Flexible Information Management, Exploration and Analysis in SAP HANA

Christof Bornhoevd, Robert Kubis, Wolfgang Lehner, Hannes Voigt and Horst Werner
SAP Labs, LLC, 3412 Hillview Ave, Palo Alto, CA 94304, U.S.A.

Keywords: Schema-flexible Database Management System, Graph Database, Flexible Information Management.

Abstract: Data management is not limited anymore to towering data silos full of perfectly structured, well integrated data. Today, we need to process and make sense of data from diverse sources (public and on-premise), in different application contexts, with different schemas, and with varying degrees of structure and quality. Because of the necessity to define a rigid data schema upfront, fixed-schema database systems are not a good fit for these new scenarios. However, schema is still essential to give data meaning and to process data purposefully. In this paper, we describe a schema-flexible database system that combines a flexible data model with a powerful data query, analysis, and manipulation language that provides both required schema information and the flexibility required for modern information processing and decision support.

1 INTRODUCTION

The monolithic databases of the past, representing self-contained fully integrated information universes, are giving way to much more fragmented ecosystems of data sources. This trend is culminating in the form of today's Web, which offers a breathtaking amount of structured, irregularly structured, and unstructured information.

To make use of multiple diverse data sources, the traditional data integration approach is to integrate those sources into a single context so as to get a comprehensive and consolidated view on the data.

As the schema of data must be known in advance and then coded explicitly in the database, the efforts for data source connection, data integration, and data cleansing, severely limit the viable number and complexity of information sources. As a consequence, many needs for data analysis remain unfulfilled, since required data is not available in an integrated and consolidated form.

SAP is developing a comprehensive solution to better support data exploration and analysis in highly dynamic and heterogeneous business contexts, based on the HANA main-memory centric data management system. The need to tap into a mostly unpredictable set of heterogeneous data sources in short time frames is necessary for agile business analysis and decision support (Cohen et al., 2009).

In contrast to conventional approaches, the solution outlined in this paper follows an "integrate as you go" paradigm to support transformation and consolidation steps on demand. By handling data together with soft-coded metadata, it is relatively easy to consolidate data from different sources with similar meaning and overlapping attribute sets.

Firstly, object types are imported together with the data without requiring upfront modeling. Secondly, it is easier to identify gaps and inconsistencies in a model where data and metadata are treated similarly, because the existence or nonexistence of a property type in a set of data objects can be treated in the same way as the occurrence of a property value. As a consequence, a stepwise integration allows a more agile use of a broader range of data sources with different schemas and with varying degrees of structure and quality (Franklin et al., 2005); (Das Sarma et al., 2008); (XML Query Language, 2007).

While the data foundation of applications gets more diverse, there is also a shift to new end user devices, such as smart phones, tablets, and electronic whiteboards, which dramatically change how people interact with data. The traditional form-based interfaces are commonly based on a three-step interaction pattern of posting a search request, picking a data object from the result list and manipulating that object. This decade-old interaction pattern limits the amount of data a user is able to

Bornhoevd C., Kubis R., Lehner W., Voigt H. and Werner H..
Flexible Information Management, Exploration and Analysis in SAP HANA.
DOI: 10.5220/0004011500150028
In *Proceedings of the International Conference on Data Technologies and Applications* (DATA-2012), pages 15-28
ISBN: 978-989-8565-18-1

review without getting lost, and is insufficient for mass manipulation of data objects. The search step does not allow real exploration but merely helps to shrink the result list; hence, it still requires the user to know what he is looking for.

Modern and future devices allow a more intuitive tactile- and gesture-based interaction, and provide various kinds of context information (e.g. location, orientation and acceleration). Based on the capabilities of these devices, new user interfaces offer a freely interactive and explorative way to work with data. With the ability to present data at instantly and easily changeable levels of detail and to dynamically create context-sensitive views or filters, these new interfaces supply the user with exactly the information that is relevant for his current situation. This flexible and intelligent way of data presentation and interaction lets users search, view, handle, and manipulate very large datasets of varying kinds and structure (Perlin et al., 1993); (Morris et al., 2010); (Werner et al., 2011).

