Michael Rudolf, Marcus Paradies, Christof Bornhövd, Wolfgang Lehner

SynopSys - large graph analytics in the SAP HANA database through summarization

SLUB
Wir führen Wissen.

TECHNISCHE
UNIVERSITÄT
DRESDEN

QUCOSA
Quality Content of Saxony

SynopSys: Large Graph Analytics in the SAP HANA Database Through Summarization

Michael Rudolf[1]   Marcus Paradies[1]   Christof Bornhövd[2]   Wolfgang Lehner[3]

[1]SAP AG
Walldorf, Germany
michael.rudolf01@sap.com

[2]SAP Labs, LLC
Palo Alto, CA 94304, USA
christof.bornhoevd@sap.com

[3]Database Technology Group
TU Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

Graph-structured data is ubiquitous and with the advent of social networking platforms has recently seen a significant increase in popularity amongst researchers. However, also many business applications deal with this kind of data and can therefore benefit greatly from graph processing functionality offered directly by the underlying database. This paper summarizes the current state of graph data processing capabilities in the SAP HANA database and describes our efforts to enable large graph analytics in the context of our research project SynopSys. With powerful graph pattern matching support at the core, we envision OLAP-like evaluation functionality exposed to the user in the form of easy-to-apply graph summarization templates. By combining them, the user is able to produce concise summaries of large graph-structured datasets. We also point out open questions and challenges that we plan to tackle in the future developments on our way towards large graph analytics.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Storage and Retrieval—*information search and retrieval*

General Terms

Algorithms, Design, Performance

Keywords

Graph matching, graph transformation, graph summarization, SAP HANA database system

1. INTRODUCTION

With the incessantly growing number of participants in social networks and their constant production of interlinked content, the last few years have seen a massive rise in the amount and complexity of graph-structured data. As new technologies were needed for efficiently managing and processing this kind of information in this new order of magnitude, more and more researchers have turned
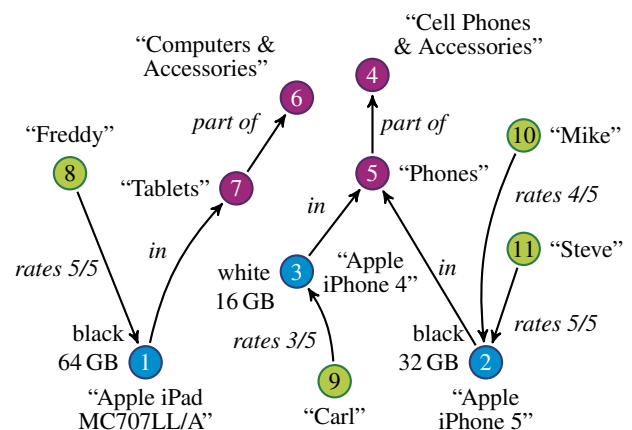
Figure 1: Example data expressed in the property graph data model. Edge attributes are typeset in italics; vertex types are indicated through different colors.

their attention to this subject and fostered the NoSQL and BigData movements. However, traditional business applications often also have to deal with graphs, for example in supply network management and traceability, transportation and logistics, as well as for the classic bill of materials. It is therefore safe to say that graph data is ubiquitous in all kinds of business applications and has a long history.

Graphs come in many different flavors. In its most basic form a graph G is made up of a set of vertices V and a relation $E \subseteq V \times V$ representing the edges between them (sometimes also noted as $V(G)$ and $E(G)$, respectively). Depending on the use case, this definition is being extended or altered accordingly (e.g., for undirected graphs or hypergraphs). In the remainder of this paper we will focus on the property graph model [10], because it is very general but nevertheless flexible enough to support many use cases. Also, other graph models can easily be mapped to the property graph model.

