (that have not been used as a splitter before) present in some iteration $i \in \{1, 2, \ldots, k\}$.

Lastly, the experiments for (B,R,S) and native Schätzle on Laundromat15B emphasise the need for external memory solutions on very large graphs. The algorithms cache the graph of an iteration i in memory for the next iteration $i + 1$, until iteration $i + 1$ is finished. Therefore, at some point in each iteration i, the graph of the previous iteration $i-1$ and the current graph are cached in memory, which causes problems for very large graphs. However, the Apache Spark API provides the utility to persist graphs on disk during execution, if they do not fit into memory. This would enable summarising large graphs like Laundromat15B without running out of memory (in case enough disk space is available).

6.2 Threat to Validity

All algorithms are implemented in Scala upon the same framework. Moreover, every algorithm was executed using the same procedure on a machine with exclusive access. Hence, the experiments for each of the algorithms were all settled in the same environment. However, in the adjustment process of the (B,R,S) algorithm (see Section 4.3), heavily nested data structures were removed from the algorithm. Additionally, all collection types (e.g., sets or maps) were omitted for signaling and merging messages. Instead, as recommended by the Spark Tuning Guide,[20] arrays of objects (tuples) and primitives (integers) were used. For native Schätzle, the reimplementation follows the implementation provided in the original work, in which they use sets of tuples for sending and merging messages. This fact may be a bias in the experimental results. It could be possible that an adjusted version of native Schätzle, which makes use of arrays of tuples for sending and merging messages, does not get outperformed by (B,R,S). However, this potential bias only effects the comparison between (B,R,S) and native Schätzle. The fact that both algorithms compute 10-bisimulations on real-world graphs of up to $2B$ triples in an applicable time, still holds true. Moreover, the fact that the generic (B,R,S) algorithm can compete with its native counterpart for a specific GSM, still holds true as well.

6.3 Generalisability

The results generalise because of following reasons. First, in the experiments synthetic and real-world graphs were considered, where especially the latter is important when analysing the practical application of an algorithm [5, 25]. Second, two different GSMs were used for evaluation of the (B,R,S) algorithm. The GSM of Schätzle computes a stratified forward bisimulation and additionally incorporates edge labels for determining equivalence (Definition 4). The GSM of Kaushik computes a stratified backward bisimulation and additionally incorporates vertex labels for determining equivalence (Definition 5). Hence, the two GSMs consider different structural features for determining vertex equivalence. Regardless, for both GSMs, (B,R,S) scales linearly with the graph's input size and further computes the 10-bisimulations the fastest on every dataset. As a note, the results provided here should only be considered generalisable for computations using memory only, as the experiments were executed on memory only. When using external memory, all algorithms will take longer to compute the 10-bisimulations.

6.4 Future Work

The provided results as well as the adjusted (B,R,S) algorithm (Algorithm 5) can be used as a foundation for several future works. First, it would be interesting to see the performance of the adjusted (B,R,S) for more GSMs that can be found in existing literature. [6] provides a large overview of existing GSMs that could be evaluated for certain values of k. Moreover, the experiment on Laundromat15B shows that the algorithm's performance when using external memory is of great interest as well. Therefore, only a small change in the algorithms implementation is necessary. Instead of caching the graphs only in memory during execution, the graphs would also be persisted on disk, if they do not fit into memory. An extensive study on large synthetic and real-world graphs of at least $5B$ triples could provide meaningful insights of the algorithms performance when using this mechanism. Further, the concept of operating on blocks during execution, rather than on vertices and edges, which is applied by the sequential Kaushik algorithm, could be evaluated for a parallel implementation. Rajasekaran and Lee [31] provide a parallel version of Paige and Tarjan's [30] optimised algorithm for computing relational coarsest partition problems (Definition 19). One would have to adapt the implementation to a stratified bisimulation, as the relational coarsest partition corresponds to a full bisimulation [18]. Alternatively, Algorithm 4 could be adjusted in the way described in Section 4.3. Furthermore, adjustments on (B,R,S), which are described in Section 4.3, removed the feature of incrementally updating an existing graph summary [5]. As vertex information is now stored in a numerical value, the information can no longer be accessed and therefore the numerical values cannot be adapted to the changes of the input graph. However, the adjusted (B,R,S) algorithm is designed for computations using the chaining parameterisation (Definition 9). Trying to incrementally update a graph summary that was obtained by applying the chaining parameterisation is quite challenging. A change in the input graph can cause a potential change of a vertex's id values in each iteration $i \in \{1, 2, \ldots, k\}$. Moreover, changes in an iteration i can potentially have an effect on all subsequent iterations $j > i$ and hence ultimately on the final relation between any two vertices v and v'. This makes incremental updates when using the chaining parameterisation quite complex.

7 CONCLUSION

The experimental results show that computing stratified bisimulations on large synthetic and real-world graphs of up to $2B$ triples is computationally feasible. Signalling and storing vertex information should be done using a numerical value for the vertex information, as nested data structures can cause heavy memory issues during execution. Therefore, an adjusted version of the (B,R,S) algorithm is provided. Moreover, parallel algorithms perform significantly faster than sequential ones, and hence one should make use of a parallel algorithm when computing stratified bisimulations. For very large graphs of $15B$ triples, an external memory solution is desirable. Without using external memory, the algorithms run out of memory during execution. Beyond, experimental results indicate the the parallel algorithms (B,R,S) and native Schätzle scale linearly with the input graph's size. Further, native algorithms do not have any advantage over the generic (B,R,S) algorithm, as (B,R,S) outperforms its native counterparts on every dataset, though its generic nature.

