DIMSpan - Transactional Frequent Subgraph Mining with Distributed In-Memory Dataflow Systems

André Petermann, Martin Junghanns and Erhard Rahm
University of Leipzig & ScaDS Dresden/Leipzig
[petermann,junghanns,rahm]@informatik.uni-leipzig.de

ABSTRACT

Transactional frequent subgraph mining identifies frequent structural patterns in a collection of graphs. This research problem has wide applicability and increasingly requires higher scalability over single machine solutions to address the needs of Big Data use cases. We introduce DIMSpan, an advanced approach to frequent subgraph mining that utilizes the features provided by distributed in-memory dataflow systems such as Apache Flink or Apache Spark. It determines the complete set of frequent subgraphs from arbitrary string-labeled directed multigraphs as they occur in social, business and knowledge networks. DIMSpan is optimized to runtime and minimal network traffic but memory-aware. An extensive performance evaluation on large graph collections shows the scalability of DIMSpan and the effectiveness of its optimization techniques.

CCS CONCEPTS

•Information systems → Data mining; •Computing methodologies → Distributed algorithms;

KEYWORDS

Distributed Frequent Subgraph Mining; Shared Nothing Cluster

1 INTRODUCTION

Mining frequent structural patterns from a collection of graphs, usually referred to as *frequent subgraph mining* (FSM), has found much research interest in the last two decades, for example, to identify significant patterns from chemical or biological structures and protein interaction networks [13]. Besides these typical application domains, graph collections are generally a natural representation of partitioned network data such as knowledge graphs [7], business process executions [25] or communities in a social network [14]. We identified two requirements for FSM on such data that are not satisfied by existing approaches: First, such data typically describes directed multigraphs, i.e., the direction of an edge has a semantic meaning and there may exist multiple edges between the same pair of vertices. Second, single machine solutions will not be sufficient for big data scenarios where either input data volume as well as size of intermediate results can exceed main memory or achievable runtimes are not satisfying.

An established approach to speed up or even enable complex computations on very large data volumes is data-centric processing on clusters without shared memory. The rise of this approach was strongly connected with the MapReduce [8] programming paradigm, which has also been applied to the FSM problem [2, 3, 10, 18, 19]. However, none of the approaches provides support for directed multigraphs. Further on, MapReduce is not well suited for complex iterative problems like FSM as it leads to a massive overhead of disk access.

In recent years, a new generation of advanced cluster computing systems like Apache Flink [6] and Apache Spark [35], in the following denoted by *distributed in-memory dataflow systems*, appeared. In contrast to MapReduce, these systems provide a larger set of operators and support holding data in main memory between operators as well as during iterative calculations.

In this work, we propose DIMSpan, an advanced approach to distributed FSM based on this kind of system. Our contributions can be summarized as follows:

- We propose DIMSpan, the first approach to parallel FSM with distributed in-memory dataflow systems (Section 3). It adapts all pruning features of the popular gSpan [32] algorithm to the dataflow programming model and supports directed multigraphs.
- We provide a comparison to existing MapReduce based approaches (Section 4) and show that DIMSpan not only requires fewer disk access but also shuffles less data over the network and can reduce the total number of expensive isomorphism resolutions to a minimum.
- We present results of experimental evaluations (Section 5) based on real and synthetic datasets to show the scalability of our approach as well as the runtime impact of our optimization techniques .
- Our implementation is practicable and works for arbitrary string-labeled graphs. We provide its source code to the community as part of the Gradoop framework [24] under an Open Source licence.

In addition, we provide background knowledge and discuss related work in Section 2. Finally, we conclude in Section 6.

2 BACKGROUND & RELATED WORK

In this section, we introduce the distributed dataflow programming model, define the frequent subgraph mining problem and discuss related work.

BDCAT'17, December 5–8, 2017, Austin, Texas, USA

ACM ISBN 978-1-4503-5549-0/17/12. . . $15.00
https://doi.org/10.1145/3148055.3148064

Table 1: Glossary of symbols

$G/v/e/P/m$	graph / vertex / edge / pattern / embedding
$\mathcal{G}/V/E/\mathcal{P}/M$	sets of $G/v/e/P/m$
$\mathcal{F}/\mu$	set of frequent patterns / pattern-embeddings map
ϕ/ϕ_w	pattern frequency / frequency within a partition
$?^k/\mathcal{G}_{i\in\mathbb{N}}$	k-edge variant of ? / partition of a graph set
$C_{min}(P)$	minimum DFS code of a pattern
$C^1(e)$	minimum DFS code of an edge
$C^1(P)$	first extension of a pattern's min. DFS code

2.1 Distributed Dataflow Model

Distributed dataflow systems like MapReduce [8], Apache Flink [6] or Apache Spark [35] are designed to implement data-centric algorithms on shared nothing clusters without handling the technical aspects of parallelization. The fundamental programming abstractions are datasets and transformations among them. A *dataset* is a set of data objects partitioned over a cluster of computers. A *transformation* is an operation that is executed on the elements of one or two input datasets. The output of a transformation is a new dataset. Transformations can be executed concurrently on $W = \{w_0, w_1, .., w_n\}$ available *worker threads*, where every thread executes the transformation on an associated dataset partition. There is no shared memory among threads.

Depending on the number of input datasets we distinguish *unary* and *binary* transformations. Table 2 shows example unary tranformations. We further divide them into those transformations processing *single elements* and those processing *groups of elements*. All of the shown functions require the user to provide a *transformation function* τ that needs to be executed for each element or group. A simple transformation is *filter*, were τ is a predicate function and only those elements for which τ evaluates to true will be added to the output. Another simple transformation is *map*, where τ describes how exactly one output element is derived from an input element. *Flatmap* is similar to map but allows an arbitrary number of output elements. MapReduce provides only one single-element transformation (denoted by *MRMap* in Table 2) which is a variant of flatmap that requires input and output elements to be key-value pairs.

The most important element group transformation is *reduce*. Here, input as well as output are key-value pairs and for each execution all elements sharing the same key are aggregated and τ describes the generation of a single output pair with the same key. Since input pairs with the same key may be located in different partitions they need to be *shuffled* among threads which is typically causing network traffic among physical machines. If τ is associative (e.g. summation), an additional combine transformation can be used to reduce this traffic. *Combine* is equivalent to reduce but skips shuffling, i.e., in the worst case one output pair is generated for each key and thread. Afterwards, these partial aggregation results can be passed to a reduce transformation.

As map and filter can also be expressed using MRMap, MapReduce and the new generation of *distributed in-memory dataflow systems* (DIMS) like Flink and Spark have the same expressive power in terms of unary transformations. However, in the case of successive or iterative MRMap-reduce phases intermediate results need to be read from disk at the beginning and written to disk at the end of each phase. Thus, MapReduce is not well suited to solve iterative problems and problem-specific distributed computing models arose, for example, to process very large graphs [20]. In contrast, MapReduce and DIMS are general purpose platforms and not dedicated to a specific problem. However, DIMS support more complex programs including iterations, binary transformations (e.g., set operators like *union* and *join*) and are able to hold datasets in main memory during the whole program execution.

Table 2: Selected Unary Tranformations

Transf.	Signature	Constraints
single element transformations		
Filter	$I, O \subseteq A$	$O \subseteq I$
Map	$I \subseteq A, O \subseteq B$	$\lvert I\rvert = \lvert O\rvert$
Flatmap	$I \subseteq A, O \subseteq B$	-
MRMap	$I \subseteq A \times B; O \subseteq C \times D$	-
element group transformations		
Reduce	$I, O \subseteq A \times B$	$\lvert I\rvert \geq \lvert O\rvert \wedge \lvert O\rvert \leq \lvert A\rvert$
Combine	$I, O \subseteq A \times B$	$\lvert I\rvert \geq \lvert O\rvert \wedge \lvert O\rvert \leq \lvert A \times W\rvert$

(I/O : input/output datasets, A..D : domains, W : worker threads)

2.2 Frequent Subgraph Mining

Frequent subgraph mining (FSM) is a variant of frequent pattern mining [1] where patterns are graphs. There are two variants of the FSM problem. *Single graph FSM* identifies patterns occurring at least a given number of times within a single graph, while *graph transaction FSM* searches for patterns occurring in a minimum number of graphs in a collection. Our proposed approach belongs to the second setting. Since there exist many variations of this problem we first define our problem precisely before discussing related work and introducing our algorithm.

