

The Graph Story of the SAP HANA Database

Michael Rudolf¹, Marcus Paradies¹, Christof Bornhövd², and Wolfgang Lehner¹

¹SAP AG; Dietmar-Hopp-Allee 16; Walldorf, Germany

²SAP Labs, LLC; 3412 Hillview Avenue; Palo Alto, CA, 94304

eMail: {michael.rudolf01, m.paradies, christof.bornhoevd, wolfgang.lehner}@sap.com

Abstract: Many traditional and new business applications work with inherently graph-structured data and therefore benefit from graph abstractions and operations provided in the data management layer. The property graph data model not only offers schema flexibility but also permits managing and processing data and metadata jointly. By having typical graph operations implemented directly in the database engine and exposing them both in the form of an intuitive programming interface and a declarative language, complex business application logic can be expressed more easily and executed very efficiently. In this paper we describe our ongoing work to extend the SAP HANA database with built-in graph data support. We see this as a next step on the way to provide an efficient and intuitive data management platform for modern business applications with SAP HANA.

1 Introduction

Traditional business applications, such as Supply Chain Management, Product Batch Traceability, Product Lifecycle Management, or Transportation and Delivery, benefit greatly from a direct and efficient representation of the underlying information as data graphs. But also not so traditional ones, such as Social Media Analysis for Targeted Advertising and Consumer Sentiment Analysis, Context-aware Search, or Intangible and Social Asset Management can immensely profit from such capabilities.

These applications take advantage of an underlying graph data model and the implementation of core graph operations directly in the data management layer in two fundamental ways. First, a graph-like representation provides a natural and intuitive format for the underlying data, which leads to simpler application designs and lower development cost. Second, the availability of graph-specific operators directly in the underlying database engine as the means to process and analyze the data allows a very direct mapping of core business functions and in turn to significantly better response times and scalability to very large data graphs.

When we refer to data graphs in this paper, we mean a full-fledged property graph model rather than a subject-predicate-object model, as used by most triple stores, or a tailored relational schema, for example in the form of a vertical schema, to generically store vertices and edges of a data graph.

A property graph [RN10] is a directed multi graph consisting of a finite (and mutable) set

Category		Rating			part of	
ID	Name	ID	Value	Product	C_ID1	C_ID2
1	Books	1	4	1	2	1
2	Literature & Fiction	2	3	2	4	3
3	Movies & TV	3	4	3		
4	Movies	4	5	3		

Product				in	
ID	Title	Year	Type	P_ID	C_ID
1	Romeo and Juliet	2012	Book	1	2
2	Romeo and Juliet	1997	DVD	2	4
3	Shakespeare in Love	1999	DVD	3	4

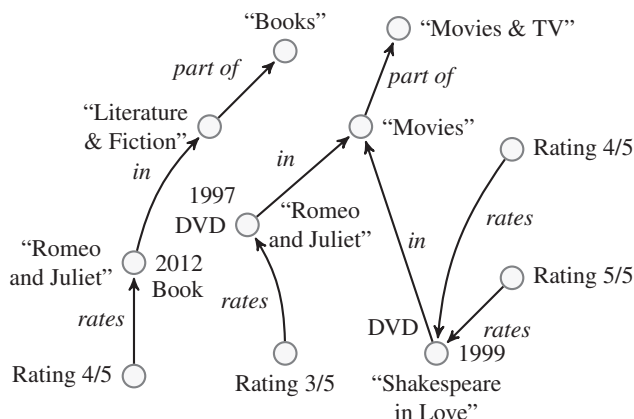


Figure 1: Example data expressed in the relational and the property graph data model

of vertices (nodes) and edges (arcs). Both, vertices and edges can have assigned properties (attributes) which can be understood as simple name-value pairs. A dedicated property can serve as a unique identifier for vertices and edges. In addition, a type property can be used to represent the semantic type of the respective vertex or edge. Properties of vertices and edges are not necessarily determined by the assigned type and can therefore vary between vertices or edges of the same type. Vertices can be connected via different edges as long as they have different types or identifiers.

Figure 1 shows a very small example data set both expressed in the relational model and the property graph model, which could be the basis for a Targeted Advertisement application. Customers can rate products, which are organized in categories. If, additionally, the relationships between customers, the products they bought, and their ratings are stored, the application can easily recommend products that might be of interest to the customer based on what other customers bought and rated.

The property graph model provides the following key characteristics, which distinguish it, in particular, from the classical relational data model.

- **Relationships as First Class Citizens.** With the property graph model relationships between entities are promoted to first class citizens of the model with unique identity, semantic type, and possibly additional attributes. The relational model focuses on the representation of entities, their attributes and relational consistency constraints between them and requires the use of link tables to represent n-to-m relationships or additional attributes of relationships. In contrast, the concept of an edge provides an explicit and flexible way to represent interrelationships between entities which is essential if relationships between entities are very important or even in the center of the processing and analysis of the data.
- **Increased Schema Flexibility.** In a property graph edges are specified at the instance and not at the class level, i.e., they relate two specific vertices, and vertices of the same semantic types can be related via different types of edges. Similarly, properties of edges and vertices are not necessarily determined by the semantic type of the respective edge or vertex, which means that edges or vertices of the same semantic type can have assigned different sets of properties.

With this the schema of a data graph does not have to be predefined in the form of a rigid schema that would be cumbersome and expensive to modify but rather evolves as new vertices are created, new properties are added, and as new edges between vertices are established.