ACKNOWLEDGEMENT

We thank (Dr.) Till Blume for the very helpful discussions and support in implementing the k-length bi-simulation in FLUID.

REFERENCES

[1] Wouter Beek, Laurens Rietveld, Hamid R. Bazoobandi, Jan Wielemaker, and Stefan Schlobach. 2014. LOD Laundromat: A Uniform Way of Publishing Other People's Dirty Data. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I (Lecture Notes in Computer Science, Vol. 8796)*, Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic,

[20] https://spark.apache.org/docs/latest/tuning.html

Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble (Eds.). Springer, 213–228. https://doi.org/10.1007/978-3-319-11964-9_14

[2] Fabio Benedetti, Sonia Bergamaschi, and Laura Po. 2015. Exposing the Underlying Schema of LOD Sources. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, WI-IAT 2015, Singapore, December 6-9, 2015 - Volume I*. IEEE Computer Society, 301–304. https://doi.org/10.1109/WI-IAT.2015.99

[3] Christian Bizer and Andreas Schultz. 2009. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.* 5, 2 (2009), 1–24. https://doi.org/10.4018/jswis.2009040101

[4] Stefan Blom and Simona Orzan. 2002. A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. *Electron. Notes Theor. Comput. Sci.* 68, 4 (2002), 523–538. https://doi.org/10.1016/S1571-0661(05)80390-1

[5] Till Blume, David Richerby, and Ansgar Scherp. 2020. Incremental and Parallel Computation of Structural Graph Summaries for Evolving Graphs. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux (Eds.). ACM, 75–84. https://doi.org/10.1145/3340531.3411878

[6] Till Blume, David Richerby, and Ansgar Scherp. 2021. FLUID: A common model for semantic structural graph summaries based on equivalence relations. *Theor. Comput. Sci.* 854 (2021), 136–158. https://doi.org/10.1016/j.tcs.2020.12.019

[7] Angela Bonifati, Stefania Dumbrava, and Haridimos Kondylakis. 2020. Graph Summarization. *CoRR* abs/2004.14794 (2020). arXiv:2004.14794 https://arxiv.org/abs/2004.14794

[8] Stéphane Campinas, Thomas Perry, Diego Ceccarelli, Renaud Delbru, and Giovanni Tummarello. 2012. Introducing RDF Graph Summary with Application to Assisted SPARQL Formulation. In *23rd International Workshop on Database and Expert Systems Applications, DEXA 2012, Vienna, Austria, September 3-7, 2012*, Abdelkader Hameurlain, A Min Tjoa, and Roland R. Wagner (Eds.). IEEE Computer Society, 261–266. https://doi.org/10.1109/DEXA.2012.38

[9] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2019. Summarizing semantic graphs: a survey. *VLDB J.* 28, 3 (2019), 295–327. https://doi.org/10.1007/s00778-018-0528-3

[10] Marek Ciglan, Kjetil Nørvåg, and Ladislav Hluchý. 2012. The SemSets model for ad-hoc semantic list search. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, Alain Mille, Fabien Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab (Eds.). ACM, 131–140. https://doi.org/10.1145/2187836.2187855

[11] Mariano P. Consens, Valeria Fionda, Shahan Khatchadourian, and Giuseppe Pirrò. 2015. S+EPPs: Construct and Explore Bisimulation Summaries, plus Optimize Navigational Queries; all on Existing SPARQL Systems. *Proc. VLDB Endow.* 8, 12 (2015), 2028–2031. https://doi.org/10.14778/2824032.2824128

[12] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. Accessed: 2021-04-16.

[13] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 137–150. http://www.usenix.org/events/osdi04/tech/dean.html

[14] François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. 2020. RDF graph summarization for first-sight structure discovery. *VLDB J.* 29, 5 (2020), 1191–1218. https://doi.org/10.1007/s00778-020-00611-y

[15] Roy Goldman and Jennifer Widom. 1997. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (Eds.). Morgan Kaufmann, 436–445. http://www.vldb.org/conf/1997/P436.PDF

[16] José-Miguel Herrera, Aidan Hogan, and Tobias Käfer. 2019. BTC-2019: The 2019 Billion Triple Challenge Dataset. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11779)*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.). Springer, 163–180. https://doi.org/10.1007/978-3-030-30796-7_11

[17] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, Roberto Navigli, Axel-Cyrille Ngonga Ngomo, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. 2020. Knowledge Graphs. *CoRR* abs/2003.02320 (2020). arXiv:2003.02320 https://arxiv.org/abs/2003.02320

[18] Paris C. Kanellakis and Scott A. Smolka. 1990. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Inf. Comput.* 86, 1 (1990), 43–68. https://doi.org/10.1016/0890-5401(90)90025-D

[19] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. 2002. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, Rakesh Agrawal and Klaus R. Dittrich (Eds.). IEEE Computer Society, 129–140. https://doi.org/10.1109/ICDE.2002.994703

[20] Kenza Kellou-Menouer and Zoubida Kedad. 2015. Schema Discovery in RDF Data Sources. In *Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19-22, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9381)*, Paul Johannesson, Mong-Li Lee, Stephen W. Liddle, Andreas L. Opdahl, and Oscar Pastor López (Eds.). Springer, 481–495. https://doi.org/10.1007/978-3-319-25264-3_36

[21] Kifayat-Ullah Khan, Waqas Nawaz, and Young-Koo Lee. 2015. Set-based approximate approach for lossless graph summarization. *Computing* 97, 12 (2015), 1185–1207. https://doi.org/10.1007/s00607-015-0454-9