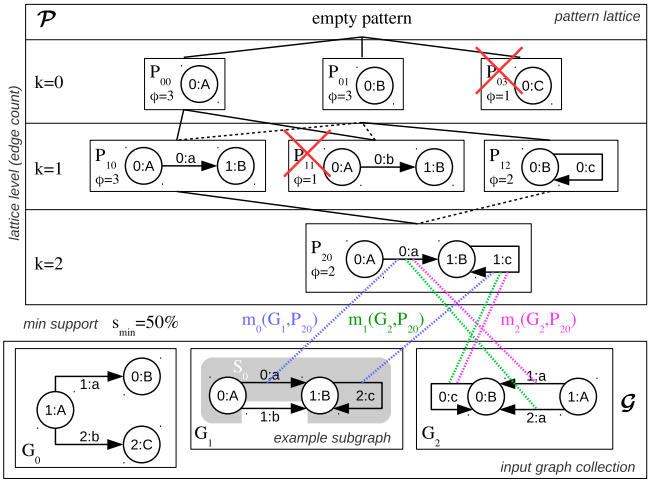
Definition 2.1. (Graph). Given two global label sets $\mathcal{L}_v, \mathcal{L}_e$, then a *directed labeled multigraph*, in the following simply referred to as *graph*, is defined to be a hextuple $G = \langle V, E, s, t, \lambda_v, \lambda_e \rangle$, where $V = \{v\}$ is the set of vertices (vertex identifiers), $E = \{e\}$ is the set of edges (edge identifiers), the functions $s : E \to V$ / $t : E \to V$ map a *source* and a *target* vertex to a every edge and $\lambda_v : V \to \mathcal{L}_v$ / $\lambda_e : E \to \mathcal{L}_e$ associate labels to vertices and edges. An edge $e \in E$ is *directed* from $s(e)$ to $t(e)$. A multigraph supports loops and parallel edges.

Definition 2.2. (Subgraph). Let S, G be graphs then S will be considered to be a *subgraph* of G, in the following denoted by $S \sqsubseteq G$, if S has subsets of vertices $S.V \subseteq G.V$ and edges $S.E \subseteq G.E$ and $\forall e \in S.E : s(e), t(e) \in S.V$ is true.

On the bottom of Figure 1, a collection of directed multigraphs $\mathcal{G} = \{G_1, G_2, G_3\}$ and an example subgraph $S_0 \sqsubseteq G_1$ are illustrated. Identifiers and labels of vertices and edged are encoded in the format `id:label`, e.g., `1:A`.

Definition 2.3. (Isomorphism). Two graphs G, H will be considered to be isomorphic ($G \simeq H$) if two bijective mappings exist for vertices $\iota_v : G.V \leftrightarrow H.V$ and edges $\iota_e : G.E \leftrightarrow H.E$ with matching labels, sources and targets, i.e., $\forall v \in G.V : G.\lambda_v(v) = H.\lambda_v(\iota_v(v))$

Figure 1: Example illustrations for a graph collection, a subgraph, a pattern lattice and embeddings.



and $\forall e \in G.E : G.\lambda_e(e) = H.\lambda_e(\iota_e(e)) \wedge G.s(e) = H.s(\iota_e(e)) \wedge G.t(e) = H.t(\iota_e(e))$.

Definition 2.4. (Pattern Lattice). A *pattern* is a connected graph isomorphic to a subgraph $P \simeq S$. Let $\mathcal{P} = \{P^{-1}, P_0, .., P_n\}$ be the set of all patterns isomorphic to any subgraph in a graph collection, than patterns form a *lattice* based on parent-child relationships. P_p will be a parent of P_c if $P_p \sqsubset P_c \wedge |P_p.E| = |P_c.E| - 1$. Based on edge count k there are disjoint *levels* $\mathcal{P}^{-1}, .., \mathcal{P}^k \subseteq \mathcal{P}$. Root level $\mathcal{P}^{-1} = \{P^{-1}\}$ contains only the empty pattern P^{-1} which is the parent of all patterns with $k = 0$. For all other patterns $\forall P^k \in \mathcal{P}, k > 0 \; \exists \, P^{k-1} \in \mathcal{P} : P^{k-1} \sqsubset P^k$ is true.

Figure 1 shows the lattice of patterns $\mathcal{P} = \{P_{00}, .., P_{20}\}$ occurring in the example graph collection $\mathcal{G}$.

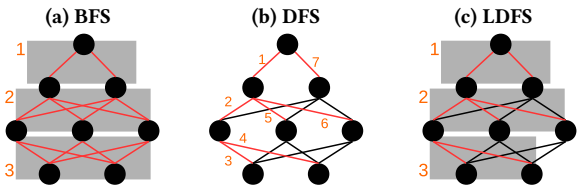
Definition 2.5. (Embedding). Let G be a graph and P be a pattern, then an *embedding* is defined to be a pair $m(G,P) = \langle \iota_v, \iota_e \rangle$ of isomorphism mappings describing a subgraph $S \sqsubseteq G$ isomorphic to P. As a graph may contain n subgraphs isomorphic to the same pattern (e.g., subgraph automorphisms), we use $\mu : \mathcal{P} \rightarrow M^n$ to denote an *embedding map*, which assoicates n elements of an embeddings set M to every pattern $P \in \mathcal{P}$. If μ maps to an empty tuple, the graph will not contain a pattern.

Figure 1 shows three differently colored edge mappings of example embeddings $m_0(G_1, P_{20}), m_1(G_2, P_{20})$ and $m_2(G_2, P_{20})$.

Definition 2.6. (Frequency/Support). Let $\mathcal{G} = \{G_0, .., G_n\}$ be a graph collection and P be a pattern, then the *frequency* $\phi : \mathcal{P} \rightarrow \mathbb{N}$ of a pattern is the number of graphs containing at least one subgraph isomorphic to the pattern. The term *support* describes the frequency of a pattern relative to the number of graphs $\sigma(P) = \phi(P)/|\mathcal{G}|$.

Definition 2.7. (Frequent Subgraph Mining). Let $\mathcal{G}$ be a graph collection, $\mathcal{P}$ the set of all contained patterns and s_{min} be the minimum support with $0 \leq s_{min} \leq 1$, then the problem of *frequent subgraph mining* is to identify the complete set of patterns $\mathcal{F} \subseteq \mathcal{P}$ where $\forall P \in \mathcal{P} : P \in \mathcal{F} \Leftrightarrow \sigma(P) \geq s_{min}$ is true.

Figure 2: Pattern lattice search strategies.



Using the example graph collection $\mathcal{G} = \{G_1, G_2, G_3\}$ of Figure 1, frequent subgraph mining with $s_{min} = 50\%/f_{min} = 2$ results in the five not-crossed patterns with $\phi(P) \geq 2$.

2.3 Related Work

A recent survey [13] by Jiang et al. provides an extensive overview about frequent subgraph mining (FSM). Due to limited space and the vast amount of work related to this problem we only discuss approaches matching Definition 2.7. Thus, we omit the single-graph setting [5, 9, 28] as well as graph-transaction approaches with incomplete results like maximal [29], closed [34] or significant [26] frequent subgraph mining.