- **No Strict Separation between Data and Metadata.** Vertices and edges in a graph can have assigned semantic types to indicate their intended meaning. These types can be naturally represented as a tree (taxonomy) or graph (ontology) themselves. This allows their retrieval and processing as either type definitions, i.e., metadata, or (possibly in combination with other vertices) as data. By allowing to treat and use type definitions as regular vertices we can give up a strict and for some applications artificial separation of data from metadata.

For example, in the context of context-aware search a given search request can be extended or refined not only by considering related content (i.e., vertices that are related to vertices directly referred to by the request) but also related concepts or terms (i.e., vertices that are part of the underlying type system used in the search).

In recent years, another graph model has gained a lot of popularity: the Resource Description Framework (RDF [CK04]). At its core is the concept that statements about resources can be made in the form of triples consisting of a subject, a predicate and an object. The subject and the predicate are always resources, whereas the object of such a statement can be either a resource or a literal. This simple concept, with almost no further constraints, offers an extremely flexible way of representing information – and hence heavily depends on what conventions individual applications use to encode and decode RDF data. All triples of a dataset form a labeled graph, which represents a network of values. An entity is decomposed into a set of statements and application logic is required to reassemble them

upon retrieval. In contrast, the property graph model provides intrinsic support for entities by permitting vertices and edges to be attributed. RDF therefore does not offer inherent means to represent an entity as a unit and requires applications to provide this semantics.

The use of a dedicated set of built-in core graph operators offers the following key performance and scalability benefits.

- **Allow Efficient Execution of Typical Graph Operations.** An implementation of graph operators directly in the database engine allows the optimization of typical graph operations like single or multi-step graph traversal, inclusive or exclusive selection of vertices or edges, or to find the shortest or all paths between vertices. Such optimizations are not possible in for example relational database systems since the basic operators are unaware of concepts like vertex and edge. In particular, depending on the physical representation of graph data in the system vertices can act like indexes for their associated vertices which allow the performance of graph traversals to be independent of the size of the overall data graph. In contrast, the realization of traversal steps in a relational database system requires join operators between tables whereby the execution time typically depends on the size of the involved tables.
- **Provide Support for Graph Operations Difficult to Express in SQL.** Similarly, the direct implementation of graph-specific operations in the database allows the support of operations that otherwise are very hard or even impossible to express for example in standard SQL. Relational databases are good at straight joins but are not good or are unable to execute joins of unpredicted length that are required to implement transitive closure calculations in graph traversals. Another example is sub-graph pattern matching, which is very difficult to express in general with the means of standard SQL.

In this paper we describe how we extended the SAP HANA [FCP⁺12] database with native graph data support. In the following section we present different classes of business applications and how they benefit from a dedicated graph support in the database engine. The key components of our technology in the context of the SAP HANA database architecture are introduced in Section 3. Section 4 details the graph data model and our declarative query and manipulation language WIPE, and Section 5 presents the underlying graph abstraction layer and the graph function library. In Section 6 we exemplarily evaluate the performance of our approach compared to the traditional SQL-based implementation. Finally, Section 7 summarizes the presented work.

2 Use Cases

In the following paragraphs we illustrate the use of the property graph model and a dedicated graph database management system by different business applications.

Transportation and Logistics. Transportation and logistics are important components of supply chain management. Every company that sells goods relies on materials or products being transported via motor carrier, rail, air or sea transport from one location to another. Therefore, accurate representation and management, as well as visibility into their transportation options and logistics processes are vital to businesses. A typical scenario would include both inbound (procurement) and outbound (shipping) orders to be managed by a transportation management module which can suggest different routing options. These options are evaluated and analyzed with the help of a transportation provider analysis module to select the best route and provider based on cost, lead-time, number of stops, risk, or transportation mode. Once the best solution has been selected, the system typically generates electronic tendering and allows to track the execution of the shipment with the selected carrier, and later supports freight audit and payment. A graph data model supports a flexible and accurate representation of the underlying transportation network. Efficient graph operations enable the fast execution of compute intensive graph operations like identification of shortest or cheapest paths or multi-stop transportation routes.

Product Batch Traceability. End-to-end product traceability is key in global manufacturing to monitor product quality and to allow efficient product recall handling to improve customer safety and satisfaction. It supports complete product batch tracing of all materials purchased, consumed, manufactured, and distributed in the supply and distribution network of a company. Backward traceability allows companies to identify and investigate problems in their manufacturing process or plants as well as in their supply chain. Forward traceability, on the other hand, allows to respond fast to encountered problems to comply with legal reporting timelines, and to minimize cost and corporate risk exposure. A graph data model allows for a direct and natural representation of the batch relation network. Graph processing capabilities in the data management layer are a prerequisite to guarantee fast root cause analysis and to enable timely product recalls and withdrawals as required by law in many industries.

Targeted Advertisement. The goal of targeted advertising is to deliver the most relevant advertisement to target customers to increase the conversion rate of customers who see the advertisements into actual buyers. Decisions of which advertisements to send to which customers can be done based on user profile, behavior, and social context. This matching process includes, in particular, customer segmentation or the creation of personas (like “sports car fan”) based on social and interest graphs that describe who the respective user knows or follows and what the user has shown interest in or likes. This information can be derived from publicly available sources that people volunteer or captured by opt-in applications, like Facebook interests, product reviews or blogs, or what they tweet or re-tweet. A data graph model and data graph processing capabilities support the flexible combination of data from the multitude of relevant sources and allows an efficient representation and management of large and frequently changing social graphs. Fast graph analytics operations on this data are a prerequisite to enable large-scale real-time targeted advertisement.