[22] Shahan Khatchadourian and Mariano P. Consens. 2010. ExpLOD: Summary-Based Exploration of Interlinking and RDF Usage in the Linked Open Data Cloud. In *The Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 30 - June 3, 2010, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 6089)*, Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache (Eds.). Springer, 272–287. https://doi.org/10.1007/978-3-642-13489-0_19

[23] Mathias Konrath, Thomas Gottron, Steffen Staab, and Ansgar Scherp. 2012. SchemEX - Efficient construction of a data catalogue by stream-based indexing of linked data. *J. Web Semant.* 16 (2012), 52–58. https://doi.org/10.1016/j.websem.2012.06.002

[24] Kristen LeFevre and Evimaria Terzi. 2010. GraSS: Graph Structure Summarization. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2010, April 29 - May 1, 2010, Columbus, Ohio, USA*. SIAM, 454–465. https://doi.org/10.1137/1.9781611972801.40

[25] Yongming Luo, George H. L. Fletcher, Jan Hidders, Paul De Bra, and Yuqing Wu. 2013. Regularities and dynamics in bisimulation reductions of big graphs. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, Peter A. Boncz and Thomas Neumann (Eds.). CWI/ACM, 13. https://doi.org/10.1145/2484425.2484438

[26] Yongming Luo, George H. L. Fletcher, Jan Hidders, Yuqing Wu, and Paul De Bra. 2013. External memory K-bisimulation reduction of big graphs. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi (Eds.). ACM, 919–928. https://doi.org/10.1145/2505515.2505752

[27] Tova Milo and Dan Suciu. 1999. Index Structures for Path Expressions. In *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1540)*, Catriel Beeri and Peter Buneman (Eds.). Springer, 277–295. https://doi.org/10.1007/3-540-49257-7_18

[28] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. 2008. Graph summarization with bounded error. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 419–432. https://doi.org/10.1145/1376616.1376661

[29] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 984–994. https://doi.org/10.1109/ICDE.2011.5767868

[30] Robert Paige and Robert Endre Tarjan. 1987. Three Partition Refinement Algorithms. *SIAM J. Comput.* 16, 6 (1987), 973–989. https://doi.org/10.1137/0216062

[31] Sanguthevar Rajasekaran and Insup Lee. 1998. Parallel Algorithms for Relational Coarsest Partition Problems. *IEEE Trans. Parallel Distributed Syst.* 9, 7 (1998), 687–699. https://doi.org/10.1109/71.707548

[32] Alexander Schätzle, Antony Neu, Georg Lausen, and Martin Przyjaciel-Zablocki. 2013. Large-scale bisimulation of RDF graphs. In *Proceedings of the Fifth Workshop on Semantic Web Information Management, SWIM@SIGMOD Conference 2013, New York, NY, USA, June 23, 2013*, Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca (Eds.). ACM, 1:1–1:8. https://doi.org/10.1145/2484712.2484713

[33] Qi Song, Yinghui Wu, and Xin Luna Dong. 2016. Mining Summaries for Knowledge Graph Search. In *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, Francesco Bonchi, Josep Domingo-Ferrer, Ricardo Baeza-Yates, Zhi-Hua Zhou, and Xindong Wu (Eds.). IEEE Computer Society, 1215–1220. https://doi.org/10.1109/ICDM.2016.0162

[34] Blerina Spahiu, Riccardo Porrini, Matteo Palmonari, Anisa Rula, and Andrea Maurino. 2016. ABSTAT: Ontology-driven Linked Data Summaries with Pattern Minimalization. In *Proceedings of the 2nd International Workshop on Summarizing and Presenting Entities and Ontologies (SumPre 2016) co-located with the 13th Extended Semantic Web Conference (ESWC 2016), Anissaras, Greece, May 30, 2016 (CEUR Workshop Proceedings, Vol. 1605)*, Andreas Thalhammer, Gong Cheng, and Kalpa Gunaratna (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-1605/paper3.pdf

[35] Robert Endre Tarjan and Jan van Leeuwen. 1984. Worst-case Analysis of Set Union Algorithms. *J. ACM* 31, 2 (1984), 245–281. https://doi.org/10.1145/62.2160

[36] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. 2008. Efficient aggregation for graph summarization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver,*

BC, Canada, June 10-12, 2008, Jason Tsong-Li Wang (Ed.). ACM, 567–580. https://doi.org/10.1145/1376616.1376675

[37] Thanh Tran, Günter Ladwig, and Sebastian Rudolph. 2013. Managing Structured and Semistructured RDF Data Using Structure Indexes. *IEEE Trans. Knowl. Data Eng.* 25, 9 (2013), 2076–2089. https://doi.org/10.1109/TKDE.2012.134

[38] Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. 2007. GRIN: A Graph Based RDF Index. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada.* AAAI Press, 1465–1470. http://www.aaai.org/Library/AAAI/2007/aaai07-232.php

Supplementary Materials

A MAPPINGS OF THE NOTATIONS USED BY KAUSHIK ET AL. AND PAIGE AND TARJAN

In Table 11 the notations used by Kaushik et al. and Paige and Tarjan are mapped to a single notation.

B RESULTS OF EXPERIMENTS

The detailed experimental results are provided as tables. Furthermore, the results are split into results for smaller datasets (Laundromat100M, BTC150M, BSBM100M) located in Section B.1 and results for larger datasets (BTC2B, BSBM1B) located in Section B.2. The tables for (B,R,S) include an additional iteration 0, which corresponds to the initialisation routine of the algorithm.