The first exact FSM algorithms, e.g., AGM [12] and FSG [16], followed an *a priori* approach. These algorithms implement a level-wise breath-first-search (BFS, illustrated by Figure 2a) in the pattern lattice, i.e., candidate patterns $\mathcal{P}^k$ are generated and the support is calculated by subgraph isomorphism testing. In a subsequent pruning step frequent patterns $\mathcal{F}^k \subseteq \mathcal{P}^k$ are filtered and joined to form children $\mathcal{P}^{k+1}$ (next round's candidates). The search is stopped as soon as $\mathcal{F}^k = \emptyset$. The disadvantage of these algorithms is that they are facing the subgraph isomorphism problem during candidate generation and support counting. Further on, it is possible that many generated candidates might not even appear.

Thus, the next generation of *pattern-growth* based FSM algorithms appeared and outperformed the a priori ones. Popular representatives of this category are MOFA [4], gSpan [32], FFSM [11] and Gaston [21]. In comparison to the a priori ones, these algorithms traverse the lattice in a depth-first search (DFS, illustrated by Figure 2b) and skip certain links in the lattice (dotted lines in Figure 1) to avoid visiting child patterns multiple times. A key concept of these algorithms are canonical labels generated during DFS. However, if labels are generated without recalculation (e.g., gSpan) they won't totally prevent false positives (non canonical labels) and thus an additional isomorphism-based verification will be required. Comparative work [22, 31] has shown that runtime can be decreased by fast label generation and holding embeddings in main memory.

While most popular exact FSM algorithms are from the first half of the 2000s, more recent work focuses on problem variations [13] as well as parallelization, for example, using GPUs [15], FPGAs [27] and multithreading [30]. All existing approaches of graph transaction FSM on shared nothing clusters [2, 3, 10, 18, 19] are based on MapReduce [8] and will be further discussed in comparison to DIMSpan in Section 4. Graph transaction FSM cannot benefit from vertex-centric graph processing approaches [20] as partitioning a single graph shows different problems than partitioning a graph collection.

Algorithm 1 Distributed FSM dataflow.

Require: $\mathcal{G} = \{\langle G, \mu^1 \rangle_i\}_{i \subset \mathbb{N}}, f_{min}$
1: $\mathcal{F} \leftarrow \emptyset$
2: $\mathcal{F}^k \leftarrow \emptyset$
3: **repeat**
4: $\mathcal{P}^k \leftarrow \mathcal{G}$.**flatmap**(*report*)
5: $\phi_w^k \leftarrow \mathcal{P}^k$.**combine**(*count*)
6: $\phi^k \leftarrow \phi_w^k$.**reduce**(*sum*)
7: $\mathcal{F}^k = \mathcal{P}^k$.**filter**($\phi^k(P) \geq f_{min}$)
8: **broadcast**($\mathcal{F}^k$)
9: $\mathcal{G} \leftarrow \mathcal{G}$.**map**(*patternGrowth*)
10: $\mathcal{G} \leftarrow \mathcal{G}$.**filter**($\exists\, P : |\mu^{k+1}(G,P)| > 0 \wedge C(P) = C_{min}(P)$)
11: $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}^k$
12: **until** $\mathcal{F}^k \neq \emptyset$
13: **return** $\mathcal{F}$

Algorithm 2 Pattern growth map function τ.

Require: $G, \mu^k, \mathcal{F}^k = \langle P_0, .., P_n \mid \text{sorted by } C_{min} \rangle$
1: $C^1_{min} \leftarrow \langle\rangle$ // minimum branch
2: $E_{\geq min} \leftarrow G.E$ // shrinking branch-validated edge set
3: **for** $P^k \in \mathcal{F}^k \mid \mu^k(G, P^k) \neq \langle\rangle$ **do**
4: **if** $C^1(P^k) > C^1_{min}$ **then**
5: $C^1_{min} \leftarrow C^1(P^k)$ // update min branch and edge set
6: $E_{\geq min} \leftarrow\subset E_{\geq min} \mid C^1(e) \geq C^1_{min}$
7: **end if**
8: **for** $m^k, e \in (\mu^k(G, P^k) \times E_{\geq min})$ **do**
9: **if** $\nexists\, m^k.\iota_e(e)$ **and** time constraint satisfied **then**
10: grow P^{k+1}, m^{k+1} and add to μ^{k+1}
11: **end if**
12: **end for**
13: **end for**
14: **return** G, μ^{k+1}

3 ALGORITHM

In the following, we provide details about the DIMSpan algorithm including its concept (3.1), the respective dataflow program (3.2) as well as pruning and optimization techniques (3.4 - 3.7).

3.1 Concept

The input graph collection $\mathcal{G}$ is represented by a dataset of graphs equally partitioned into disjoint subsets $\mathcal{G}_1, \mathcal{G}_2, .., \mathcal{G}_n$ corresponding to the availble *worker threads* $W = \{w_1, w_2, .., w_n\}$. Thus, transformations can be executed on $|W|$ graphs in parallel but every exchange of global knowledge (e.g., local pattern frequencies) requires synchronization barriers in the dataflow program which cause network traffic. Our major optimization criteria were minimizing delays dependent on exchanged data volume and, as FSM contains the NP-complete subgraph isomorphism problem, minimize the number of isomorphism resolutions.

To achieve the latter, we adapted approaches of two efficient pattern-growth algorithms gSpan [32] and Gaston [21]. These algorithms basically are iterations of pattern growth, counting and filter operations but differ in detail. gSpan allows fast append-only generation of canonical labels representing patterns but records only pattern-graph occurrence lists. This requires subgraph isomorphism testing to recover embeddings. In contrast, Gaston has a more complex label generation tailored to the characteristics of molecular databases but stores complete pattern-embedding maps. For the design of DIMSpan, we combine the strong parts of both algorithms. In particular, we use a derivate of gSpan canonical labels (Section 3.3) but also store embedding maps to avoid subgraph isomorphism testing at the recovery of previous iterations' embeddings. To minimize the additional memory usage, we use optimized data structures and compression (Section 3.7).

As in absence of shared memory every iteration is causing a synchronization barrier, the DFS search of pattern growth algorithms is less suited as it requires $|\mathcal{P}|$ iterations (one for each visited pattern) while a BFS search only takes k^{max} iterations (maximum edge count). Thus, we decided to perform a *level-wise depth-first search* (LDFS, illustrated by Figure 2c), which can be abstracted as a set of massive parallel constrained DFSs with level-wise forking. This approach allows us to benefit from the efficiency of pattern growth algorithms and to apply level-wise frequency pruning at the same time. For example, in Figure 1 we apply the frequency pruning of P_{10}, P_{11}, P_{12} in parallel within the same iteration but use search constraints (Section 3.4) to grow only from P_{10} to P_{20}.

By using a distributed in-memory dataflow system instead of MapReduce, DIMSpan further benefits from the capability to hold graphs including supported patterns and their embeddings in main memory between iterations and to exchange global knowledge by sending complete copies of each iteration's k-edge frequent patterns to every worker without disk access. In Apache Flink and Apache Spark this technique is called broadcasting[1][2].

3.2 Distributed Dataflow

Algorithm 1 shows the distributed dataflow of DIMSpan. Inputs are a dataset of graphs $\mathcal{G}$ and the minimum frequency threshold f_{min}. The output is the dataset of frequent patterns $\mathcal{F}$. For each graph, supported 1-edge patterns $\mathcal{P}^1$ and the embedding map μ^1 are already computed in a preprocessing step (see Section 3.6). Our algorithm is iterative and per iteration one level of the pattern lattice is processed until no more frequent patterns exist (line 12). In the following, we describe transformations and intermediate datasets of the iteration body (lines 4 to 11) in more detail:

Line 4 - Report: In the beginning of each iteration every graph reports all k-edge ($k \geq 1$) supported patterns, i.e., the keys of the last iteration's embedding map μ^k, through a *flatmap* transformation.