Bill of Materials. Complex products are usually described with the help of a hierarchical decomposition into parts, sub-components, intermediate assemblies, sub-assemblies and raw materials together with the quantities of each, a so-called bill of materials (BOM [ISO12]). Manufacturing industries, such as the automotive and aeronautics sectors, use BOMs to plan the assembly processes. Two important operations on BOMs are linking pieces to assemblies (“implosion”) and breaking apart each assembly into its component parts (“explosion”). Since hierarchies are directed acyclic graphs with a single start node, applications working with BOMs can benefit from a natural graph representation and fast graph processing.

3 Architecture Overview

The SAP HANA database [SFL⁺12] is a memory-centric database. It leverages the capabilities of modern hardware, in particular very large amounts of main memory, multi-core CPUs, and SSD storage, to increase the performance of analytical and transactional applications. Multiple database instances may be distributed across multiple servers to achieve good scalability in terms of data volume and number of application requests. The SAP HANA database provides the high-performance data storage and processing engine within the SAP HANA Appliance product.

The Active Information Store (AIS) project aims at providing a platform for efficiently managing, integrating, and analyzing structured, semi-structured, and unstructured information. It was originally started as an extension to SAP’s new in-memory database technology [BKL⁺12] and has now evolved into a part of it. By tightly integrating the graph processing capabilities into the SAP HANA database rather than providing a separate system layer on top of it, we can directly leverage the fast infrastructure and efficiently combine data from the relational engine and the text engine with graph data in one database query. We tried to build on the existing database engine infrastructure for the new graph capabilities by re-using or extending existing physical data structures and query execution capabilities as much as possible. This helped to keep complexity manageable, both in terms of the number of new system components and in terms of new concepts introduced.

Figure 2 shows the integration of the different AIS components in the architecture of the SAP HANA database. WIPE is the declarative query and manipulation language of the AIS and uses a property graph model extended with semantic information. Database clients can pass in WIPE statements via ODBC or JDBC. Both the language and the AIS data model are described in more detail in the following section. For the execution of WIPE relational operations are re-used where applicable. The basic graph abstractions, operations, and the library of built-in graph processing functionality used to realize the non-relational aspects of WIPE are presented in Section 5. Complex processing tasks are encapsulated as operators, which are implemented on top of the in-memory column store primitives with very little overhead and therefore profit from the efficient information representation and processing of compression and hardware-optimized instructions, respectively.

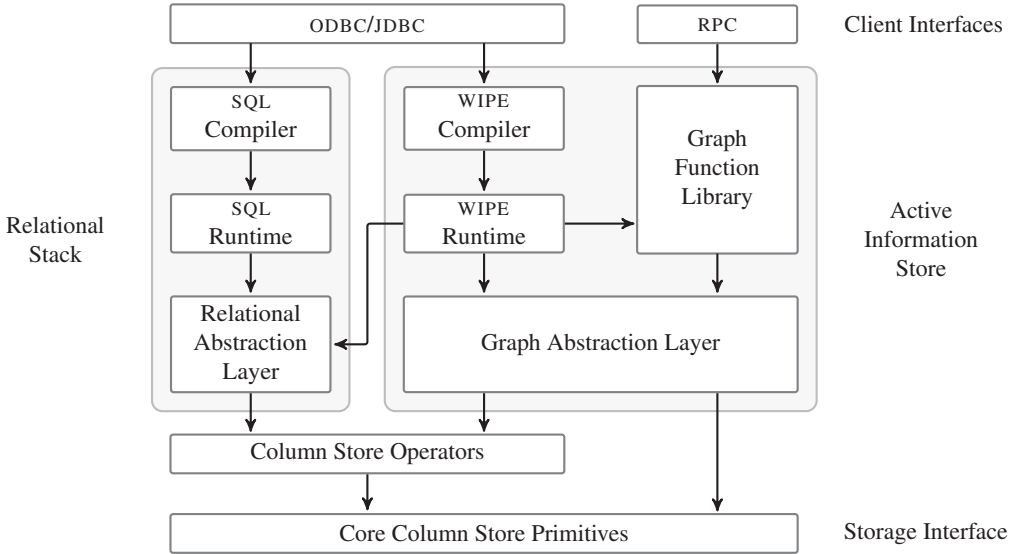


Figure 2: Integration of the Active Information Store in the SAP HANA database

4 The Active Information Store Runtime and WIPE

The AIS data model has been designed such that it permits the uniform handling and combination of structured, irregularly structured, and unstructured data. It extends the property graph model [RN10] by adding concepts and mechanisms to represent and manage semantic types (called *Terms*), which are part of the graph and can form hierarchies. Terms are used for nominal typing: they do not enforce structural constraints, such as the properties a vertex (called *Info Items*) must expose. Info Items that have assigned the same semantic type may, and generally do, have different sets of properties, except for a unique identifier that each Info Item must have. Info Items and Terms are organized in *workspaces*, which establish a scope for visibility and access control. Data querying and manipulation are always performed within a single workspace and user privileges are managed on a per-workspace basis. Finally, Terms can be grouped in domain-specific *taxonomies*.

A pair of Info Items can be connected by directed associations, which are labeled with a Term indicating their semantic type and can also carry attributes. As for Info Items, the number and type of these attributes is not determined by the semantic type of the association. The same pair of Info Items can be related via multiple associations of different types. Figure 3 visualizes the relationships between these concepts as a UML diagram.