A property graph is a directed vertex-labeled and edge-labeled graph, meaning that both vertices and edges can have attributes each consisting of a key and a value. Figure 1 shows an example of a property graph for products, categories, users, and ratings, which could serve as the underlying data for a recommendation engine. Sometimes it can be helpful to have a dedicated attribute designate the semantic type of the vertex or edge. However, this semantic type does not imply any structural constraints, i.e., two vertices of the same type can have different attributes. In Figure 1 the type attribute
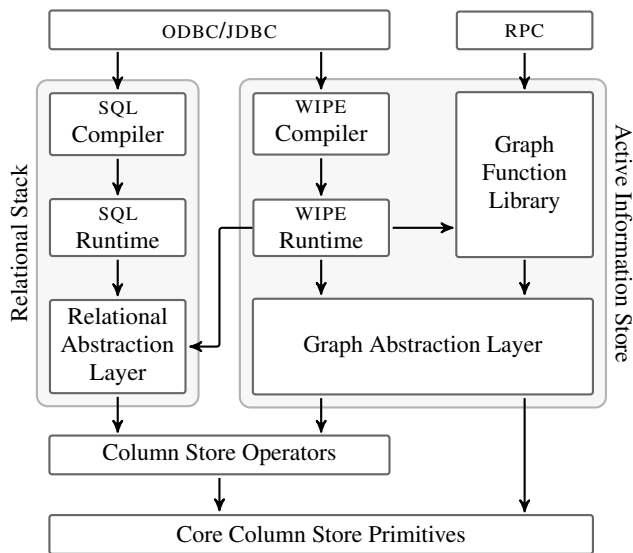
Figure 2: Integration of the graph processing functionality within the SAP HANA database [11].

of vertices is not made explicit, rather it is indicated through the different colors for the sake of simplicity.

Making sense of large amounts of graph-structured data is a challenging task. Business applications need to extract meaningful information from large graphs in reasonable time for decision makers to rely upon. Therefore, in addition to providing the classic OLAP functionality, graph summaries should be produced in such a way that they help uncover knowledge embodied in the graph topology.

In this paper we describe the existing and upcoming graph processing functionality within the SAP HANA database system. The following section shortly introduces SAP HANA and gives an overview of the available graph processing support. In Section 3 we analyze related work in the field of graph summarization. Section 4 presents large graph analytics by means of pattern matching and templates as our main contribution. This is followed by Section 5, which illustrates the main challenges in graph pattern matching for us to tackle in the context of our research project SynopSys. Finally, Section 6 concludes the paper with a sketch of future work.

2. GRAPH PROCESSING IN THE SAP HANA DATABASE

Situated at the core of the SAP HANA Appliance product, the SAP HANA database is an in-memory relational database system that was designed for supporting complex analytical business processes as well as for handling high transactional workloads [6]. Exploiting the recent hardware developments, it leverages large amounts of main memory and multi-core CPUs to provide high-performance storage and processing capabilities for a tremendously heterogeneous spectrum of business applications [12]. To that end, the SAP HANA database federates a hybrid relational engine supporting both row- and column-oriented physical representations with engines for text search and graph processing.

The graph data processing capabilities have only recently been added to the SAP HANA database and are tightly integrated into the in-memory column store engine, as is shown in Figure 2. They rely on universal tables for storing vertices and edges, with each attribute being mapped to a table column. Exploiting the characteristics of

the columnar data layout, the operations for adding and removing vertex and edge attributes show good performance, because they do not imply any physical reorganizations. By offering graph processing functionality directly within the database core instead of creating a new layer on top of it, we can leverage the infrastructure and efficiently integrate it with the relational engine. Thus, the established relational data processing toolkit is extended with graph processing functionality in a way, such that data can be queried and manipulated in the same place without having to convert it into a different format. With this powerful foundation we envision the creation of completely novel types of business applications; but also existing business applications dealing with graph-structured information can benefit tremendously from these new capabilities.

As part of the ongoing Active Information Store project [2], support for the declarative domain-specific graph query and manipulation language WIPE was implemented. In addition to the basic operations for creating, updating, and deleting vertices and edges, it allows the combination of BI-like aggregation operations with efficient path traversals. The latter are carried out with the help of a dedicated database execution plan operator. WIPE permits multiple complex operations to be combined within a single statement, thereby reducing the need for several roundtrips between the application and the database system. Similar to other domain-specific languages of the SAP HANA database, statements are executed within a transaction context, so that the system guarantees atomicity, consistency, durability, and the required level of isolation.