Line 5 - Combine: The partition frequency of patterns $\phi_w : \mathcal{P} \times W \rightarrow \mathbb{N}$ is counted in a *combine* tranformation.

Line 6 - Reduce: The global frequency of patterns $\phi : \mathcal{P} \rightarrow \mathbb{N}$ is calculated in a *reduce* transformation. Here, partition frequencies are shuffled among workers and summed up.

Line 7 - Frequency pruning and verification: After global frequencies of all patterns are known, a *filter* transformation is used to determine the frequent ones. Additionally, every remaining pattern is verified to be no false positive (see Section 3.5). This is the step we resolve subgraph isomorphism with cardinality $|\mathcal{F}|$.

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html#broadcast-variables
[2] http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables

Line 8 - Broadcasting: After $\mathcal{F}^k$ is known, a complete copy is sent to the main memory of all workers using *broadcasting*.

Line 9 - Pattern growth: Here, the previously broadcasted set $\mathcal{F}^k$ is used to filter each graph's embeddings μ^k to those of frequent patterns. For each of the remaining embeddings, the constrained pattern growth (Section 3.4) is performed to generate μ^{k+1}.

Line 10 - Obsolescence filter: After pattern growth, we apply another *filter* operation and only graphs with non-empty μ^{k+1} will pass. Thus, $\mathcal{G}$ can potentially shrink in each iteration if only a subset of graphs accumulates frequent patterns.

Line 11 - Result storage: Finally, we use a binary *union* transformation to add the iteration's results to the final result set.

3.3 Canonical Labels for Directed Multigraphs

We use a derivate of the gSpan minimum DFS code [32] as canonical labels for directed multigraph patterns:

Definition 3.1. (DFS Code). A *DSF code* representing a pattern of j vertices and k edges ($j, k \geq 1$) is defined to be a k-tuple $C = \langle x_1, x_2, .., x_k \rangle$ of extensions, where each *extension* is a hextuple $x = \langle t_a, t_b, l_a, d, l_e, l_b \rangle$ representing the traversal of an edge e with label $l_e \in \mathcal{L}_e$ from a *start* vertex v_a to an *end* vertex v_b. $d \in \{in, out\}$ indicates if the edge was traversed in or against its direction. A traversal will be considered to be in direction, if the start vertex is the source vertex, i.e., $v_a(x) = s(e)$. The fields $l_a, l_b \in \mathcal{L}_v$ represent the respective labels of both vertices and their initial discovery times $t_a, t_b \in T \mid T = \langle 0, .., j \rangle$ where the vertex at $t = 0$ is always the start vertex of the first extension. A DFS code C_p will be considered to be the parent of a DFS code C_c, iff $\forall i \in \langle 1, .., k-1 \rangle : C_c.x_i = C_p.x_i$.

According to this definition child DFS codes can be easily generated by adding a single traversal to their parent. Further on, DFS codes support multigraphs since extension indexes can be mapped to edges identifiers to describe embeddings.

However, there may exist multiple DFS codes representing the same graph pattern. To use DFS codes as a canonical form, gSpan is using a lexicographic order to determine a minimum one among all possible DFS codes [33]. This order is a combination of two linear orders. The first is defined on start and end vertex times of extensions $T \times T$, for example, a backwards growth to an already discovered vertex is smaller than a forwards growth to a new one. The second order is defined on the labels of start vertex, edge and end vertex $\mathcal{L}_v \times \mathcal{L}_e \times \mathcal{L}_v$, i.e., if a comparison cannot be made based on vertex discovery times, labels and their natural order (e.g., alphabetical) are compared from left to right. To support directed graphs, we extended this order by direction $D = \{in, out\}$ with $out < in$ resulting into an order over $\mathcal{L}_v \times D \times \mathcal{L}_e \times \mathcal{L}_v$, i.e., in the case of two traversals with same start vertex labels, a traversal of an outgoing edge will always be considered to be smaller.

Definition 3.2. (Minimum DFS Code). There exists an order among DFS codes such that $\forall C_1, C_2 : C_1 < C_2 \vee C_1 = C_2 \vee C_1 > C_2$ is true. Let C_P be the set of all DFS codes describing a pattern P and C_{min} be its minimum DFS code, than $\nexists\, C_i \in C_P : C_i < C_{min}$ is true.

3.4 Constrained Pattern Growth

Besides gSpan's canonical labels we also adapted the growth constraints to skip parent-child relationships in the pattern lattice (dotted lines in Figure 1). However, in contrast to gSpan, we don't perform a pattern-centric DFS (Figure 2b) but an level-wise DFS (Figure 2c), i.e., we perform highly concurrent embedding-centric searches. Due to limited space, we refer to [33] for the theoretical background and focus on our adaptation to the distributed dataflow programming model.

There are two constraints for growing children of a parent embedding. The first, in the following denoted by *time constraint*, dictates that forwards growth is only allowed starting from the rightmost path and backwards growth only from the rightmost vertex, where *forwards* means an extension to a vertex not contained in the parent, *backwards* means an extension to a contained one, the *rightmost vertex* is the parent's latest discovered vertex and the *rightmost path* is the path of forward growths from the initial start vertex to the rightmost one. The second constrained, in the following denoted by *branch constraint*, commands that the minimum DFS code of an edge $C^1(e)$ needs to be greater than or equal to the parent's *branch* $C^1(P)$ which is the 1-edge code described by only the initial extension of the a pattern's minimum DFS code.

Algorithm 2 shows our adaption of these constraints to the distributed dataflow programming model, in particular, a map function τ that executes pattern growth for all embeddings of frequent patters in a single graph (line 9 of Algorithm 1). Therefore, we hold not only G but also embedding map μ^k for each element of $\mathcal{G}$ and enable τ access to $\mathcal{F}^k$ as it was received by every worker in the broadcasting step (line 8 of Algorithm 1).

In an embedding-centric approach, a naive solution would be testing possible growth for the cross of supported frequent patterns' embeddings and the graph's edges. As an optimization, we use a merge strategy based on the branch constraint to reduce the number of these tests. Therefore, $\mathcal{F}^k$ in Algorithm 2 is an n-tuple and ascendantly sorted by minimum DFS code. When executing the map function, we keep a current minimum branch C^1_{min} and a current edge candidate set $E_{\geq min}$ (lines 1,2). Then, for every supported frequent pattern (line 3) we compare its branch to the current minimum (line 4) and only if it is greater, the current minimum will be updated (line 5) and the set of growth candidates can be shrunk (line 6). Thus, only for the cross of embeddings and branch-validated edges (line 8) parent containment and the time constraint need to be checked (line 9). In the case of a successful growth (line 10) the resulting pattern and its embedding will be added to μ^{k+1}, the output of the map function (line 14). Sorting and rightmost path calculation are not part of the map function and executed only $|W \times \mathcal{F}|$ times at broadcast reception.

3.5 False Positive Verification

Although the constrained pattern growth described previously helps skipping links in the pattern lattice (dotted lines in Figure 1), it gives no guarantee for visiting every pattern only once. In the case of multiple (n) visits, $n-1$ non-minimal DFS codes (*false positives*) will be generated. Thus, they need to be verified. Therefore, we turn the label into a graph and recalculate the minimum DFS code. Since this is the only part of the algorithm resolving the isomorphism problem, reducing its cardinality may reduce total runtime [33]. Thus, we placed the verification step after frequency pruning (line 7 of Algorithm 1). Hence, false positives are counted and shuffled but verification is only executed $|\mathcal{F}|$ times.