WIPE is the data manipulation and query language built on top of the graph functionality in the SAP HANA database. “WIPE” stands for “Weakly-structured Information Processing and Exploration”. It combines support for graph traversal and manipulation with BI-like data aggregation. The language allows the declaration of multiple insert, update, delete, and query operations in one complex statement. In particular, in a single WIPE statement

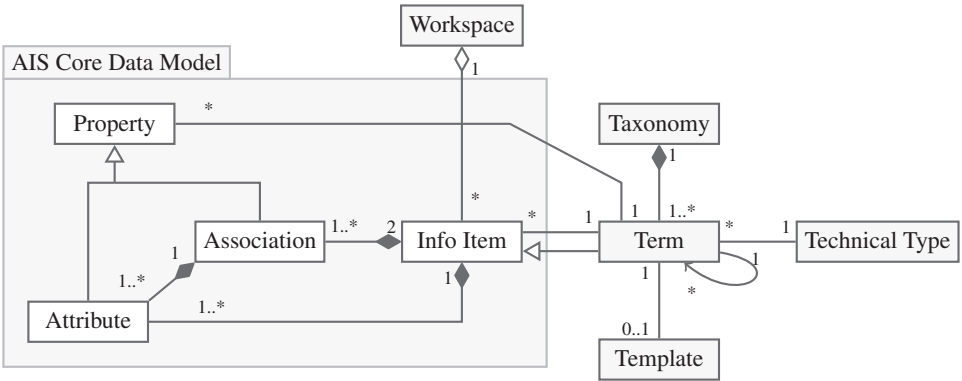


Figure 3: UML [Obj11] class diagram of the Active Information Store data model

multiple named query result sets can be declared and are computed as one logical unit of work in a single request-response roundtrip.

The WIPE language has been designed with several goals in mind [BKL⁺12]. One is the ability to deal with flexible data schemas and with data coming from different sources. Not maintaining metadata separately from the actual data, the AIS permits introspecting and changing type information in an intuitive way. WIPE offers mass data operations for adding, modifying, and removing attributes and associations, thereby enabling a stepwise integration and combination of heterogeneous data. While navigational queries can be used for data exploration, WIPE also supports information extraction with the help of grouping and aggregation functionality. A rich set of numerical and string manipulation functions helps in implementing analytical tasks.

Like the other domain-specific languages provided by the SAP HANA database, WIPE is embedded in a transaction context. Therefore, the system supports the concurrent execution of multiple WIPE statements guaranteeing atomicity, consistency, durability, and the required isolation.

Listing 1 shows an example WIPE query on the data set presented in Figure 1 returning all books that have received the highest rating at least once. In the first step the graph to operate on is chosen. Thereafter, the set containing the single Info Item representing the “Books” category is assigned to a local name for later use. The third line computes the transitive closure over the “partOf” associations starting from the set specified in the previous step and thereby matches all subcategories of the “Books” category. From there, all Info Items connected via “in” associations are selected and assigned to another local name. Finally, a result is declared that consists of the Info Items matched by the existence quantification, which accepts all Info Items having a “rated” association with a “rating” attribute of value 5.

Listing 1: Example WIPE statement

```
//Tell WIPE which graph data to consult
USE WORKSPACE uri:AIS;

//Save a reference to the "Books" category in a local variable
$booksCategory = { uri:books };

//Traverse to all products in the "Books" category
//The transitive closure (1, *) reaches all arbitrarily nested categories
$allBooks = $booksCategory<-uri:partOf(1, *)<-uri:in;

//Return the books with at least one highest rating using a quantification
RESULT uri:bestBooks FROM $b : $allBooks WITH ANY $b<-uri:rated@uri:rating = 5;
```

5 The Graph Abstraction Layer and Function Library

Modern business applications demand support for easy-to-use interfaces to store, modify and query data graphs inside the database management system. The graph abstraction layer in the SAP HANA database provides an imperative approach to interact with graph data stored in the database by exposing graph concepts, such as vertices and edges, directly to the application developer. Its programming interface, called Graph API, can be used by the application layer via remote procedure calls.

The graph abstraction layer is implemented on top of the low-level execution engine of the column store in the SAP HANA database. It abstracts from the actual implementation of the storage of the graph, which sits on top of the column store and provides efficient access to the vertices and edges of the graph. The programming interface has been designed in such a way, that it seamlessly integrates with popular programming paradigms and frameworks, in particular the Standard Template Library (STL, [Jos99]).

Figure 4 shows the basic concepts of the Graph API and their relationships as a simplified UML class diagram. Method and template parameters as well as namespaces have been omitted for the sake of legibility.

Beside basic retrieval and manipulation functions, the SAP HANA database provides a set of built-in graph operators for application-critical operations. All graph operators interact directly with the column store engine to execute very efficiently and in a highly optimized manner. Well-known and often used graph operators, such as breadth-first and depth-first traversal algorithms, are implemented and can be configured and used via the Graph API. Beside the imperative interface, all graph operators can also be used in a relational execution plan as custom operators.

In the following, we summarize the key functions and methods that are being exposed to the application developer.