B.1 Smaller Datasets

See Tables 12 to 14.

B.2 Larger Datasets

See Tables 15 and 16

Notation	Meaning	Original	
		Paige and Tarjan	Kaushik
V	Finite set on which the relational coarsest partition will be computed (interpreted as a set of vertices in this work)	U	V_G
$E \subseteq V \times V$	Binary relation over the finite set V (interpreted as a set of directed edges between the vertices in this work)	E	E_G
P_i and P_i^{copy}	Partition of V and its copy in iteration $i \in \{0, 1, 2, \ldots\}$, where P_0 is the initial Partition	P and Q	$\mathcal{Q}$ and $\mathcal{X}$
$S \subseteq V$	Any subset of the finite set V	S	A and B
$T^{+}(S) = \{y \mid \exists x \in S \text{ with } xEy \text{ and } y \in V\}$	The set of elements, which are related to any element $x \in S$ (if one interprets V as vertices and E as edges, then this refers to the **successors** of the vertices in S)	$E(S)$	$Succ(A)$
$T^{-}(S) = \{x \mid \exists y \in S \text{ with } xEy \text{ and } x \in V\}$	The set of elements, which have any related element $y \in S$ (if one interprets V as vertices and E as edges, then this refers to the **predecessors** of the vertices in S)	$E^{-1}(S)$	Not used but would be $Pred(A)$
B_{ij}	Block j of Partition P_i	B	Q and X

Table 11: Notations used in this work for the approach of Kaushik et al.

(a) (B,R,S) algorithm executed with GSM Schaetzle

Run	**Iteration**											**Aggregates**		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	1.1	0.4	0.5	0.5	0.4	0.4	0.6	0.6	0.5	0.5	0.5	6.0	0.49	0.07
2	1.1	0.3	0.4	0.3	0.3	0.3	0.6	0.4	0.3	0.3	0.3	4.6	0.35	0.09
3	1.1	0.4	0.5	0.4	0.7	0.6	0.5	0.4	0.4	0.4	0.4	5.8	0.47	0.1
4	1.1	0.4	0.5	0.4	0.4	0.7	0.5	0.5	0.4	0.4	0.5	5.8	0.47	0.09
5	1.1	0.4	0.4	0.4	0.4	0.8	0.4	0.4	0.4	0.4	0.5	5.6	0.45	0.12
μ	1.1	0.38	0.46	0.4	0.44	0.56	0.52	0.46	0.4	0.4	0.44	**5.56**		
σ	0.0	0.04	0.05	0.06	0.14	0.19	0.07	0.08	0.06	0.06	0.08	**0.5**		

(b) Schaetzle algorithm

Run	**Iteration**										**Aggregates**		
	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	1.1	0.8	0.7	0.8	0.8	0.7	0.7	0.7	0.7	0.8	7.8	0.78	0.12
2	1.1	0.8	0.7	0.8	0.7	0.7	0.7	0.7	0.7	0.7	7.6	0.76	0.12
3	0.7	0.8	1.1	0.8	0.7	0.8	0.8	0.7	0.7	0.7	7.8	0.78	0.12
4	0.7	0.8	1.1	0.8	0.7	0.8	0.7	0.7	0.7	0.7	7.7	0.77	0.12
5	0.7	0.7	1.1	0.8	0.7	0.8	0.8	0.7	0.7	0.7	7.7	0.77	0.12
μ	0.86	0.78	0.94	0.8	0.72	0.76	0.74	0.7	0.7	0.72	**7.72**		
σ	0.0	0.0	0.0	0.08	0.04	0.07	0.0	0.0	0.05	0.04	**0.07**		

(c) (B,R,S) algorithm executed with GSM Kaushik

Run	**Iteration**											**Aggregates**		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	1.1	0.3	0.4	0.4	0.4	0.4	0.4	0.6	0.7	0.5	0.5	5.7	0.46	0.11
2	1.1	0.3	0.4	0.3	0.3	0.4	0.5	0.7	0.4	0.4	0.5	5.3	0.42	0.12
3	1.1	0.3	0.4	0.4	0.5	0.7	0.5	0.5	0.4	0.4	0.5	5.7	0.46	0.1
4	1.1	0.3	0.4	0.4	0.4	0.4	0.7	0.6	0.5	0.4	0.5	5.7	0.46	0.11
5	1.1	0.3	0.4	0.4	0.4	0.7	0.4	0.5	0.4	0.4	0.6	5.6	0.45	0.11
μ	1.1	0.3	0.4	0.38	0.4	0.52	0.5	0.58	0.48	0.42	0.52	**5.6**		
σ	0.0	0.0	0.0	0.04	0.06	0.15	0.11	0.07	0.12	0.04	0.04	**0.15**		

(d) Kaushik algorithm

Run	**Iteration**										**Aggregates**		
	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	11.4	21.2	35.1	60.0	85.5	106.7	98.8	81.8	59.3	44.1	603.9	60.39	30.9
2	13.6	22.7	35.8	66.8	76.8	91.9	85.6	69.6	52.1	35.0	549.9	54.99	25.81
3	10.9	19.3	31.9	53.7	81.3	100.1	98.7	79.9	62.5	42.5	580.8	58.08	30.21
4	11.2	21.1	33.3	58.2	78.6	100.5	98.1	85.1	61.6	41.2	588.9	58.89	30.04
5	11.8	21.3	34.1	55.1	81.8	107.7	97.0	91.9	69.0	40.1	609.8	60.98	31.81
μ	11.78	21.12	34.04	58.76	80.8	101.38	95.64	81.66	60.9	40.58	**586.66**		
σ	0.96	1.08	1.37	4.59	2.97	5.67	5.06	7.28	5.45	3.09	**21.09**		

Table 12: Detailed Results (minutes) on Laundromat100M for a 10-bisimulation.