In addition to that, a graph abstraction layer with an object-oriented programming interface has been introduced, which enables the implementation of custom graph algorithms for example within a stored procedure [11]. A set of parameterizable implementations of frequently-used algorithms will be provided in the form of a Graph Function Library for application developers to choose from.

3. RELATED WORK

In order to understand and use the information encoded in large graphs it is crucial to be able to summarize them and by that, to extract small pieces of information that are meaningful within a particular context of use. The problem of finding good graph summaries is also related to the fields of graph visualization and graph compression [9].

In a 2008 paper a group from the Universities of Illinois at Urbana-Champaign and Chicago presented their approach for extending online analytical processing (OLAP) to graphs [3]. Together with the IBM T.J. Watson Research Center they contribute the basic definitions of dimensions and map the well-known OLAP operations roll-up, drill-down, and slice/dice to them. They assume snapshots of a graph changing over time and consider the associated attributes the *informational dimensions*. In contrast, the *topological dimensions* are those coming from the attributes of vertices and edges in each such snapshot. In their paper they also present a theoretical foundation for computing aggregated graphs, which form the measures in OLAP terminology, and show how partial materialization techniques can help reduce memory consumption. However, their approach is not accompanied by any processing or evaluation specification, concepts, or architecture.

In a different paper from the University of Illinois at Urbana-Champaign this time in collaboration with Microsoft and Google, a novel data warehousing model called Graph Cube [15] is introduced. Based on a restricted graph model (e.g., no attributes on edges) introduced as multidimensional network (with the dimensions being the vertex attributes), they define the notion of an aggregate network (called *cuboid*). A graph cube constitutes then the set of all possible aggregations of the original network. The authors foresee two kinds
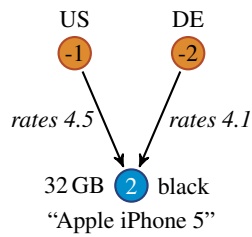
Figure 3: Summary graph showing the average product rating of all US-american and German users. Negative identifiers indicate newly created vertices.

Figure 4: Graph summarization rule for deriving the summary graph shown in Figure 3.

of OLAP operations: cuboid and crossboid queries. The former simply returns the aggregate network of the desired cuboid from the graph cube, while the latter can be seen as somewhat similar to a join operation between multiple different cuboids. As for the specification and evaluation of such queries, the authors do not propose any mechanism and again focus on partial materialization techniques.

Researchers from the University of Michigan and the Nokia Research Center have proposed the SNAP operation for grouping vertices based on user-selected attributes and pairwise relationships [13]. The resulting vertex groups are homogenous with respect to the selected attributes and their relationships. This means that if two groups are connected via an edge, each vertex of one group is connected to some vertices of the other, whereas if the groups are not connected, no vertex of one group is connected to any vertex of the other. In practice this behavior turns out to be quite limiting, because it can result in a large number of groups. Therefore, they propose the k-SNAP operation as an extension, where the homogeneity constraint for group relationships is relaxed and the user can specify the number of groups in the graph summaries. Changing this number then has the same effect as the OLAP operations drill-down and roll-up. They prove that the computation of the k-SNAP operation is NP-complete and propose heuristics to approximate it. Although the two proposed operations are designed to work with different edge types, additional edge attributes are not supported.

In a follow-up paper by a group from the University of Wisconsin-Madison and the IBM Almaden Research Center [14], the previous approach is improved in two ways: first, the homogeneity requirement for vertex attributes is relaxed by permitting the user to specify the number of partitions for numerical attributes. Second, for helping users specifying a sensible number of groups for the k-SNAP operation, an interestingness measure for graph summaries is defined. It is based on different aspects the authors describe as diversity, coverage, and conciseness of a summary.