- **Creation and deletion of graphs.** The graph abstraction layer allows to create a new graph by specifying minimal database schema information, such as an edge store name, a vertex store name, and a vertex identifier description. This information is

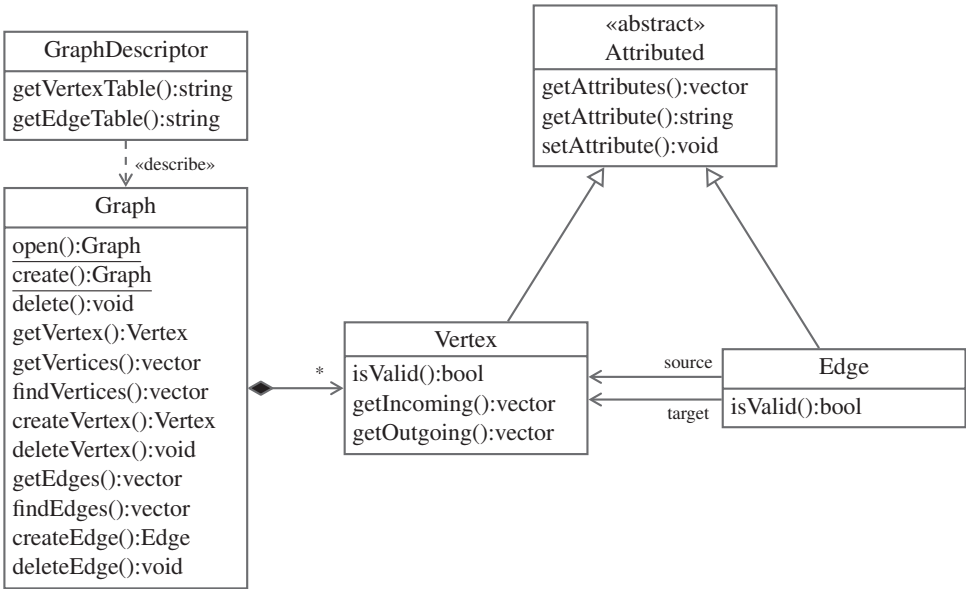


Figure 4: UML [Obj11] class diagram of the Graph API

encapsulated in a graph descriptor object. The creation of a graph is atomic, i.e., if an error occurs during the creation of the graph store object, an exception is thrown and the creation of the graph is aborted.

A graph can be deleted by specifying the corresponding graph descriptor. The deletion process removes the vertex and the edge store from the database system and invalidates the corresponding graph object in the graph abstraction layer.

- **Access to existing graphs.** An existing graph can be opened by specifying the edge store and the vertex store of the graph. All missing information, such as the edge description, are automatically collected from the store metadata. If the graph does not exist in the database management system, an exception is thrown.
- **Addition, deletion, and modification of vertices and edges.** Vertices and edges are represented by light-weight objects that act as an abstract representative of the object stored in the database. The objects in the graph abstraction layer only point to the actual data of the object and hold the internal state during processing. If the graph abstraction layer executes a function call that requests data from the objects, it gets loaded on demand.
- **Retrieval of sets of vertices based on a set of vertex attributes.** Vertices can have assigned multiple properties. These properties can be used to filter vertices, for example, in graph traversal operations.
- **Retrieval of sets of edges based on a set of edges attributes.** Similarly, properties on edges can be leveraged to select possible paths to follow in a graph traversal.

Listing 2: Example pseudo code showing how old ratings can be purged from the data set

```
//Open an existing graph specified with a description object
GraphDescriptor descriptor("VERTEX_TABLE", "EDGE_TABLE");
Graph graph = Graph::open(descriptor);

//Find a specific vertex assuming that the product title is the unique identifier
Vertex vertex = graph.getVertex("Shakespeare_in_Love");

//Iterate over all incoming edges (those from ratings)
for (Edge incoming : vertex.getIncoming()) {
    Vertex source = incoming.getSource();

    //Find old ratings and delete them
    if (source.getAttribute("created") < threshold) {
        //All incoming and outgoing edges will be removed as well
        graph.deleteVertex(source);
    }
}
```

- **Configurable and extensible graph traversals.** Efficient support for configurable and extensible graph traversals on large graphs is a core asset for business applications to be able to implement customized graph algorithms on top of the Graph API. The SAP HANA database provides native and extensive support for traversals on large graphs on the basis of a graph traversal operator implemented directly in the database kernel.

The operator traverses the graph in a breadth-first manner and can be extended by a custom visitor object with user-defined actions that are triggered during defined execution points. At any execution point, the user can operate on the working set of vertices that have been discovered during the last iteration. Currently, only non-modifying operations on the working set of vertices are allowed to not change the structure of the graph during the traversal.

Listing 2 illustrates the use of these functions in a C++-like pseudo code. Header file includes, qualified identifiers, exception handling, and STL iterators have been deliberately omitted from the example for the sake of simplicity. In the first line a graph descriptor object is created; it consists of the names of the vertex and edge tables to work with. This is passed to the static open method to obtain a handle to the graph in the next line. Thereafter, a handle to the vertex with the identifier “Shakespeare in Love” is retrieved. The for-loop then iterates over all incoming edges of that vertex and for each edge obtains a handle to the source vertex. The value of the “created” attribute of that vertex is compared to some threshold and if it is less, the vertex is removed. All edges connecting the vertex are removed automatically as well.

The graph abstraction layer has to be used from within a transaction context. All modifying and non-modifying operations on the graph data are then guaranteed to be compliant to the ACID properties offered by the SAP HANA database. To achieve this goal, multi version concurrency control (MVCC) is used internally.

Many applications using the graph abstraction layer are built around well-known graph algorithms, which are often only slightly adapted to suit their application-specific needs. If each application bundles its own version of the algorithms it uses, a lot of code will be duplicated. Furthermore, not every application always provides an implementation that is optimal with regards to the data structures the graph abstraction layer offers.