(a) (B,R,S) algorithm executed with GSM Schaetzle

Run	**Iteration**											**Aggregates**		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	1.4	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	2.4	0.1	0.0
2	1.4	0.4	0.4	0.4	0.8	0.5	0.4	0.4	0.4	0.5	0.4	6.0	0.46	0.12
3	1.7	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.6	3.3	0.16	0.15
4	1.4	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.4	2.8	0.14	0.09
5	1.7	0.4	0.4	0.4	0.6	0.4	0.4	0.4	0.4	0.4	0.4	5.9	0.42	0.06
μ	1.47	0.26	0.22	0.22	0.34	0.24	0.22	0.22	0.22	0.24	0.38	**4.08**		
σ	0.14	0.12	0.15	0.15	0.3	0.17	0.15	0.15	0.15	0.17	0.16	**1.55**		

(b) Schaetzle algorithm

Run	**Iteration**										**Aggregates**		
	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	0.6	0.6	0.7	0.8	0.7	0.6	0.6	0.6	0.6	0.6	6.4	0.64	0.07
2	0.5	0.6	0.7	0.8	0.6	0.6	0.6	0.6	0.6	0.6	6.2	0.62	0.07
3	0.6	0.6	0.8	0.8	0.6	0.6	0.6	0.6	0.6	0.6	6.4	0.64	0.08
4	0.5	0.6	0.8	0.8	0.6	0.6	0.6	0.6	0.6	0.6	6.3	0.63	0.09
5	0.5	0.5	0.6	0.7	0.6	0.5	0.5	0.5	0.5	0.5	5.4	0.54	0.07
μ	0.54	0.58	0.72	0.78	0.62	0.58	0.58	0.58	0.58	0.58	**6.14**		
σ	0.05	0.04	0.07	0.04	0.04	0.04	0.04	0.04	0.04	0.04	**0.38**		

(c) (B,R,S) algorithm executed with GSM Kaushik

Run	**Iteration**											**Aggregates**		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	1.4	0.4	0.4	0.3	0.4	0.6	0.4	0.4	0.4	0.4	0.4	5.5	0.41	0.07
2	1.4	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.7	3.0	0.16	0.18
3	1.4	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.5	2.8	0.14	0.12
4	1.4	0.4	0.5	0.5	0.4	0.4	0.4	0.4	0.4	0.4	0.4	5.6	0.42	0.04
5	1.4	0.4	0.6	0.5	0.4	0.4	0.4	0.4	0.4	0.4	0.5	5.8	0.44	0.07
μ	1.44	0.28	0.34	0.3	0.28	0.32	0.28	0.28	0.28	0.28	0.5	**4.54**		
σ	0.14	0.15	0.21	0.18	0.15	0.19	0.15	0.15	0.15	0.15	0.11	**1.34**		

(d) Kaushik algorithm

Run	**Iteration**										**Aggregates**		
	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	3.9	0.9	3.7	12.2	17.0	17.8	12.7	6.6	2.8	1.0	78.6	7.86	6.17
2	3.8	0.9	3.7	11.2	16.8	17.0	12.9	6.0	2.9	1.5	76.7	7.67	5.92
3	4.1	0.9	3.9	11.7	17.6	17.6	12.6	6.1	3.1	1.1	78.7	7.87	6.14
4	3.9	0.9	4.0	14.1	20.5	18.5	12.2	6.0	2.5	1.2	83.8	8.38	6.95
5	3.6	0.9	3.5	10.8	15.5	16.0	12.0	6.3	2.7	1.0	72.3	7.23	5.55
μ	3.86	0.9	3.76	12.0	17.48	17.38	12.48	6.2	2.8	1.16	**78.02**		
σ	0.16	0.0	0.17	1.15	1.66	0.84	0.33	0.23	0.2	0.19	**3.71**		

Table 13: Detailed Results (minutes) on BTC150M for a 10-bisimulation.

(a) (B,R,S) algorithm executed with GSM Schaetzle

Run	**Iteration**											**Aggregates**		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	1.0	0.5	0.5	0.5	0.6	0.8	0.5	0.5	0.5	0.5	0.4	6.3	0.53	0.1
2	1.0	0.5	0.6	0.5	0.5	0.9	0.5	0.5	0.5	0.5	0.5	6.5	0.55	0.12
3	1.0	0.5	0.6	0.5	0.5	0.9	0.5	0.5	0.5	0.5	0.5	6.5	0.55	0.12
4	1.0	0.5	0.5	0.5	0.6	0.9	0.5	0.5	0.5	0.5	0.5	6.5	0.55	0.12
5	1.0	0.5	0.5	0.5	0.9	0.5	0.5	0.5	0.5	0.6	0.5	6.5	0.55	0.12
μ	1.0	0.5	0.54	0.5	0.62	0.8	0.5	0.5	0.5	0.52	0.48	**6.46**		
σ	0.0	0.0	0.05	0.0	0.15	0.15	0.0	0.0	0.0	0.04	0.04	**0.08**		