Other approaches to graph summarization are mostly statistical in nature. They compute sets of figures (e.g., degree distributions, hop-plots, and clustering coefficients), which describe the characteristics of the graph. The approaches presented above are different: they can produce aggregated views on the graph data in various, user-controlled resolutions by means of OLAP-like operations. Although much more flexible, in our opinion these approaches are still too rigid in some ways. For example, ideally a user should be able to query the property graph in Figure 1 for the average rating of a specific product differentiated by US-american and German users, and the system should return the summary graph shown in Figure 3 as the result. Furthermore, neither of the approaches proposes a method for specifying or evaluating arbitrary graph summarization recipes.
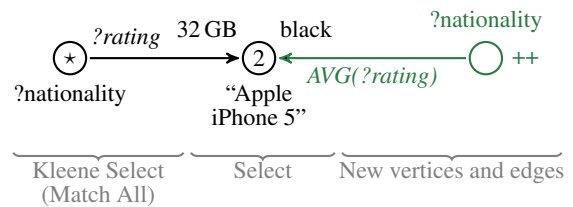
4. GRAPH ANALYTICS THROUGH SUMMARIZATION

We propose an approach to graph summarization that is related to the transformation of graphs using graph grammars: with the help of graph patterns a user can identify the items of interest and produce a summary of them.

4.1 Pattern Matching as the Foundation

Pattern matching in graphs has a broad range of applications. In addition to the well-known use cases in the fields of pattern recognition and artifical intelligence, there are also many business applications that can benefit from such functionality. For instance, fraud detection systems try to find re-occurring patterns in insured events or money withdrawals that could indicate criminal activity. Another example are purchase recommendation engines, which can use graph patterns to capture the context information for a specific customer and match other purchase orders that are related in some way and might be of interest to that customer.

For a graph pattern matching technology serving as the foundation of large graph analytics we see the following requirements:

- **Match Modes:** With increasing data volumes it makes sense to differentiate between two match modes: match-all and match-any. As the names suggest, the former returns all occurrences of the pattern in the graph, while the latter returns only the one found first and stops the matching process. This distinction permits the user to indicate that only the existence of a match is of relevance and thereby helps saving computation resources.

- **Versatility:** For a general graph pattern matching functionality to be of sufficient practical use, a certain versatility is inevitable. This is especially true with regard to the supported *predicate types*. In practice, only offering value-based equality comparisons might quickly turn out to be a limiting factor; relational comparisons, regular expressions, and negation (i.e., testing for the non-existence of some property) is also needed.

- **Regular Path Expressions:** In some cases the basic building blocks for graph patterns (i.e., vertices, edges, and attributes) are not expressive enough, for example whenever a pattern should reflect that two vertices have to be connected via an arbitrary number of hops (i.e., there is a path connecting the two vertices). This can be achieved with the help of regular path expressions [5], which permit the specification of regular expressions for such paths.

In order to actually construct graph summaries, a user has to specify an action to be executed once a graph pattern matches. Both the graph pattern and the actions constitute a *summarization rule*. Figure 4 shows such a summarization rule for deriving the summary graph depicted in Figure 3 and illustrates the required functionality

beyond pattern matching. Operations for adding vertices, edges, and attributes are indicated in green color. We extend the concept of match modes to also apply to a single vertex instead of the whole pattern. By marking at most one vertex with a star symbol $\star$, we express that for each match of the rest of the pattern all matches of that vertex will be grouped to form a single result. In the absence of the star symbol, the default match mode match-any is used. By prepending an attribute name with a question mark, we define a variable of that name and bind it to all the values in such a group. Finally, for deriving meaningful information from the grouped values, we require a set of aggregation functions to apply to such variables.

4.2 Summarization Templates

Summarization rules are a powerful tool and can quickly become quite complex. To reduce the hurdles for user adoption we propose to break down the summarization functionality into several simpler templates, which are described in the following. They have to be instantiated by specifying arguments for their parameters and can then be combined by applying them sequentially.

To distinguish the templates from ordinary summarization rules, we introduce a slightly different notation: Template parameters are enclosed in angle brackets $\langle$ and $\rangle$ and printed in blue color if they constitute a part of the graph pattern. New vertices, edges, and attributes are printed in green color.

Collecting Attribute Values

For summarizing information in graphs it is required to collect the attribute values of a set of vertices or edges. The summarization templates shown in Figure 5 can do exactly that. Since multiple vertices or edges can have the same attribute values, using a set for collecting them would result in the undesired elemination of duplicates. On the other hand, a list would require an ordering of vertices or edges. Therefore, we use multisets for holding attribute values, indicated by the delimiter symbols $\{|$ and $|\}$.