To avoid these problems, the SAP HANA database also contains a graph function library built on top of the core graph operations, which offers parameterizable implementations of often-used graph algorithms specifically optimized for the graph abstraction layer. Applications can reuse these algorithms, which are well-tested, to improve their stability and thereby reduce their development costs.

For example, the graph function library currently contains implementations of algorithms for finding shortest paths, vertex covers, and (strongly) connected components, amongst others. As new applications are built in the future, more algorithms will be supported.

6 Evaluation

In this section we present the first experimental analysis of the integration of the AIS and its query and manipulation language WIPE into the SAP HANA database. In our experiments we show that the AIS is an advantageous approach for supporting graph processing and handling of large data graphs directly within the SAP HANA database. Beside the comparison of WIPE against a pure relational solution for graph traversals using SQL, we also show the scalability of the AIS engine to handle very large graphs efficiently.

6.1 Setup and Methodology

All experiments are conducted on a single server machine running SUSE Linux Enterprise Server 11 (64-bit) with Intel Xeon X5650, 6 cores, 12 hardware threads running at 2.67 GHz, 32 KB L1 data cache, 32 KB L1 instruction cache, 256 KB L2 cache and 12 MB L3 cache shared and 24 GB RAM.

We generated five graph data sets that represent multi-relational, directed property graphs using the R-MAT graph generator [CZF04]. Since the R-MAT generator does not support multi-relational graphs, we enhanced the graph data generation process and labeled edges according to collected edge type distribution statistics from a set of examined real-world batch traceability data sets. We distributed the edge labels randomly across all available edges whereby we labeled edges with types a , b , and c . The selectivities for the edge types are 60 % for type a , 25 % for type b , and 15 % for type c , respectively. Table 1 lists all generated data sets as well as graph statistics that characterize the graph topology. For the remainder of the experimental analysis we will refer to the data sets by their *Data Set ID*.

Further, we use a real-world product co-purchasing graph data set that has been prepared and analyzed by Leskovec et al. [LAH07]. Figure 1 shows an example derived from this data

Table 1: Statistical information for generated graph data sets $G1$ – $G5$.

<i>Data Set ID</i>	<i># Vertices</i>	<i># Edges</i>	<i>Avg. Vertex Out-Degree</i>	<i>Max. Vertex Out-Degree</i>
G1	524 306	4 989 244	22.4	453
G2	1 048 581	9 985 178	25.3	5 621
G3	2 097 122	19 979 348	27.2	9 865
G4	4 192 893	29 983 311	28.1	14 867
G5	15 814 630	39 996 191	28.3	23 546

set. The co-purchasing data set models products, users, and product categories as vertices. Ratings, relationships between product categories, and product category memberships are modeled as edges. Table 2 depicts the most important graph statistics that can be used to describe the graph topology of the data set. Since the co-purchasing graph is a multi-relational graph with highly varying subgraph topologies, we gathered the statistical information for each subgraph separately. The three subgraphs are described by the three edge type labels that exist in the data set. The subgraph *User-Product* contains all users and products as vertices and shows the relationship between these two vertex types via ratings. The subgraph *Product-Category* describes the membership of certain products to product categories. The third subgraph describes the category hierarchy of the co-purchasing graph.

Please note that we do not show the relationships between products, which are also known as co-purchasing characteristics here for the sake of simplicity. Additionally, it is worth to mention that vertices in the data sets contribute to multiple subgraphs. Because of this, the summation of number of vertices from all subgraphs is larger than the actual number of vertices in the complete graph.

We loaded the data sets into two tables, one for storing the vertices and one for storing the edges of the graph. Thereby, each vertex is represented as a record in the vertex table *VERTICES* and each edge is represented as a record in the edge table *EDGES*. Each edge record comprises a tuple of vertex identifiers specifying source and target vertex as well as an edge type label and a set of additional application-specific attributes.

Listings 3 and 4 depict a qualitative comparison between a WIPE query performing a graph traversal and the equivalent SQL query that heavily relies on chained self-joins. Please note that both queries are based on the same physical data layout (vertex and edge table). While the SQL query addresses the storage of vertices and edges explicitly via table name and schema name, a WIPE query only needs to specify a workspace identifier. The table names for vertices and edges are preconfigured in the database configuration and do not need to be specified during query execution.

Both queries perform a breadth-first traversal starting from vertex A , following edges with

Table 2: Statistical information for co-purchasing graph data set $A1$.

<i>Subgraph</i>	<i># Vertices</i>	<i># Edges</i>	<i>Avg. Vertex Out-Degree</i>	<i>Max. Vertex Out-Degree</i>	<i>Avg. Vertex In-Degree</i>	<i>Max. Vertex In-Degree</i>
User-Product	832 574	7 781 990	5.4	124	6.3	237
Product-Category	588 354	2 509 422	3.1	13.3	53.4	23 121
Category-Category	23 647	7 263	2.1	78	2.1	78

Listing 3: SQL statement

```

SELECT DISTINCT V.id
  FROM AIS.EDGES AS A, AIS.EDGES AS B,
       AIS.EDGES AS C, AIS.EDGES AS D,
       AIS.VERTICES AS V
 WHERE A.source = "A"
       AND D.target = V.id
       AND A.type = "a"
       AND A.target = B.source
       AND B.target = C.source
       AND C.target = D.source

```

Listing 4: WIPE statement

```

USE WORKSPACE uri:AIS;
$root = { uri:A };
$t = $root->uri:a(4,4);
RESULT uri:res FROM $t;

```

type label a , and finally return all vertices with a distance of exactly 4 from the start vertex. The SQL query executes an initial filter expression on the edge type label a to filter out all non-matching edges. Next, the start vertex is selected and the corresponding neighbor vertices are used to start the chained self-joins. For $n = 4$ traversal steps, $n - 1$ self-joins have to be performed.