(b) Schaetzle algorithm

Run	**Iteration**										**Aggregates**		
	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	0.9	1.3	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	9.4	0.94	0.12
2	0.9	1.0	1.2	0.9	1.0	0.9	0.9	0.9	0.9	0.9	9.5	0.95	0.09
3	0.9	0.9	1.2	0.9	0.9	0.9	0.9	0.9	0.9	0.9	9.3	0.93	0.09
4	0.9	1.0	1.2	0.9	0.9	0.9	0.9	0.9	0.9	0.9	9.4	0.94	0.09
5	0.9	1.0	1.2	0.9	0.9	0.9	0.9	0.9	0.9	0.9	9.4	0.94	0.09
μ	0.9	1.04	1.14	0.9	0.92	0.9	0.9	0.9	0.9	0.9	**9.4**		
σ	0.0	0.14	0.12	0.0	0.04	0.0	0.0	0.0	0.0	0.0	**0.06**		

(c) (B,R,S) algorithm executed with GSM Kaushik

Run	**Iteration**											**Aggregates**		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	1.0	0.4	0.5	0.4	0.4	0.6	0.7	0.4	0.4	0.4	0.4	5.6	0.46	0.1
2	1.0	0.3	0.4	0.3	0.3	0.3	0.3	0.5	0.5	0.3	0.3	4.5	0.35	0.08
3	1.1	0.4	0.4	0.4	0.5	0.7	0.5	0.4	0.4	0.4	0.5	5.7	0.46	0.09
4	1.0	0.4	0.5	0.4	0.4	0.6	0.6	0.4	0.4	0.4	0.4	5.5	0.45	0.08
5	1.1	0.3	0.4	0.3	0.3	0.3	0.3	0.5	0.5	0.3	0.4	4.7	0.36	0.08
μ	1.05	0.36	0.44	0.36	0.38	0.5	0.48	0.44	0.44	0.36	0.4	**5.2**		
σ	0.05	0.05	0.05	0.05	0.07	0.17	0.16	0.05	0.05	0.05	0.06	**0.5**		

(d) Kaushik algorithm

Run	**Iteration**										**Aggregates**		
	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	8.0	16.5	33.9	19.0	0.8	0.6	0.6	0.0	0.0	0.0	79.4	7.94	11.03
2	8.8	17.2	29.3	22.1	0.9	0.6	0.6	0.0	0.0	0.0	79.5	7.95	10.44
3	8.2	16.2	30.7	22.6	0.8	0.7	0.7	0.0	0.0	0.0	79.9	7.99	10.71
4	8.0	19.0	26.4	21.5	0.8	0.6	0.6	0.0	0.0	0.0	76.9	7.69	9.97
5	9.2	15.9	25.8	20.6	0.8	0.6	0.6	0.0	0.0	0.0	73.5	7.35	9.43
μ	8.44	16.96	29.22	21.16	0.82	0.62	0.62	0.0	0.0	0.0	**77.84**		
σ	0.48	1.11	2.96	1.27	0.04	0.04	0.04	0.0	0.0	0.0	**2.41**		

Table 14: Detailed Results (minutes) on BSBM100M for a 10-bisimulation.

(a) (B,R,S) algorithm executed with GSM Schaetzle

Run	Iteration											Aggregates		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	16.0	4.7	5.2	4.9	5.0	5.6	5.1	5.1	5.1	5.1	5.9	67.7	5.17	0.33
2	16.0	4.5	5.1	4.9	5.0	5.9	5.0	5.0	5.8	5.1	5.3	67.6	5.16	0.4
3	16.4	1.6	1.8	1.7	1.9	2.2	1.9	1.9	1.9	1.9	6.0	39.2	2.28	1.25
4	16.0	4.5	5.1	4.9	5.2	5.5	5.1	5.0	5.8	5.2	5.4	67.7	5.17	0.33
5	16.4	4.6	4.9	4.8	6.2	5.7	5.0	5.0	4.9	5.0	5.1	67.6	5.12	0.45
μ	16.22	3.98	4.42	4.24	4.66	4.98	4.42	4.4	4.7	4.46	5.54	**61.96**		
σ	0.16	1.19	1.31	1.27	1.45	1.4	1.26	1.25	1.45	1.28	0.35	**11.38**		

(b) Schaetzle algorithm

Run	Iteration										Aggregates		
	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	8.3	8.7	8.7	9.2	8.7	8.7	8.7	8.7	8.7	8.9	87.3	8.73	0.21
2	8.1	8.3	8.3	8.8	8.3	8.3	8.3	8.3	8.3	8.3	83.3	8.33	0.17
3	7.9	8.2	8.2	8.2	8.1	8.2	8.1	8.6	8.1	8.2	81.8	8.18	0.17
4	7.9	8.3	8.3	8.4	8.4	8.7	8.8	8.4	8.4	8.5	84.1	8.41	0.23
5	7.9	8.2	8.1	8.2	8.3	8.3	8.3	8.3	8.3	8.3	82.2	8.22	0.12
μ	8.02	8.34	8.32	8.56	8.36	8.44	8.44	8.46	8.36	8.44	**83.74**		
σ	0.16	0.19	0.2	0.39	0.2	0.22	0.27	0.16	0.2	0.25	**1.96**		

(c) (B,R,S) algorithm executed with GSM Kaushik

Run	Iteration											Aggregates		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	16.1	1.9	2.5	3.8	7.0	7.0	7.5	7.2	7.4	7.2	7.3	74.9	5.88	2.11
2	16.1	2.1	2.7	4.2	7.0	8.3	7.5	8.0	7.6	8.0	11.8	83.3	6.72	2.77
3	16.6	2.4	2.8	4.3	7.9	7.8	8.0	7.8	7.8	7.8	10.9	84.1	6.75	2.55
4	16.1	1.9	2.6	3.9	7.1	7.3	7.7	7.2	7.3	7.1	7.1	75.3	5.92	2.1
5	16.6	5.2	5.9	7.4	11.2	11.1	11.2	10.7	10.6	10.8	11.3	112.0	9.54	2.28
μ	16.29	2.7	3.3	4.72	8.04	8.3	8.38	8.18	8.14	8.18	9.68	**85.92**		
σ	0.2	1.26	1.3	1.35	1.62	1.47	1.42	1.3	1.24	1.35	2.05	**13.6**		

Table 15: Detailed Results (minutes) on BTC2B for a 10-bisimulation.