Putting the large variety of available data and the new UI capabilities together, a wide range of new applications that leverage potentially large weakly structured and only partially integrated data becomes possible. For instance, in the area of competitive market intelligence future applications can analyze customer sentiment, but can also be used to discover market trends, competitor moves, regulatory constraints, and so on, early, and handle these proactively. Such applications will be able to crawl the Web and combine information extracted from unstructured content (typically blogs, news pages, press releases) with publicly available structured information (e.g., from Freebase) and private structured information (contained mostly in business applications).

In contrast to the established transactional applications, the content of these collaborative applications - a shared and incrementally composed work context - is less predictable. Hence, it does not fit well with a predefined rigid schema, which is the key design concept of relational database systems.

More flexibility in the data model is provided by key-value stores or document-oriented database management solutions. However, abandoning a rigid up-front defined database schema also implies that the intended meaning of data, which so far has been contained in the schema, must be made explicit and completely handled within the application coding.

Semantic Web technology is one attempt to create a more explicit representation of the meaning of data. But representing meaning by complete and consistent sets of computer-processable rules requires a rigid standardization of metadata vocabularies, and does not provide a concept for dealing with different context-specific views on identical entities, with incompleteness and inconsistencies.

In summary, we see this kind of heterogeneity and the strong scalability demand resulting from the fast growing amount of available and potentially relevant data as one of the main challenges for database technology. Future data repositories need to support the coexistence of heterogeneous content and its incremental integration to the degree needed, based on specific application knowledge and requirements. We therefore see the need for data management systems that do not require the upfront definition of rigid database schemas and that support the efficient representation and processing of large volumes of irregularly structured, not necessarily fully integrated data.

In this paper, we describe an extension to SAP's HANA data management and analytics engine. (Färber et al., 2011); (Gupta et al., 2012). HANA is a main-memory centric database system that leverages new technologies such as multi-core, SSD, and large main-memory capacities to significantly increase performance of analytical and transactional applications. HANA is based on a fundamentally new system architecture that allows storing data either column- or row-oriented. We leverage HANA as the foundation of a new schema-flexible data management system that we call Active Information Store (AIS). AIS capabilities are realized directly on top of the HANA database engine to guarantee good query performance and scalability.

2 REQUIREMENTS

We studied a broad range of uses cases, such as ad-hoc data integration and analysis, competitive market intelligence, social network analysis, idea management business user enablement, and next-generation collaboration tools. Based on these use cases, we identified common requirements for an optimal underlying data store. A data store developed specifically for such applications has to be able to efficiently manage schema-flexible, irregularly structured and partially integrated data with a soft-coded or system generated schema.

With such a data store, schema information does not need to be defined upfront. Instead, the data is self-describing (a soft-coded schema). Data objects of similar structure, or representing similar kinds of entities, do not need to have the same semantic type (are not fully integrated), and data objects of the

same semantic type do not need to exhibit the same structure (are irregularly structured). Once the data is in the data store, changing the schema of data objects has to be possible at any point in time, and must be processed as efficiently as any other first-class data operation (schema-flexible). In short, the data store needs to support the coexistence of heterogeneous content and its incremental integration to the degree needed by the specific application at hand.

In addition, the provided data model must provide strong support for links between data instances of different types to support information correlation and re-organization. Functionally, the data store has to provide the following mass data operations:

- Load, insert, update, and delete objects with possibly irregular structure and without a predefined fixed schema.
- Efficiently add and remove attributes and links/associations to and from data objects.
- Efficiently filter and classify data objects.
- Efficiently traverse links between data objects.
- Calculate the union, difference, and intersection of large sets of objects.
- Group, aggregate, and consolidate possibly heterogeneous data objects.

All operations are mass-data oriented, i.e., they process large sets of data objects at once. Additionally, these operations have to treat data objects as a whole and preserve a data object's integrity as a logical unit.