In contrast, the WIPE query selects the start vertex A and binds the result to a temporary variable $\$root$. Next, a traversal expression evaluates a breadth-first traversal from the start vertex over edges with edge type label a and performs 4 traversal steps. Finally, the output of the traversal expression is returned to the user. For more examples of WIPE queries, we refer to [BKL⁺12].

Both queries are functionally equivalent and return the same set of vertices. However, a WIPE query provides a much more intuitive and compact interface to directly interact with a graph stored in the database.

6.2 Experiments

The first experiment shows a comparison of the scalability of SQL queries performing graph traversals versus their corresponding WIPE query counterpart and is depicted in Figures 5. For the experiment in Figure 5, we varied the number of path steps between 1 and 10 and randomly chose a start vertex for each graph traversal run. We ran each query 10 times and averaged the execution time after removing the best and the worst performing query. We restricted the number of path steps to 10 since all graphs exhibit the small-world property that can be found in many real-world data graphs [Mil67]. In this experiment, we showcase how the execution time evolves when more path steps are to be performed during the graph traversal. To illustrate the behavior, we use the graph traversal over one path step as baseline and relate subsequent queries traversing over multiple path steps to this one-step graph traversal. Consequently, the execution time of queries over multiple path steps is always a multiple of the execution time of a one-step traversal.

Figure 5 shows the gained relative execution factor when comparing a one-step traversal as baseline against a multi-step traversal in subsequent queries. The relative execution factor is

plotted with logarithmic scale to the base 2. The green-colored plot with the circle markers shows the relative execution factor for the SQL query traversal with respect to the one-step traversal. The plot reflects the poor behavior for multi-step traversals of SQL queries with a number of path steps larger than 2. The blue-colored plot with square markers shows the relative execution factor of the WIPE query with respect to the one-step traversal baseline. The relative execution factor of WIPE grows much slower than the relative execution factor of the equivalent SQL statement. Thereby, a linear and slow rise is better in terms of scalability to the number of path steps to perform. The WIPE implementation shows for data set $G4$ a maximum relative execution factor of 3 and the SQL implementation shows a maximum relative execution factor of 91.

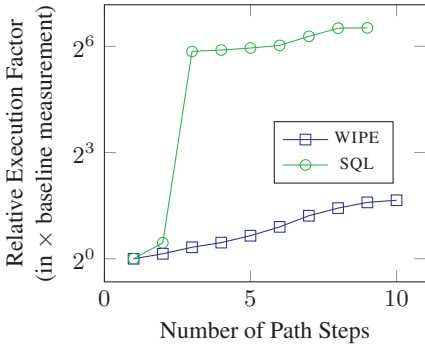


Figure 5: Scalability of SQL and WIPE for $G4$

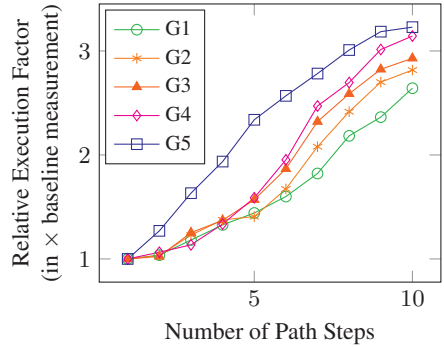


Figure 6: Scalability of WIPE

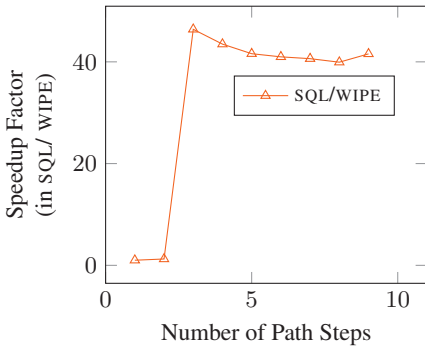


Figure 7: SQL and WIPE compared for $G4$

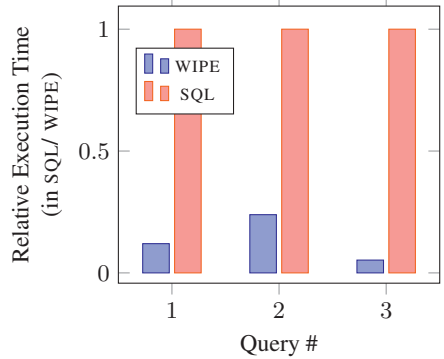


Figure 8: SQL and WIPE compared for $A1$

The next experiment illustrates the scalability of WIPE and the AIS engine with respect to growing graph data sets. Figure 6 depicts the results for given data sets $G1-G5$. As for the first experiment, we varied the number of path steps between 1 and 10 and chose a start vertex randomly for each graph traversal run. We used the one-step traversal as baseline and related all subsequent multi-step traversal queries to it.

Table 3: Real-world queries from the co-purchasing domain.

<i>Query ID</i>	<i>Description</i>
1	Select all super categories from product category "Movies."
2	Find all users which rated "Hamlet" and "Romeo and Juliet" with 5 stars.
3	Find all books that are in category "Fiction" or sub categories and have been released to DVD.