(a) (B,R,S) algorithm executed with GSM Schaetzle

Run	Iteration											Aggregates		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	10.1	3.9	4.7	4.3	4.3	4.5	5.3	4.7	5.1	4.7	5.7	57.3	4.72	0.5
2	10.1	2.9	3.7	3.3	3.4	3.3	4.1	4.1	4.1	3.8	3.1	45.9	3.58	0.42
3	10.0	3.9	4.8	4.3	4.4	4.5	5.2	4.7	4.9	4.5	4.2	55.4	4.54	0.36
4	10.1	3.9	4.8	4.4	4.4	4.5	5.2	4.9	5.2	4.5	5.8	57.7	4.76	0.51
5	10.0	3.9	4.8	4.4	4.4	4.6	5.4	4.8	5.0	4.4	4.2	55.9	4.59	0.41
μ	10.2	3.7	4.56	4.14	4.18	4.28	5.04	4.64	4.86	4.38	4.6	**54.44**		
σ	0.62	0.4	0.43	0.42	0.39	0.49	0.48	0.28	0.39	0.31	1.02	**4.35**		

(b) Schaetzle algorithm

Run	Iteration										Aggregates		
	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	9.0	8.1	8.1	8.2	8.3	8.2	8.2	8.5	8.2	8.2	83.0	8.3	0.26
2	10.0	8.3	8.2	8.3	8.4	8.5	8.4	8.4	8.3	8.3	85.1	8.51	0.5
3	8.8	8.9	8.9	9.0	9.0	12.5	9.3	9.0	9.0	9.0	93.4	9.34	1.06
4	8.1	8.2	8.4	8.2	8.3	8.6	8.7	8.2	8.2	8.2	83.1	8.31	0.19
5	8.5	8.4	8.3	8.4	8.5	9.2	8.5	8.5	8.5	8.5	85.3	8.53	0.23
μ	8.88	8.38	8.38	8.42	8.5	9.4	8.62	8.52	8.44	8.44	**85.98**		
σ	0.64	0.28	0.28	0.3	0.26	1.58	0.38	0.26	0.3	0.3	**3.83**		

(c) (B,R,S) algorithm executed with GSM Kaushik

Run	Iteration											Aggregates		
	0	1	2	3	4	5	6	7	8	9	10	Σ	μ	σ
1	10.7	4.0	5.6	5.8	5.9	6.0	5.8	5.8	5.8	6.3	5.0	66.7	5.6	0.62
2	10.7	3.2	5.2	5.5	5.4	5.5	5.5	5.5	5.4	6.3	5.6	63.8	5.31	0.75
3	10.3	3.1	5.0	5.2	5.2	5.1	5.1	5.2	5.2	6.0	4.0	59.4	4.91	0.76
4	10.7	3.2	5.1	5.2	5.5	5.4	5.3	5.3	5.3	5.7	5.1	61.8	5.11	0.66
5	10.3	4.1	5.8	6.1	6.1	6.1	6.1	6.1	6.1	6.6	4.9	68.3	5.8	0.7
μ	10.43	3.52	5.34	5.56	5.62	5.62	5.56	5.58	5.56	6.18	4.92	**64.0**		
σ	0.27	0.44	0.31	0.35	0.33	0.38	0.36	0.33	0.34	0.31	0.52	**3.22**		

Table 16: Detailed Results (minutes) on BSBM1B for a 10-bisimulation.

TECHNICAL DOCUMENTATION

Following sections describe how to (1) setup the working environment, (2) run the (B,R,S) Algorithm , (3) run the Schätzle and Kaushik Algorithm, (4) run tests, (5) run experiments, (6) collect metrics (7) define own graph summary models, (8) add a bisimulation algorithm.

Setup Working Environment

- clone the git repository `git@gitlab.informatik.uni-ulm.de:dbis/data-science-and-big-data-analytics/teaching/2021ss-thesis-jannik.git`.
- install java oracle jdk 11 or later (tested with 11.05).
- install scala `sudo apt-get install scala`.
  - check version `scala -version`.
- install sbt.
  - `echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.d/sbt.list`.
  - `sudo apt-key adv –keyserver hkp://keyserver.ubuntu.com:80 –recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823`.
  - `sudo apt-get update`.
  - `sudo apt-get install sbt`.
- download and install spark.
  - download spark `wget. https://archive.apache.org/dist/spark/spark-3.1.1/spark-3.1.1-bin-hadoop3.2.tgz`.
  - create spark directory `mkdir /usr/local/spark`.
  - extract spark `tar xvf spark-3.1.1-bin-hadoop3.2.tgz -C /usr/local/spark/`.
  - you may need to change user permission so non-root users can read spark.
  - configure local environment for spark and scala. Add the following variables to. your bashrc, adapt if necessary (Max RAM size).
    - `export SCALA_HOME=/usr/bin/scala`.
    - `export SPARK_HOME=/usr/local/spark/spark-3.1.1-bin-hadoop3.2`.
    - `export SBT_OPTS="-Xmx1940G -Xms1940G"`.
  - update bashrc `source ~/.bashrc`.