3.6 Preprocessing and Dictionary Coding

Before executing the dataflow shown by Algorithm 1, we apply preprocessing that includes label-frequency based pruning, string-integer dictionary coding and sorting edges according to their 1-edge minimum DFS codes. The original gSpan algorithm already used these concepts but we improved the first two and adapted the third to our level-wise DFS strategy. In the first preprocessing step, we determine frequent vertex labels and broadcast a dictionary to all workers. Afterwards, we drop all vertices with infrequent labels as well as their incident edges. Then, we determine frequent edge labels, in contrast to the original, only based on the remaining edges. Thus, we can potentially drop more edges, for example, e_1 of G_1 in Figure 1 would be removed. This would not be the case by just evaluating its edge label since without dropping e_2 of G_0 before (because v_2 has infrequent label C) the frequency of edge label b would be 2, i.e., considered to be frequent.

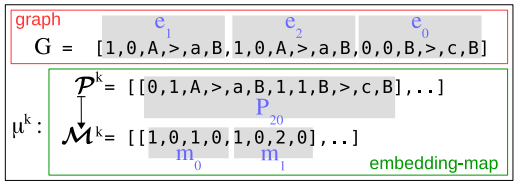
After dictionaries for vertex and edge labels are made available to all workers by broadcasting, we not only replace string labels by integers to save memory and to accelerate comparison but also sort edges according to their minimum DFS code, i.e., we use n-tuples instead of sets to store edges. We benefit from the resulting sortedness in every execution of the constrained pattern growth (see Section 3.4) as the effort of determining branch-valid edge candidates (line 6 of Algorithm 2) is reduced from a set filter operation to a simple increase of the minimum edge index.

3.7 Data Structures and Compression

We not only use minimum DFS codes as canonical labels but also a data structure based thereon to support all pattern operations (counting, growth and verification) without format conversions. We further store graphs as sorted lists of 1-edge DFS codes to allow a direct comparison at the lookup for the first valid edge of a branch in the pattern growth process (line 6 of Algorithm 2). Figure 3 illustrates a single element of $\mathcal{G}$ in Algorithm 1 representing G_2 from Figure 1 and its embedding map μ^k in the $k = 2$ iteration. Graphs and patterns are stored according to Definition 3.1 but encoded in integer arrays where all 6 elements store a graph's edge or a pattern's extension. For the sake of readability we use alphanumerical characters in Figure 3. μ^k is stored as a pair of nested integer arrays $\langle \mathcal{P}^k, \mathcal{M}^k \rangle$ where equal indexes map embeddings to patterns. All embeddings of the same pattern are encoded in a single multiplexed integer array where all $|P.V| + |P.E|$ elements store a single embedding. Here, indexes relative to their offset relate vertex ids to their initial discovery time and edge ids to extension numbers.

This data structure not only allows fast pattern operations but also enables lightweight and effective integer compression. Therefore, we exploit the predictable value ranges of our integer arrays. As we use dictionary coding and vertex discovery times are bound by the maximum edge count k_{max} the array's values may only range from $0..(max(k_{max}, l_v, l_e) - 1)$ where l_v, l_e are the numbers of distinct vertex and edge labels. In the context of FSM, the maximum value will typically be much less than the integer range of 2^{32}. There are compression techniques benefiting from low-valued integer arrays [17]. In preliminary experiments we found that Simple16 [36] allows very fast compression and gives an average compression ratio of about 7 over all patterns found in our synthetic test dataset (see Section 5.2). We apply integer compression not only

Figure 3: Dataset element representing graph G_2, pattern P_{20} and embedding set $M(G_2, P_{20})$ of Figure 1.



to patterns but also to graphs and embeddings, which also have low maximum values, to decrease memory usage. Embeddings and graphs are only decompressed on demand and at maximum for one graph at the same time. All equality-based operations (map access and frequency counting) are performed on compressed values. Our experimental evaluation results show a significant impact of this compression strategy (see Section 5).

4 COMPARISON TO APPROACHES BASED ON MAPREDUCE

To the best of our knowledge, only five approaches to transactional FSM based on shared nothing clusters exist [2, 3, 10, 18, 19]. They are all based on MapReduce. Since [2, 3] show relaxed problem definitions in comparison to Definition 2.7, we compare DIMSpan only to I-FSM [10], MR-FSE [19] and the filter-refinement (F&R) approach of [18]. The authors of MR-FSE and F&R have shown to be faster than I-FSM in experimental evaluations. Initially, we wanted to reproduce evaluation results of MR-FSE and F&R on our own cluster. Unfortunately, MR-FSE is not available to the public. Regarding F&R, only binaries[3] are accessible. However, there is no sufficient English documentation and the binaries rely on an outdated non-standard Hadoop installation. Thus, we were not able to execute the binaries without errors despite notable support of the author. For this reason, we qualitatively compare the main execution costs of the MapReduce approaches with DIMSpan w.r.t volume of disk access and data exchange (shuffling) and the number of isomorphism resolutions.

4.1 Methodical Comparison

Table 3 compares the considered methods w.r.t. the steps of preprocessing, two map-reduce phases and postprocessing. All approaches except one are iterative, i.e., perform a level-wise search. For these iterative methods, the map-reduce phases of Table 3 represent a single iteration's body. In contrast, F&R is partition-based and requires only two map-reduce phases to extract frequent patterns of all sizes. In the following, we briefly describe the MapReduce approaches with regard to Table 3:

I-FSM is using full subgraphs (structure and labels) as its main data structure. In map phase 1 (Map 1) k-edge subgraphs of the previous iteration are read from disk. In reduce phase 1 (Reduce 1), subgraphs are shuffled by graph id and graphs are reconstructed by a union of all subgraphs. Afterwards, $k + 1$-edge subgraphs are generated and written to disk. In Map 2 they are read again and a canonical label is calculated for every subgraph. In Reduce 2, all subgraphs are shuffled again according to the added label and label frequencies are counted. Finally, all subgraphs showing a frequent label are written to disk.

[3] https://sourceforge.net/projects/mrfsm/

Table 3: Methodical comparison of DIMSpan and approaches based on MapReduce.

	Pre.	Map 1	Reduce 1	Map 2	Reduce 2	Post.
I-FSM [10] (iterative)		read subgraphs	shuffle subgraphs, pattern growth, write subgraphs	read subgraphs, **add canonical label**	shuffle subgraphs, find frequent labels, filter subgraphs by label, write subgraphs	
MR-FSE [19] (iterative)		read pattern-embeddings map, read frequent patterns, **pattern growth**, write pattern-embeddings map		read pattern-embeddings map, extract patterns	shuffle patterns, count and filter, write frequent patterns	
F&R [18] (2-phase)		read graphs, **FSM for each partition**	shuffle partition frequencies, filter candidates, write candidates	read graphs, read candidates, **refine partition frequencies**	shuffle patterns, count and filter, write frequent patterns	
DIMSpan (iterative)	read graphs	receive frequent patterns, pattern growth, update pattern-embeddings map		extract patterns from pattern-embeddings map	count partition frequencies, shuffle partition frequencies, count and filter, **verify frequent patterns**, send frequent patterns	write frequent patterns

MR-FSE is using pattern-embedding maps as its main data structure. In Map 1 k-edge maps of the previous iteration are read from disk. Additionally, all k-edge frequent patterns are read by each worker. Then, graphs are reconstructed based on embeddings, pattern growth is applied and updated maps are written back to disk. Reduce 1 is not used. In Map 2 the grown maps are read again and a record for each pattern and supporting graph is extracted. In Reduce 2, these records are shuffled to count their frequency. After filtering, frequent ones are written to disk.

F&R reads graphs from disk and runs a modified version of Gaston [21], an efficient single-machine algorithm, on each partition in Map 1. Then, a statistical model is used to report partition frequencies of patterns. In Reduce 1, local frequencies are evaluated for each pattern and a set of candidate patterns $\mathcal{P}$ including some frequency information are written to disk. In Map 2 graphs and information about candidate patterns are read from disk. For some partitions, local pattern frequencies may be unknown at this stage. Thus, they are refined by subgraph-isomorphism testing. In Reduce 2, refined pattern frequencies are summed up, filtered and written to disk.