The results indicate a good scalability of WIPE and the AIS engine with respect to increasing data set sizes and varying number of path steps. For all examined data sets, we found maximum relative execution factors between 2.6 and 3.2. Thereby, we saw only marginal variations between the different data sets which are mainly caused by differences in the underlying graph topologies. Independent from the graph topology present in the examined data set, the traversal operator scales very well with respect to large graph data sets as well as increasing number of path steps to perform.

Figures 7 and 8 depict the speedup in execution time of WIPE queries compared to an equivalent SQL query. Here, we relate the execution time of the SQL query to the execution time of the equivalent WIPE query.

Figure 7 illustrates the speedup factor between SQL and WIPE for data set $G4$. We obtain a maximum speedup factor of 46 when comparing and relating the execution times of SQL queries to their equivalent WIPE query performing the same number of path steps.

For graph traversals with a path length of one or two, the gained speedup for WIPE was between 1.03 and 1.24. For one-step traversals, the SQL query can directly return the neighbors of the given start vertex without the need to perform expensive self-joins. For two-step traversals, a self-join has to be performed to retrieve vertices with path distance 2 from the root vertex. In general, for graph traversals with a larger number of path steps, a built-in operator clearly outperforms a functionally equivalent SQL query on average by a factor of 30 in our experiments.

For the SQL variant of the graph traversal, the execution time is highly dependent on the number of vertices that are being discovered at each level of the graph traversal algorithm. When a large fraction of all reachable vertices has been discovered, the speedup between SQL and WIPE will decrease slightly as can be seen for graph traversals with three or more path steps. However, WIPE still performs about 30 times as fast as the SQL implementation of the graph traversal algorithm.

In Figure 8, we compare the relative execution time for three real-world queries from the co-purchasing domain. The queries are described in Table 3. The figure shows a relative execution factor of WIPE against the equivalent SQL implementation between a factor of about 4 and 20. The most beneficial query is query 3 since it involves the most complex traversal execution and cannot be handled efficiently by the relational engine. If there are only a limited number of path steps to perform (as for query 2), the results are similar to those obtained in the other experiments. Since the execution time of the SQL query is dependent on the number of self-joins to perform (the number of path steps to traverse), the relative execution factor is lower for simple graph traversals.

7 Summary

A variety of business applications work with inherently graph-structured data and therefore profit from a data management layer providing an optimized representation of graph data structures and operators optimized for this representation. Within this paper, we outlined the SAP HANA Active Information Store project with its query and manipulation language WIPE and a property graph model as the underlying data representation. The general goal of the project on the one hand is to provide built-in graph-processing support leveraging the performance and scalability of the SAP HANA main-memory database engine. On the other hand, the project aims at providing a powerful and intuitive foundation for the development of modern business applications.

Therefore, we first motivated the project by presenting different scenarios ranging from classical graph-processing in social network analytics to the domain of supply chain management and product batch traceability. We then briefly touched the overall architecture of SAP HANA with respect to the functionality related to graph processing. The original design of SAP HANA allows to operate on multiple language stacks in parallel with local compilers exploiting a common set of low-level column-store primitives running in a scalable distributed data-flow processing environment.

In the second part of the paper, we described the graph abstraction layer using a practical example and presented results of extensive experiments. The experiments were conducted on several synthetic graph data sets to show the effects of different topologies with respect to certain query scenarios. Overall, the optimized graph-processing functionality performs significantly better than the comparable SQL representation using only relational operators. The specific support for graph-structured data sets and the matured distributed query processing engine of SAP HANA provides a superb solution for complex graph query expressions and very large graph data sets.

Acknowledgement

We would like to express our gratitude to Hannes Voigt for many inspiring discussions and for his feedback on earlier versions of this paper. We also thank our fellow Ph.D. students in Walldorf for their encouragement and support.

References

- [BKL⁺12] Christof Bornhövd, Robert Kubis, Wolfgang Lehner, Hannes Voigt, and Horst Werner. Flexible Information Management, Exploration, and Analysis in SAP HANA. In Markus Helfert, Chiara Francalanci, and Joaquim Filipe, editors, *Proceedings of the International Conference on Data Technologies and Applications*, pages 15–28. SciTePress, 2012.
- [CK04] Jeremy J. Carroll and Graham Klyne. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C recommendation, W3C, February 2004.

- [CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining, SDM '04*, pages 442–446. Society for Industrial and Applied Mathematics, 2004.
- [FCP⁺12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.*, 40(4):45–51, January 2012.
- [ISO12] ISO. *Technical product documentation – Vocabulary – Terms relating to technical drawings, product definition and related documentation (ISO 10209:2012)*. International Organization for Standardization, Geneva, Switzerland, 2012.
- [Jos99] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 1999.
- [LAH07] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. The Dynamics of Viral Marketing. *ACM Trans. Web*, 1(1), May 2007.
- [Mil67] Stanley Milgram. The Small World Problem. *Psychology Today*, 61(1):60–67, 1967.
- [Obj11] Object Management Group. *OMG Unified Modeling Language (OMG UML), Infrastructure*, 2011. Version 2.4.1.
- [RN10] Marko A. Rodriguez and Peter Neubauer. Constructions from Dots and Lines. *Bulletin of the American Society for Information Science and Technology*, 36(6):35–41, 2010.
- [SFL⁺12] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, pages 731–742, New York, NY, USA, 2012. ACM.