The template on the left-hand side of Figure 5a expects the vertex predicates vp_1 and vp_2, the edge predicate ep, the source vertex attribute name a_s, and the target vertex attribute name a_t. Note that the edge direction is part of the edge predicate and indicated in the figure using half arrowheads pointing in both directions. The match mode of the source vertex is *match-all*, meaning that for a fixed matching vertex determined by vp_2 all vertices satisfying vp_1 that are connected to the former via an edge satisfying ep contribute their attribute values to the new multiset attribute.
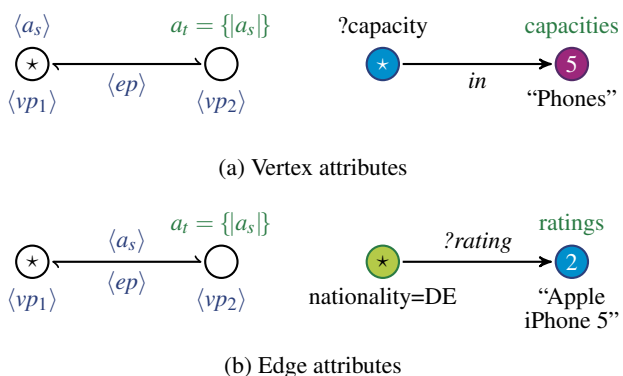

(a) Vertex attributes

(b) Edge attributes

Figure 5: Summarization templates and example instantiations for collecting attribute values.

(a) Vertex attribute
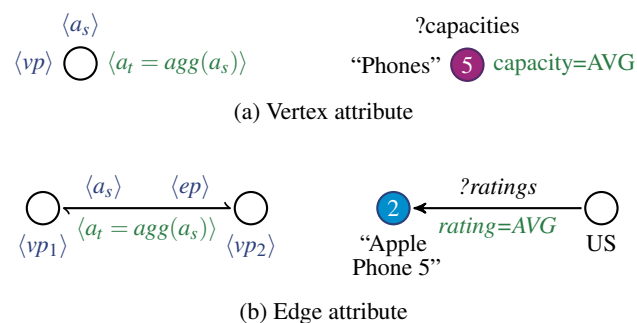

(b) Edge attribute

Figure 6: Summarization templates for aggregating attribute values.

The right-hand side of Figure 5a shows an exemplary instantiation of the template to illustrate its use. It collects all (storage) capacities of products in the category "Phones" into a new multiset attribute that is attached to the vertex representing the category. The different vertex colors are used to represent vertex types consistent with Figure 1 and make up the vertex predicates vp_1 and vp_2. The latter furthermore filters vertices based on the identifier "5". The edge predicate ep is the edge type *in*, the name of the source vertex attribute a_s is *capacity* (prepended with a question mark to differentiate it from a predicate and convey the notion of a variable), and finally the name of the target vertex attribute is *capacities*.

The approach for collecting edge attribute values is similar: the template depicted in the left-hand side of Figure 5b only expects a_s to be the name of an edge attribute instead of a vertex attribute. Then, not the matching source vertices but the edges connecting them to a matching target vertex contribute their attribute values to the new multiset attribute.

The example on the right-hand side of Figure 5b collects all ratings of a specific product given by German users into a new multiset attribute attached to that product vertex. vp_1 is composed of the vertex type and a filter for nationality attribute; vp_2 checks the vertex identifier in addition to the type. The template instantiation does not specify an argument for the edge predicate parameter ep, so that all edges will match the pattern. The source edge attribute name a_s is *rating* and the name of the target vertex attribute is *ratings*.

Scalar Aggregation

When collecting attribute values into multisets, it is desirable to compute aggregate values, such as sums or averages. This can be done with the help of one of the two summarization templates shown in Figure 6.

For aggregating a multiset of vertex attribute values, the following things have to be specified for the template depicted on the left-hand side of Figure 6a: the vertex predicate vp, the name of the source vertex attribute a_s, the name of the target vertex attribute a_t, and the aggregation function *agg*. For every matched vertex, the application of a template instance will yield a new vertex attribute.