4.2 Cost Comparison

Table 4 shows a comparison of upper bounds for the three stated dimensions. We consider our way of comparing iterative and non-iterative methods as valid since with regard to upper bounds every step can be considered as the union of all k-edge results, e.g., $\mathcal{P} = \mathcal{P}^1 \cup .. \cup \mathcal{P}^k$.

Disk access: I-FSM uses the most voluminous data structure of full subgraphs $\mathcal{S}$. Additionally, these subgraphs are read and written twice. Thus, I-FSM clearly has the highest cost for disk access. MR-FSE uses embedding maps $\mathcal{M}$ as it's main data structure, which is with regard to vertex- and edge labels an irredundant version of $\mathcal{S}$ that describes subgraphs by patterns and embeddings (see Section 2.2). This map is written once and read twice. Additionally, patterns $\mathcal{P}$ are read and written once. Thus, MR-FSE is superior to I-FSM. F&R reads graphs twice but writes no intermediate results despite rather small pattern information. Since the volume of $\mathcal{G}$ roughly corresponds to the one of $\mathcal{S}^1$ or $\mathcal{M}^1$, F&R requires the lowest disk access of the three MapReduce approaches. However, DIMSpan further reduces disk access to a minimum as it is based on a distributed in-memory system. In particular, graphs are read only once from disk before the iterative part and patterns are written only once to disk afterwards.

Network traffic: Since I-FSM shuffles the complete set of subgraphs twice, it clearly causes the most network traffic. All other approaches only exchange pattern information. However, since MR-FSE is neither partition-based like F&R nor uses a combine operation like DIMSpan, a record for each pattern and graph ($|\mathcal{G}| \cdot \mathcal{P}$) may be shuffled among physical machines. With regard to network traffic, F&R and DIMSpan are comparable to each other, especially since both are using compression to further reduce the volume of the few exchanged records.

Isomorphism resolutions: All of the four compared approaches resolve the subgraph isomorphism problem in different ways and with different cardinalities. The respective steps are highlighted by bold font in Table 3. I-FSM calculates a (in [10] not further specified) canonical label from scratch for each grown subgraph and, thus, the isomorphism problem is resolved with maximum cardinality $|\mathcal{S}|$. MR-FSE is using DFS codes like DIMSpan but in [19] it is clearly stated that no verification is performed at any time. Instead, false positives are detected by enumerating all DFS code permutations of each distinct edge set (subgraph) to choose the minimal one. Consequently, isomorphisms among DFS codes are in fact resolved $|\mathcal{S}|$ times, too. F&R is facing the problem in two steps. First, when running FSM for each partition ($w \cdot |\mathcal{P}|$) and, second, when counting patterns by a priori like subgraph isomorphism testing in the refinement step. Since the local frequency of each pattern must be known for at least on partition, the upper bound is not fully $|\mathcal{G}| \cdot |\mathcal{P}|$. For this dimension, DIMSpan is clearly superior because no a priori like operations are applied at any time and every pattern is verified only once.

Summary: DIMSpan shows the lowest costs with regard to all of the stated dimensions. Besides this, DIMSpan is the only approach that provides source code to the public, supports directed multi-graphs and already applies first pruning steps in a preprocessing (see Section 3.6).

5 EVALUATION

In this section we present the results of a performance evaluation of DIMSpan based on a real molecular dataset of simple undirected graphs and a synthetic dataset of directed multigraphs. We evaluate scalability for increasing volume of input data, increasing result sizes (decreasing minimum support) and variable cluster size. For all experiments, we evaluate the improvement gained by our optimizations. Furthermore, we analyze the impact by adding and omitting single optimizations and show their dependency on each other.

5.1 Implementation

We evaluate DIMSpan using Java 1.8.0_102, Apache Flink 1.1.2 and Hadoop 2.6.0. More precisely we use Flink's DataSet API[4] for all transformations and its *bulk iteration* for the iterative part. We further use the Simple16 implementation from JavaFastPFOR[5] for compression. The source code is available on GitHub[6] under Apache licence, version 2.0 (Alv2). To show the impact of our optimizations, we made them configurable. In all evaluations, the term *baseline* refers to a configuration without preprocessing, without compression and pattern verification at reporting, i.e., resolving isomorphism $|\mathcal{G}| \cdot |\mathcal{P}|$ times. We use Flink's aggregation to count pattern frequencies (lines 5,6 of Algorithm 1). To disable the combine step, we would have had to re-implement aggregation using the external API and this would have significantly blurred a potential comparison. Thus, also the baseline contains the combine operation.

5.2 Datasets

We evaluate three data-related dimensions that impact the runtime of a distributed FSM algorithm: structural graph characteristics, *input size* $|\mathcal{G}|$ and *result size* $|\mathcal{F}|$. To show scalability for one of these dimensions, the other two need to be fixed. While $|\mathcal{F}|$ can be increased by decreasing the minimum support threshold, varying the other two dimensions separately is less trivial. Thus, we decided to use two base datasets with divergent structural characteristics and just copy every graph several times to increase $|\mathcal{G}|$ under preservation of structural characteristics and $|\mathcal{F}|$.

The first base dataset is *yeast-active*[7], in the following denoted by *molecular*, a real dataset from anti-cancer research. It was chosen to represent molecular databases because structural characteristics among them do not fundamentally differ due to the rules of chemistry. For example, all molecular databases describe simple undirected graphs with only few different edge labels (e.g., single and double bond) and most frequent patterns are paths or trees [21]. The base dataset contains around 10K graphs (9567) and is scaled up to datasets containing around 100K to 10M graphs. We did not use an optimized version of DIMSpan for undirected graphs but provide an according parameter. If the parameter is set to undirected, the direction indicator (see Section 3.3) will just be ignored. Dedicated application logic is only used when it is unavoidable, for example, an 1-edge DFS code desribing a non-loop edge with two equal vertex labels (automorphism) leads to two embeddings in undirected mode.

The second category of datasets, in the following denoted by *synthetic*, was created by our own data generator[8]. It generates unequally sized connected directed multigraphs where each 10th graph has a different size ranging from $|V| = 10, |E| = 14$ to $|V| = 91, |E| = 140$. There are 11 distinct vertex and $5 + |\mathcal{G}|/1000$ distinct edge labels. The result is predictable and contains 702 frequent patterns with 1 to 13 edges for each min support decrement of 10% (i.e., 702 for 100%, 1404 for 90% , ..). The patterns contain loops, parallel edges (in and against direction), different subgraph automorphisms (e.g., "rotated" and "mirrored") separately as well as in all combinations. The data generator was not only designed for the comparative evaluations but also for testing the correctness of implementations. To verify the number of contained frequent patterns we implemented a simple pruning-free brute-force FSM algorithm and manually verified all patterns of sizes 1..4, 12,13.

Table 4: Cost comparison of DIMSpan and approaches based on MapReduce.