3 SYSTEM OVERVIEW

We have developed the Active Information Store (AIS) as an extension to the existing HANA engine to provide a powerful schema-flexible data store to address the posed data management challenges. In this section, we will summarize the design principles that guided the development of AIS:

(1) In AIS, a data object consists of a group of semantically and technically typed values that form a semantic unit, e.g. to describe a person or a product. Semantic types, e.g. *person*, describe the intended meaning of a data object, whereas technical types like *Integer* or *String* describe the representation of data in the system. Semantically typed associations represent links between these data objects. For efficient value processing, such as aggregation, all attributes of the same semantic type have the same technical representation type. To remain structurally flexible, semantic types do not impose restrictions on other semantic types related to a data object, e.g. the type *person* does not determine the attributes a *person* instance can have.

(2) AIS strengthens the integrity and identity of a data object as a logical unit of storage and retrieval. Each data object instance has a single identifier that is immutable during regular query processing. Only consolidation or aggregation of data objects will result in new data object instances.

(3) AIS provides light-weight support for semantic data processing. AIS is primarily a data store that incorporates a number of helpful semantic features: (a) Semantic types, which are modeled (conceptually) through the means of the AIS data model; (b) Taxonomies, to organize semantic types; (c) basic semantic operations, such as subsumption and transitivity, which can be used in queries.

(4) AIS uses URIs as the primary mechanism for all identifiers exposed outside of AIS. URIs are a well-established standard for identifiers, and the XML namespace mechanism is adopted.

(5) AIS generally processes data objects in sets, with good support for mass data processing through built-in mass operations in the query language.

(6) To allow for complex operations, all elementary operations of AIS must have minimal side-effects, so that they can be freely combined. This implies that all operations should be as general as possible, and closed in regards to the AIS data model, i.e. produce output that can be used directly as input for other operations. By this principle AIS tries to provide a set of operations that combine compactness with expressiveness.

(7) With AIS, a statement can encompass many complex operations. Multiple insert, update, delete, and query operations can be combined within a single statement. This helps to reduce the number of round trips between client and server, and provides a context for cross-operation optimizations.

(8) AIS leverages the highly efficient in-memory storage layer of SAP HANA. AIS capabilities are implemented directly on top of the HANA engine to guarantee good performance and scalability.

(9) AIS provides a stateless, RESTful client API. The use of complex statements allows the definition of client actions that involve larger units of work that need to be processed as one atomic unit.

4 DATA MODEL

The AIS offers a very flexible data representation model that allows the uniform handling and combination of structured, irregularly structured, and unstructured text data. All data managed and processed by AIS is converted to this common data model.

In this model, data objects are represented as *Info Items*. Info Items are organized and persisted in *Workspaces*, which establish a scope for visibility and access control.

An Info Item provides a single uniquely identifiable data instance, holding a set of *Properties,* which describe the Info Item. A property can be either an *Attribute* or an *Association*. Info Items and their properties have an attached semantic type label that indicates the assumed item class (e.g. *person*), attribute type (e.g. *age*), and relationship type (e.g. *is-parent-of* or *works-for*), respectively. We call these type labels *Terms*.

Formally, an Info Item I is a quadruple (u, T, v, t_I), where u represents the item's URI, T is a set of n property Terms $\{t_1, \ldots, t_n\}$, v is the value function and t_I is the item's Term. The value function $v: T \rightarrow T_1^+ \times \ldots \times T_n^+$ maps every property Term t_i to a list of all values of the Info Item's property of Term t_i.

An Attribute associates a value with an Info Item, and is labeled with a Term. A multi-valued attribute can be represented by multiple attributes of the same Term. *An Association* describes a unidirectional typed relationship between a pair of Info Items. An Association is labeled with a Term indicating the semantic type of the Association. The same pair of Info Items can be related via multiple Associations of different types, and Info Item instances of the same semantic type can be related through different types of Associations.

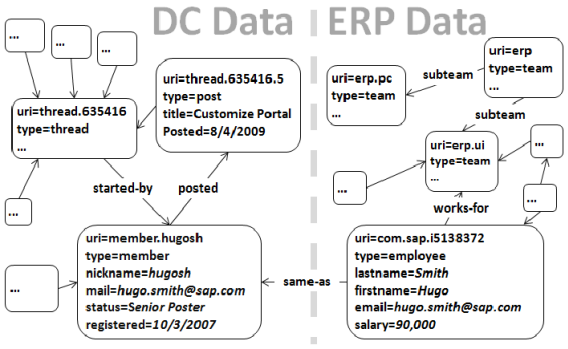

Figure 1: Example of info items and associations.