The exemplary instantiation on the right-hand side of Figure 6a computes the average (storage) capacity of all products in the category "Phones" (see also Figure 5a). The vertex predicate vp consists of the vertex type (expressed through the color) and identifier "5". The name of the source vertex attribute a_s is *capacities*; it has to designate a multiset that is then passed to the aggregation function *agg* – in the example this is the *AVG* function. The name of the target vertex attribute a_t is specified as *capacity* and can be used to access the computed average value after the application of the summarization rule.

$\langle a_s \rangle$ $\langle ea_1, \ldots, ea_n \rangle$ $a_t = \{|a_s|\}$ ★ ++ ?capacity capacities ★ ++
$\langle vp \rangle$ $\langle va_1, \ldots, va_m \rangle$ "Apple*" "Apple Products"

(a) Collapse vertices

$\langle va_s \rangle$
$\langle vp_1 \rangle$ ★
$\langle ep \rangle$ $\langle ea_s \rangle$
$\langle vp_2 \rangle$ ○ $\langle ea_1, \ldots, ea_n \rangle$ $va_t = \{|va_s|\}$ ++
$ea_t = \{|ea_s|\}$ $\langle va_1, \ldots, va_m \rangle$

nationality=US
★
?rating
2 ratings ++
"Apple iPhone 5" US
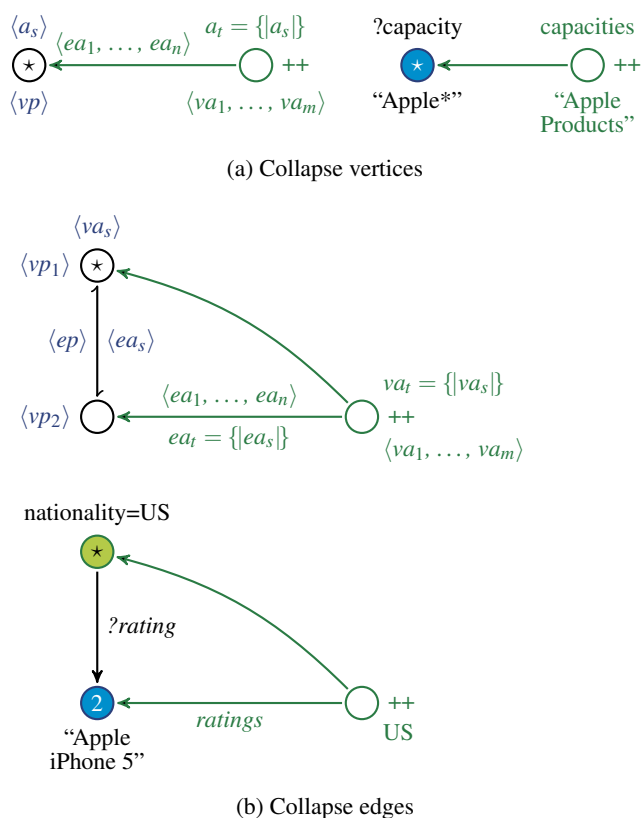
(b) Collapse edges

Figure 7: Summarization templates and example instantiations for collapsing vertices and edges.

The template on the left-hand side of Figure 6b can be used for aggregating a multiset of edge attribute values in a similar way. It expects the vertex predicates vp_1 and vp_2 and the edge predicate ep; a_s and a_t denote the names of edge attributes.

The application of this summarization template results in a new attribute for every matched edge. This is illustrated by the example on the right-hand side of Figure 6b, which computes the average rating of a specific product for a user group.

Collapsing Vertices and Edges

The most complex summarization templates are the ones for collapsing vertices and edges shown in Figure 7. Their main purpose is to introduce new vertices that represent a group of related vertices or edges.

When collapsing vertices with the help of the template depicted on the left-hand side of Figure 7a, the user can specify the vertex predice vp, the source vertex attribute name a_s, the target vertex attribute name a_t, and additional attributes $va_1, \ldots, va_m$ and $ea_1, \ldots, ea_n$ for the vertices and edges to be created.