	Pre	M1	R1	M2	R2	Post
disk access						
I-FSM		$\uparrow \mathcal{S}$	$\downarrow \mathcal{S}$	$\uparrow \mathcal{S}$	$\downarrow \mathcal{S}$	
MR-FSE		$\uparrow \mathcal{M}, \mathcal{P} \downarrow \mathcal{M}$		$\uparrow \mathcal{M}$	$\downarrow \mathcal{P}$	
F&R		$\uparrow \mathcal{G}$	$\downarrow \mathcal{P}$	$\uparrow \mathcal{G}, \mathcal{P}$	$\downarrow \mathcal{P}$	
DIMSpan	$\uparrow \mathcal{G}$					$\downarrow \mathcal{P}$
network traffic						
I-FSM			$\mathcal{S}$		$\mathcal{S}$	
MR-FSE		$w \cdot \mathcal{P}$			$\lvert\mathcal{G}\rvert \cdot \mathcal{P}$	
F&R			$w \cdot \mathcal{P}$		$w \cdot \mathcal{P}$	
DIMSpan					$2w \cdot \mathcal{P}$	
isomorphism resolution						
I-FSM				$\lvert\mathcal{S}\rvert$		
MR-FSE		$\lvert\mathcal{S}\rvert$				
F&R		$w \cdot \lvert\mathcal{P}\rvert$		$(\lvert\mathcal{G}\rvert - 1) \cdot \lvert\mathcal{P}\rvert$		
DIMSpan					$\lvert\mathcal{P}\rvert$	

w: number of worker threads (partitions, $w = |W|$)
$\mathcal{P}$: set of all grown patterns
$\mathcal{G}$: set of input graphs ($\mathcal{G} \gg \mathcal{P}$)
$\mathcal{M}$: all grown patterns and their embeddings ($\mathcal{M} \gg \mathcal{G}$)
$\mathcal{S}$: all grown subgraphs (unit of pattern and embedding, $\mathcal{S} > \mathcal{M}$)

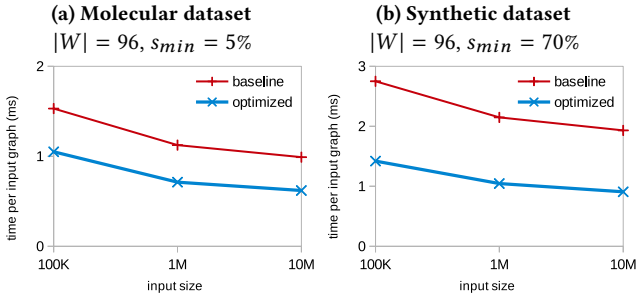
5.3 Experimental Results

All experiments are performed on our in-house cluster with 16 physical machines equipped with an Intel E5-2430 2.5 Ghz 6-core CPU, 48 GB RAM, two 4 TB SATA disks and running openSUSE 13.2. The machines are connected via 1 Gigabit Ethernet.

Input Size: Figure 4 shows measurement results for increasing input size $|\mathcal{G}|$ for both datasets under fixed minimum support thresholds on a cluster with 16 machines and 96 worker threads ($|W| = 96$). To compare runtimes for different input sizes the charts show the average time to process a single input graph for the molecular (4a) and the synthetic dataset (4b). This time is constantly decreasing with an increasing input size for both workloads. The reason is our optimization strategy that verifies DFS codes after counting (see Section 3.5) which makes the number of isomorphism

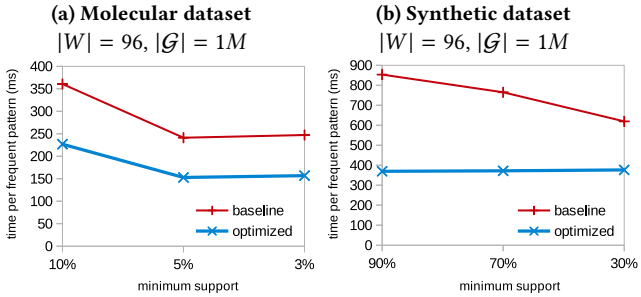
[4] https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html
[5] https://github.com/lemire/JavaFastPFOR
[6] https://github.com/dbs-leipzig/gradoop; org.gradoop.examples.dimspan
[7] https://www.cs.ucsb.edu/~xyan/dataset.htm
[8] org.gradoop.flink.datagen.transactions.predictable

Figure 4: Scalability for varying input size.



dataset	molecular			synthetic		
$t_b / \|\mathcal{G}\|$ (input size)	100K	1M	10M	100K	1M	10M
baseline runtime total t_b (sec)	153	1124	9902	275	2148	19315
optimized runtime total t_o (sec)	105	712	6193	142	1045	9076
$t_b / \|\mathcal{G}\|$ (ms)	1.5	1.1	1.0	2.8	2.1	1.9
$t_o / \|\mathcal{G}\|$ (ms)	1.1	0.7	0.6	1.4	1.0	0.9
improvement $(t_b - t_o)/t_b$	31%	37%	37%	48%	51%	53%

Figure 5: Scalability for varying result size.

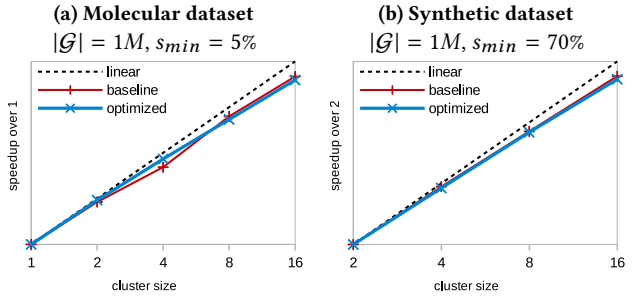


dataset	molecular			synthetic		
s_{min} (minimum support)	10%	5%	3%	90%	70%	30%
$\|\mathcal{F}\|$ (result size)	1270	4660	12807	1404	2808	5616
baseline runtime total t_b (sec)	458	1124	3166	1198	2148	3478
optimized runtime total t_o (sec)	288	712	2010	519	1045	2115
$t_b / \|\mathcal{F}\|$ (ms)	361	241	247	853	765	619
$t_o / \|\mathcal{F}\|$ (ms)	227	153	157	370	372	377
improvement $(t_b - t_o)/t_b$	37%	37%	37%	57%	51%	39%

resolutions only dependent on the result size, which is fixed in this benchmark. For the same reason the *improvement* of our optimized configuration in comparison to the baseline (last row of the table in Figure 4) is slightly increasing for larger data sets. This outcome confirms the positive effect of minimizing the total number of isomorphism resolutions.

Result Size : Figure 5 shows measurement results for decreasing minimum support, i.e., increasing result size $|\mathcal{F}|$, for both datasets under fixed input size on a cluster with 16 machines. The charts show the average time to extract a single frequent pattern for the molecular (5a) and the synthetic dataset (5b). Except for small result size on the molecular dataset, this time is constant for the optimized version on both workloads, while the baseline time is decreasing for increasing input size. This shows, that the total runtime of the optimized version only depends on the result size, which is a desirable behavior. In contrast to the molecular dataset, the improvement on the synthetic workload is decreasing for larger results. The reason is, that due to its label diversity a relatively large part of the input data can be pruned during the preprocessing for the synthetic dataset while rather rare as well as extremely frequent

Figure 6: Horizontal scalability for varying cluster size.



dataset	molecular					synthetic				
physical machines	1	2	4	8	16	1	2	4	8	16
worker threads $\|W\|$	6	12	24	48	96	6	12	24	48	96
baseline runt. (sec)	14292	7521	4437	2054	1124	35586	14538	7540	4022	2148
optimized runt. (sec)	8588	4394	2354	1293	712	16470	6829	3599	1909	1045
improvement	40%	42%	47%	37%	37%	54%	53%	52%	53%	51%
baseline speedup	1.0	1.9	3.2	7.0	12.7		1.0	1.9	3.6	6.8
optimized speedup	1.0	2.0	3.6	6.6	12.1		1.0	1.9	3.6	6.5

Table 5: Impact of single optimization techniques.

configuration	modification	molecular runtime	molecular improv.	synthetic runtime	synthetic improv.
baseline		1124		2148	
optimized		712	37%	1045	51%
baseline	with preprocessing	1594	-42%	1606	25%
optimized	without preprocessing	717	36%	2566	-19%
baseline	with compression	984	12%	2050	5%
optimized	without compression	1235	-10%	1511	30%
baseline	post counting verification	1191	-6%	3402	-58%
optimized	pre counting verfication	1345	-20%	1369	36%

$|W| = 16, |\mathcal{G}| = 1M$, mol.: $s_{min} = 5\%$, syn.:$s_{min} = 70\%$

patterns in the molecular database contain the same atoms (vertex labels) and bonds (edge labels).