Figure 1 shows an example of Info Items taken from a data integration and analysis example. The two Info Items on the left side represent entities from a developer community (DC). The two Info Items on the right side originate from an ERP database. The *work-as* and the *posted* Associations link Info Items as their corresponding entities are associated in the respective source databases. The *same-as* Association has been added later as the result of integrating the datasets for analysis. By this means, the data is integrated to the level sufficient to analyze how actively employees participate in the developer community, for example.

As mentioned earlier, a Term represents the semantic type of an Info Item or Property. In addition to its semantic meaning, a Term also has an assigned *Technical Type* that determines the physical representation of the corresponding data element. In the case that a Term describes an Info Item or an Association, the technical type is *Info Item* or *Association*, respectively. For Attributes, the technical type can be *Integer*, *String*, *Date*, etc.

The AIS data model is strict and consistent in its semantic and technical typing of data. All Info Items and all Properties have a semantic type. All Property values of the same semantic type have the same Technical Type. To gain flexibility, the AIS data model loosens consistency regarding the structural implication between semantic Info Item types and semantic Property types. Info Items that have assigned the same semantic type may, and in general do, have different sets of properties. However, the structural information about which Info Item types expose which Property types is not lost; it can be captured with *Templates*. A template is associated with a specific Term and provides information about the structure, i.e., the Properties of the Info Items currently in the store of the corresponding Term. A Template describes the set of *frequent* Properties that are available for most (e.g. 95%) of the currently available Info Items, and the *optional* Properties that are only given for some of the Info Items of the corresponding type.

Terms are represented as Info Items. This allows their retrieval and manipulation either as Terms, i.e., as metadata, or (possibly together with other Info Items) as Info Items, i.e., as data. By allowing Terms to be treated and used as regular Info Items, we give up a strict, and in some applications artificial, separation of data from metadata.

Terms can be taken from domain-specific *Taxonomies* that can be provided to AIS as semantic metadata. Terms from a specific taxonomy are assigned to Info Items and their Associations when they are loaded into AIS, but can be changed later. In this way, Terms can provide a first hook to make

the intended meaning of an Info Item, its Attributes, and Associations, more explicit, by putting it into the context of a taxonomy of type denominators.

Figure 2 illustrates the use of Terms. Terms form a subsumption tree. If a knowledge worker, for example, wants to analyze all Info Items representing a person, i.e., *employees* from the ERP system and *members* of the developer community, she could easily introduce a new common Super Term, e.g., *person* for *employee* and *member*, as shown in the figure. Additionally, she might let AIS create a Template based on all Info Items that the Term *person* subsumes.

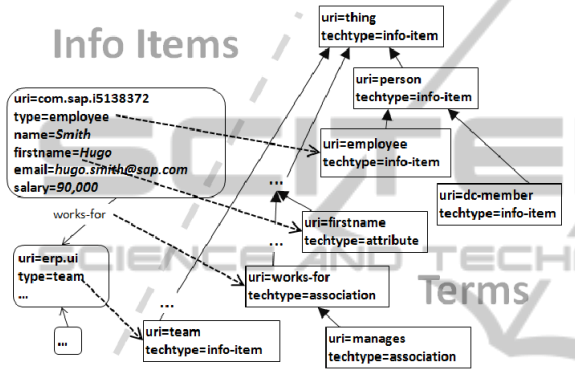

Figure 2: Example of info items and terms.

In summary, AIS provides a very generic and flexible graph-oriented data model, where Info Items form the nodes and Associations the edges of a graph. The data model does not enforce a tight integration of and structural consistency among data items, which typically come from different heterogeneous sources into one common data schema. Rather, it supports the coexistence of data items from different sources and their stepwise integration. While providing rich schema information, AIS does not require defining the schema of the data upfront in a rigid way. Rather, the schema evolves as new Info Items are imported or created, new attributes are added, new Associations between Info Items are established, and new Terms are introduced.

5 QUERY LANGUAGE

AIS offers a powerful data query and manipulation language called WIPE. In this section, we outline the basic structure of an AIS statement, and describe the set of operations supported in WIPE.