Similar to the summarization templates for collecting attribute values, the match mode of the source vertex is match-all. That means that for all vertices satisfying vp a new vertex representing them will be created with the attributes $va_1, \ldots, va_m$ and connected to them via new edges with the attributes $ea_1, \ldots, ea_n$. Furthermore, the values of the source vertex attribute va_s will be collected into a new multiset attribute va_t of the new vertex.

The right-hand side of Figure 7a shows an example for collapsing all Apple products into a single vertex while at the same time

collecting their (storage) capacities into a new multiset attribute. The vertex predicate vp is a regular expression, the source vertex attribute name a_s is *capacity*, and the target vertex attribute name is *capacities*. While no additional edge attributes $ea_1, \ldots, ea_n$ have been specified, a single additional vertex attribute with the value "Apple Products" will be created.

The template for collapsing edges is shown in the upper part of Figure 7b. Its parameters are the vertex predicates vp_1 and vp_2, the names of the source vertex attribute va_s and the target vertex attribute va_t, the edge predicate ep, the edge direction, the names of the source edge attribute ea_s and the target edge attribute ea_t, and additional attributes $va_1, \ldots, va_m$ and $ea_1, \ldots, ea_n$ for the vertices and edges to be created.

The match mode of the source vertex is again match-all. Thus, for each vertex v satisfying vp_2 a new vertex will be created with the attributes $va_1, \ldots, va_m$ and connected to it via a new edge e with the attributes $ea_1, \ldots, ea_n$. Then all vertices satisfying vp_1 that are connected to v via an edge satisfying ep will be matched and an edge to the newly created vertex will be added. Finally, the values of the source vertex attribute va_s will be collected into a new multiset attribute va_t of the new vertex v and the values of the source edge attribute ea_s will similarly be gathered into a new multiset attribute ea_t of the new edge e.

The lower part of Figure 7b illustrates the template with an example that will collapse all US-american users and their rating of a specific product into a new vertex. The vertex predicate vp_1 is composed of one filter for the vertex type and another one for the nationality; vp_2 checks the vertex identifier and type. Since the template instantiation does not specify an argument for the edge predicate parameter ep, all edges will match the pattern. The name of the source edge attribute ea_s is *rating*; all values in a match will be collected in the new multiset edge attribute ea_t named *ratings*. The new *nationality* attribute with the value "US" is the only additional vertex attribute; no additional edge attributes $ea_1, \ldots, ea_n$ have been specified and the names of the source and target vertex attributes va_s and va_t have been omitted as well.

5. CHALLENGES IN GRAPH PATTERN MATCHING

The foundation of the template-based graph summarization approach for large graph analytics presented in the previous section is a powerful graph pattern matching mechanism. The problem of finding occurrences of a pattern within a larger graph has already been investigated for decades [4]. However, only very few commercial products actually offer a solution to it. In this section we formulate the challenges that we need to tackle in order to provide an effective and efficient matching technology.

Fuzzy Matching

Business applications having to process large amounts of data are often confronted with erroneous information (e.g., information that contains spelling mistakes or is outright false). As a consequence, strict pattern matching will in many cases not return satisfactory results. However, when extended with the notion of *fuzziness*, matching algorithms can be made lenient towards the problems outlined above and also return valuable results that only match the given pattern to some degree.

There are several approaches to achieve this, the most obvious ones stemming from the area of fuzzy text search, where the metric used to rank inexact matches is based on some edit distance defined on graphs [7]. Still, for some use cases computing such an edit distance with a fixed algorithm might be insufficient and a

more fine-grained mechanism may be required: the most flexible approach would certainly permit users to specify weights indicating which vertices, edges, or attributes in the pattern are more important than others and which might even be mandatory for a match to be considered valid.

Pattern Representation

A question open for debate is the *representation of patterns*. For example, a dedicated domain-specific languange based on regular path expressions [5] could be used. In general, a declarative approach (i.e., specifying what to match) is preferable over a procedural encoding of how to perform the matching (possibly against a system-provided programming interface), because it can be better optimized.