Run (B,R,S) Algorithm

- To run the (B,R,S) Algorithm execute the following command.
  - `sbt "runMain MainBisim <path_to_config>"`.
- For all experiments with the (B,R,S) Algorithm, configurations are provided under `resources/configs/bisimulation/experiments/fluid/<dataset>/`.
- If you want to declare you own configuration, copy and edit. `resources/configs/template_fluid.conf`.
- **The template configuration contains explanations for each config entry**.

Run Schätzle and Kaushik Algorithm

- To run the Schaetzle and Kaushik Algorithm execute the following command.
  - `sbt "runMain bisimulation.Main <path_to_config>"`.
- For all experiments with the Schaetzle and Kaushik Algorithm, configurations are provided under `resources/configs/bisimulation/experiments/native/<dataset>`.
- If you want to declare your own configuration, copy and edit `resources/configs/template_native.conf`.
- **The template configuration contains explanations for each config entry**.

Run Tests

- To run all tests, execute the following command.
  - `sbt "test"`.
- To only run a specific test file, execute the following command.
  - `sbt "testOnly <package>.<test_file>"`.

Run Experiments

- To run an experiment, follow the steps below.
  - store the corresponding dataset on your machine.
  - create six corresponding config files, e.g., `btc_2b_brs_schaetzle_1.conf, btc_2b_brs_schaetzle_2.conf, ...`
    - alternatively, you can use the existing ones, however you may have to adjust some entries, like the location of the dataset or your log directory.
  - execute `sbt "runMain <main-method> <conf1>; sleep 60; sbt "runMain <main-method> <conf2>; sleep 60; ...; sbt "runMain <main-method> <conf6>`.
    - Here `<main-method>` is either
      - `MainBisim` in case of (B,R,S) or
      - `bisimulation.Main` in case of native Schätzle or Kaushik.

    * it is advised to wait for one minute between each run, otherwise nearly all data of the previous run will still be in RAM on start of execution.

Collect Metrics

- Make sure `<path_to_logDir>` only contains the log files for the experiments you want to collect metrics for!
- Collect run times for (B,R,S)
  - `python collectPerformanceFluid.py <path_to_logDir> <experiment_name> <k>`
  - The run times will be stored in `<path_to_logDir>/performance`
- Collect run times for native Schaetzle
  - `python collectPerformanceSchaetzle.py <path_to_logDir> <experiment_name> <k>`
  - The run times will be stored in `<path_to_logDir>/performance`
- Collect run times for native Kaushik
  - `./aggregateMetricsKaushik <path_to_logDir>`
  - `python collectPerformanceKaushik.py <path_to_logDir>/metrics <experiment_name> <k>`
  - The run times will be stored in `path_to_logDir>/metrics/performance`
  - Note: The first script is needed, because monitoring is done manually for Kaushik. This is because it is difficult to use Spark's Monitoring for algorithms that do not make use of the Spark API in execution.
- Collect Max Memory for any algorithm
  - `python collectMaxMemory.py <path_to_logDir> <experiment_name>`
  - Output is stored in `<path_to_logDir>/memory`

Define Own Graph Summary Model

- To define your own Graph Summary Model, create a file `SE_<name_of_GSM>.scala` in the `schema` package.
- In the file, create a scala object `SE_<name_of_GMS>` that extends the scala trait `SchemaExtractionBisim`.
- Now you have to define one variable, and implement four methods.
  - **variable** `edgeDirection`: In which direction messages shall be sent.
    * EdgeDirection.Out -> send messages to source (like in FW-Bisimulation).
    * EdgeDirection.In -> send messages to destination (like in BW-Bisimulation)
    * EdgeDirection.Either -> send. messages to source and destination (like in FW-BW-Bisimulation).
  - **method** `initialization`: Used to derive the initial id values for each vertex.
  - **method** `sendMessage`: Used to send messages in iterations 1 to k-1.
  - **method** `sendMessageFinal`: Used to send messages in iteration k.
  - **method** `initialiseNeighbours`: Used in the initialisation step to derive neighbours and properties of a vertex.
    * If the initialisation step does neither use neighbour nor property information, return an empty Array.
    * If it uses only neighbour information, return Array(0, vertexIdOfNeighbour).
    * If it uses only property information return Array(property, 0).
    * 0 is just an arbitrary value here.
- It is recommended to test your Graph Summary Model properly.

Add a Bisimulation Algorithm

- To add a bisimulation algorithm, create a file `name_of_algorithm.scala` in the `bisimulation` package.
- Implement your algorithm.
  - The provided parser outputs a graph with vertex type Set[(Int, Int)] and edge type (Int, Int)
  - A tuple in the vertex's set attribute contains the information (hashOfLabel, hashOfSource)
  - The edge attribute is a tuple that contains the information (hashOfEdgeLabel, hashOfSource)
  - You may have to change this for example by mapping over the vertices or edges, such that the result is of your necessary type
- Add an `else if(algName == "nameOfYourAlgorithm"){...}` block to the `bisimulation.Executor.scala` file.
  - The Executor.scala file executes the algorithm that is specified in the config file, hence the need for this block.
- It is recommended to test your bisimulation algorithm properly.