In the case of AIS, a statement can be composed of multiple data manipulation operations and may return multiple result sets. The data operations of a given statement are executed in their order of declaration as one logical unit of work. More formally, an AIS statement is defined as a triple (w, N, O), where w is the workspace the statement is operating in, N is a set of namespace definitions and $O = op_1, \ldots, op_n$ is a list of data operations.

```
1 | WORKPACE uri:dcAnalysis;
  | UPDATE $e : uri:employee {
2 |   ADD uri:same-as = $m : uri:member WITH $e->email == $m->mail
  | };
  | $contributers = $e : uri:employee WITH COUNT($e->same-as->posted) > 0;
3 | $recentContributers = $e : uri:employee WITH ONE $p : $e->same-as->posted;
  | RETRIEVE uri:overview {
  |   ITEM { uri:noMembers = COUNT(uri:member)
4 |          uri:noEmployees = COUNT(uri:employee)
  |          uri:noContributers = COUNT($contributers)
  |          uri:noRecentContributers = COUNT($recentContributers) }
  | };
5 | RETRIEVE uri:recentContributers PROPERTIES {uri:uri, uri:name, uri:email}
  |          $recentContributers;
```

Figure 3: Example of an AIS statement.

During execution, every statement has a statement context C, to keep track of all execution-related information. Primarily, a statement context encom-passes a set of local identifiers, which can be used to structure a statement. In that sense, a statement context C is a function defined by a set $L = \{l_1, \ldots, l_n\}$ of local names and a set $E = \{e_1, \ldots, e_n\}$ of expressions so that $C(l_i) = e_i$.

Figure 3 shows a statement to find employees contributing to the development community. First, the statement specifies the workspace it will be executed in. Second, it establishes associations between employee items and member items having the same email address. Third, it defines two Info Item sets, one containing all employees that have contributed to the community and one containing employees that have recently contributed. Forth, it retrieves some basic counts, to give an overview of what the data looks like. Fifth, the statement retrieves the recently contributing employees. The example illustrates the use of the update, assign, and retrieval operations within a single statement.

AIS offers *Load*, *Insert*, *Update*, and *Delete* as manipulation operations, *Retrieval* to retrieve data, and *Assign* for structuring complex statements. In the next two sections, we describe these data operations in more detail.

Load and insert operations allow to bulk load and to create new Info Items in a workspace. Update operations allow changing the values of properties well as adding new properties or removing properties from Info Items. The example statement shown in Figure 3 adds an association *same-as* to all employees (line 2). The target of an association can be specified by any valid AIS expression that represents a set of Info Items.

The retrieval operation allows fetching data and is defined as a triple (e, u, T), where e is an expression, u is an optional URI, and T is an optional set of n property Terms $\{t_1, \ldots, t_n\}$. Expression e defines the set of Info Items to retrieve. Any valid AIS expression that results in a set of Info Items may be used here. In case of multiple retrieval operations within a single statement, the name (i.e. URI) of a result set can be used to distinguish the result sets on the client side. The example statement retrieves two result sets (line 4 and 5), called `overview` and `recentContributers`, respectively.

When an Info Item is shipped to the client, only a subset of the item's properties might be of interest to the client. If T is provided, the retrieval operation will select only properties of the Terms in T to the result set (projection). Formally, for every Info Item $I = (u, T_I, v, t_I)$ an Info Item $I' = (u, T', v', t_I)$ is shipped so that $T' = T \cap T_I$ and v' is only defined for $t \in T'$. In the example, the second retrieval operation projects only the properties `uri`, `name` and `email`.

Finally, the assign operation allows the client to structure a complex statement by naming and re-using sub-expressions. In particular, an assign operation is defined as a pair (l, e), with l being the local name and e being the expression that l will be bound to. The binding is stored in the statement context C, so that C will be defined by $L = L \cup \{l\}$ and $E = E \cup \{e\}$ after the assignment. The example statement illustrates that in line 3.