The example instantiations of the templates presented in the previous section are specified as property graphs themselves. On the right-hand side of Figure 7a a pattern for matching certain products by name uses a regular expression. However, the data model usually only permits a limited set of types for attribute values (e.g., integers, floating point numbers, and strings) and does not support designating vertex or edge attribute with special semantics. Still, for a matching algorithm to differentiate between the various predicate types (e.g., relational comparisons, regular expressions, and negation), this would be required. One possible solution could be to rely on a convention for the type and structure of attribute values as is done in the Metaweb Query Language of Freebase [1].

Leveraging Existing Functionality

By implementing support for graph matching as part of the graph abstraction layer directly within the SAP HANA database, we can benefit from the efficiency of the in-memory column store operations. For example, wherever possible we should try to map the different predicate types to similar functionality in the relational engine or the text engine. Another case for re-using existing functionality are regular path expressions. They could be implemented with the help of the traversal operator that is part of the graph abstraction layer in the SAP HANA database.

Exploiting Parallelism

The widespread availability of multi-core CPUs and the parallel execution support of the SAP HANA database provide the foundation for fast data processing. By carefully choosing easily parallelizable graph matching algorithms [8] we can exploit this concurrency potential and craft an efficient solution for large graph analytics. In addition, this approach also scales in the case of very large graphs that are partitioned and distributed over a number of database instances.

6. CONCLUSION AND OUTLOOK

In this paper we have given an overview of the current state of the technology of graph data processing in the SAP HANA database. We have sketched use cases to illustrate the need for pattern matching in graphs and described the challenges and open questions we face. En route to enabling large graph analytics within the SAP HANA database as part of our research project SynopSys, we have outlined how this technique can form the basis of graph summarization functionality. Finally, we have presented the basic operations needed for effectively summarizing large graphs, captured in the form of summarization templates to be instantiated and applied by the user.

In the future we would like to investigate whether and how the summarization functionality can be extended to support arbitrary general-purpose graph transformations and how those can be efficiently implemented in the context of the SAP HANA database.

7. ACKNOWLEDGEMENTS

8. REFERENCES

[1] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proc. ACM SIGMOD*, pages 1247–1250, New York, NY, USA, 2008. ACM.

[2] C. Bornhövd, R. Kubis, W. Lehner, H. Voigt, and H. Werner. Flexible Information Management, Exploration, and Analysis in SAP HANA. In *Proc. International Conference on Data Technologies and Applications*, pages 15–28. SciTePress, 2012.

[3] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu. Graph OLAP: Towards Online Analytical Processing on Graphs. In *Proc. 8th International Conference on Data Mining*, pages 103–112, Pisa, Italy, Dec. 2008. IEEE.

[4] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty Years of Graph Matching in Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

[5] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding Regular Expressions to Graph Reachability and Pattern Queries. In *Proc. 27th ICDE*, pages 39–50, Hannover, Germany, Apr. 2011. IEEE.

[6] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[7] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Anal. Appl.*, 13(1):113–129, Jan. 2010.

[8] M. Karpinski and W. Rytter. *Fast Parallel Algorithms for Graph Matching Problems*. Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, May 1998.

[9] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *Proc. ACM SIGMOD*, pages 419–432, New York, NY, USA, 2008. ACM.

[10] M. A. Rodriguez and P. Neubauer. Constructions from Dots and Lines. *Bull. American Society for Information Science and Technology*, 36(6):35–41, 2010.

[11] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The Graph Story of the SAP HANA Database. In *BTW*, LNI, pages 403–420. GI, 2013.

[12] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proc. ACM SIGMOD*, pages 731–742, New York, NY, USA, 2012. ACM.

[13] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient Aggregation for Graph Summarization Categories and Subject Descriptors. In *Proc. ACM SIGMOD*, pages 567–580, Vancouver, BC, Canada, 2008. ACM.

[14] N. Zhang, Y. Tian, and J. M. Patel. Discovery-Driven Graph Summarization. In *Proc. 26th ICDE*, pages 880–891, Long Beach, CA, USA, 2010. IEEE.

[15] P. Zhao, X. Li, D. Xin, and J. Han. Graph Cube: On Warehousing and OLAP Multidimensional Networks. In *Proc. ACM SIGMOD*, pages 853–864, Athens, Greece, 2011. ACM.