Cluster Size : Figure 6 shows measurement results for a variable cluster size, i.e., increasing number of worker threads $|W|$, for both datasets with fixed input size and under fixed minimum support thresholds. The charts show the speedup gained over one machine for the molecular (6a) and over two machines for the synthetic (6b) dataset. The latter was chosen since we achieved a superlinear speedup from 1 to 2 machines. Similar effects occur for 10K and 100K synthetic graphs as well as for different minimum support thresholds. We cannot explain these effects and thus attribute them to Apache Flink's program execution. For larger cluster sizes, we see that DIMSpan scales sligtly sublinear but still achieves notable speedups on both datasets for an increasing number of machines. The slight decreases compared to an optimal speedup is influenced by the fact that the baseline already contains our efficient data structure and a combine operation for counting that minimizes network traffic. Further on, the number of shuffled records in the counting phase is smaller for the baseline since false positives are verified before sending them over the network.

Single Optimizations : Table 5 shows the impact of adding the individual optimizations to the baseline and omitting single optimizations from the optimized configuration while all of the previously varied dimensions are fixed.

The parameter *preprocessing* enables removing vertices and edge with infrequent labels (see Section 3.6) and applying the merge strategy in pattern growth (see Section 3.4). Within the measured

minimum support thresholds there were nearly no infrequent labels in the molecular dataset. Thus, adding a preprocessing to the baseline even lead to a slowdown and is just balanced by the merge strategy for omission. For the synthetic dataset, we see a notable speedup for addition and an immense slowdown for omission.

Compression leads to smaller records and, thus, to fewer network traffic and faster counting. We see that omission leads to a larger slowdown than the addition's speedup. The reason is, that due to the post counting verification the optimized version counts and shuffles more records than the baseline.

Moving *verification* behind counting lead to a slowdown for addition and omission on both workloads. The addition slowdown is originated by the missing compression, i.e., the increased time for counting and shuffling is higher than the time saved by fewer isomorphism resolutions. On the other hand, moving verification before counting lead to an even greater slowdown, which again confirms the positive effect of this strategy.

In summary, we observed that the effects of our optimizations highly depend on each other as well as on dataset characteristics.

6 CONCLUSIONS AND FUTURE WORK

We proposed DIMSpan, the first approach to parallel transactional FSM that combines the effective search space pruning of a leading single-machine algorithm with the technical advantages of state-of-the-art distributed in-memory dataflow systems. DIMSpan is part of Gradoop [14, 24], an open-source framework for distributed graph analytics. Our experimental evaluation showed the high scalability of DIMSpan for large datasets, low minimum support thresholds and increasing cluster size. We found that different optimizations depend on each other and should be chosen with regard to data set characteristics. A functional comparison to approaches based on MapReduce (Section 4) has shown that DIMSpan is superior in terms of network traffic, disk access and the number of isomorphism resolutions. Additionally, it is the only approach to frequent subgraph mining on shared nothing clusters that supports directed multigraphs and that is available for practical application.

In future work, we will use DIMSpan as the basis for advanced graph mining techniques on shared nothing clusters such as generalized and multi-dimensional frequent subgraph mining [23].

7 ACKNOWLEDGMENTS

This work is partially funded by the German Federal Ministry of Education and Research under project ScaDS Dresden/Leipzig (BMBF 01IS14014B).

REFERENCES

[1] C. C. Aggarwal and J. Han. *Frequent pattern mining*. Springer, 2014.
[2] S. Aridhi, L. D'Orazio, M. Maddouri, and E. Mephu. A novel mapreduce-based approach for distributed frequent subgraph mining. In *Reconnaissance de Formes et Intelligence Artificielle (RFIA)*, 2014.
[3] M. A. Bhuiyan and M. Al Hasan. An iterative mapreduce based frequent subgraph mining algorithm. *Knowledge and Data Engineering, IEEE Transactions on*, 27(3):608–620, 2015.
[4] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *IEEE International Conference on Data Mining (ICDM)*, pages 51–58, 2002.
[5] B. Bringmann and S. Nijssen. What is frequent in a single graph? In *PAKDD*, pages 858–863. Springer, 2008.
[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, page 28, 2015.
[7] R. Cyganiak, A. Harth, and A. Hogan. N-quads: Extending n-triples with context. *W3C Recommendation*, 2008.
[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. volume 51, pages 107–113. ACM, 2008.
[9] M. Elseidy, E. Abdelhamid, S. Skiadopoulos, and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528, 2014.
[10] S. Hill, B. Srichandan, and R. Sunderraman. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *Proc. ACM Conf. on Bioinformatics, Computational Biology and Biomedicine*, pages 661–666, 2012.
[11] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *IEEE Int. Conf. on Data Mining (ICDM)*, pages 549–552, 2003.
[12] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer, 2000.
[13] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Eng. Review*, 28(01):75–105, 2013.
[14] M. Junghanns, A. Petermann, N. Teichmann, K. Gómez, and E. Rahm. Analyzing extended property graphs with apache flink. In *Proc. ACM SIGMOD Workshop on Network Data Analytics*, pages 3:1–3:8, 2016.
[15] R. Kessl, N. Talukder, P. Anchuri, and M. Zaki. Parallel graph mining with gpus. In *BigMine*, pages 1–16, 2014.
[16] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *IEEE Int. Conf. on Data Mining (ICDM)*, pages 313–320, 2001.
[17] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.
[18] W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in mapreduce. In *International Conference on Data Engineering (ICDE)*, pages 844–855. IEEE, 2014.
[19] W. Lu, G. Chen, A. Tung, and F. Zhao. Efficiently extracting frequent subgraphs using mapreduce. In *IEEE Int. Conf. on Big Data*, pages 639–647, 2013.
[20] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.
[21] S. Nijssen and J. N. Kok. The gaston tool for frequent subgraph mining. *Electronic Notes in Theoretical Computer Science*, 127(1):77–87, 2005.
[22] S. Nijssen and J. N. Kok. Frequent subgraph miners: runtimes don't say everything. *MLG 2006*, page 173, 2006.
[23] A. Petermann et al. Mining and Ranking of Generalized Multi-Dimensional Frequent Subgraphs. In *IEEE Int. Conf. on Digital Inf. Management (ICDIM)*, 2017.
[24] A. Petermann, M. Junghanns, S. Kemper, K. Gómez, N. Teichmann, and E. Rahm. Graph mining for complex data analytics. In *IEEE Int. Conf. on Data Mining Workshops (ICDMW)*, pages 1316–1319, 2016.
[25] A. Petermann, M. Junghanns, R. Müller, and E. Rahm. Graph-based Data Integration and Business Intelligence with BIIIG. *PVLDB*, 7(13), 2014.
[26] S. Ranu and A. K. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *IEEE Int. Conf. on Data Engineering (ICDE)*, pages 844–855. IEEE, 2009.
[27] A. Stratikopoulos et al. Hpc-gspan: An fpga-based parallel system for frequent subgraph mining. In *IEEE Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2014.
[28] C. H. Teixeira et al. Arabesque: a system for distributed graph mining. In *Proc. of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM, 2015.
[29] L. T. Thomas, S. R. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):10, 2010.
[30] B. Vo, D. Nguyen, and T.-L. Nguyen. A parallel algorithm for frequent subgraph mining. In *Advanced Computational Methods for Knowledge Engineering*, pages 163–173. Springer, 2015.
[31] M. Wörlein et al. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 392–403. Springer, 2005.
[32] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *IEEE International Conference on Data Mining (ICDM)*, pages 721–724, 2002.
[33] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Technical Report UIUCDCS-R-2002.2296*, 2002.
[34] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *KDD*, pages 286–295. ACM, 2003.
[35] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2, 2012.
[36] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web*, pages 387–396. ACM, 2008.