Expressions form the core of the AIS query language. In particular, an AIS expression is defined to be a literal, a reference (e.g. a URI), a local name or an operation applied to one or more sub-expressions. As a starting point, AIS provides the built-in local name `$ALL`, which represents the set of all Info Items persisted in the current workspace. A set of all Info Items of a specific Term can simply be referenced by the URI of the Term. For example `uri:member` references the set of all members of the developer community.

Based on these initial sets of Info Items, expressions allow the refinement of the requested Info Item set in a retrieval operation. Besides standard numeric algebra, Boolean algebra, string operations, and other value-type specific operations, set operations are most crucial for mass data processing. AIS offers six essential kinds of set operations: (1) Set Algebra, (2) Graph Traversal, (3) Filtering, (4) Quantification, (5) Aggregation Functions, and (6) Grouping. We will describe these categories in the following.

Set Algebra. Set Algebra operations encompass the three basic operations *Union* ∪, *Intersection* ∩ and *Difference* \ (asymmetric difference). Given two sets A and B of Info Items the three operations are defined as follows:

$$A \cup B = \{x : x \in A \vee x \in B\}$$
$$A \cap B = \{x : x \in A \wedge x \in B\}$$
$$A \backslash B = \{x : x \in A \wedge x \notin B\}$$

For example, the set of all developer community posts and blog articles can be expressed as:

```
uri:post UNION uri:blog-article
```

Graph Traversal. The traversal operation allows resolving properties, i.e., associations and attributes. It is defined as a pair (S, T), where S is the set of Info Items to start the traversal from and T is the set of property Terms to traverse over. The resulting set contains all values that the properties of any Term $t \in T$ have on all Info Items $I \in S$:

$$(S, T) = \bigcup_{i,j} v_i(t_j) \quad \text{with} \quad v_i \in I_i \in S \wedge t_j \in T$$

In the case of Associations, the result is a set of Info Items. In the case of attributes, the result is a set of primitive values. Terms for Attributes and Terms for Associations cannot be mixed in a single traversal step to avoid a result set consisting of both Info Items and primitive values.

For association traversals, the operation can be marked with the Reverse flag (`<-`) or the Transitive flag (`*`). In the first case, the associations will be traversed in the opposite direction. In the second case, the traversal results in all items reachable from S by any association path in T^+. In our example, all emails of the employees can be retrieved with the expression:

```
uri:employee->uri:email
```

while the expression

```
{uri:com.sap.std.erp.ui}<-uri:works-for
```

retrieves all employees of the ERP UI team. Note that the traversal operation strictly works on sets and we have to create a set containing the specific team Info Item, referenced by its URI. Assuming that teams have sub-teams, we can get all emails of all team members and sub-team members with:

```
{uri:erp.ui}->uri:subteam*<-uri:works-for
```

Filter. With the filter operation, Info Item sets can be filtered according to a given condition. Formally, a filter operation is defined as a pair $(S, b : S \rightarrow \{\text{false}, \text{true}\})$, where S is the Info Item set to filter on, and b is a condition function that maps every Info Item of S to a Boolean value. The resulting set

contains only Info Items of S mapped to true:

$$(S, b) = \{s: s \in S \wedge b(s)\} \ .$$

The condition function b is a logical expression tree which can consist of Boolean operations, relational conditions and quantification. A relational condition represents a basic value comparison. It has a left value expression, a comparator, and a right value expression, and returns true if the comparison holds for the two values. For example, we can filter all employees by their salary:

```
$e: uri:employee
WITH $e->uri:salary > 100k
```

Quantification. Quantification conditions check whether a given quantity of elements of an Info Item set or a value set satisfy a condition function. A quantification operation is defined as a triple (Q, S, c), where Q is a quantifier, S is the set, and c is the condition function. The quantifier specifies the required quantity for a quantification to yield true. AIS supports the existential quantifier ∃ and the universal quantifier ∀:

$$(\exists, S, c) = \exists s \in S, c(s)$$
$$(\forall, S, c) = \forall s \in S, c(s)$$

In the example, the quantification condition allows us to filter all teams with at least one high-income earner:

```
$t : uri:team
WITH ONE $e : $t<-uri:works-for
         WITH $e->uri:salary > 100k
```

Aggregation Functions. Aggregation functions map a set or a list of Info Items to a single value. AIS supports the basic aggregation functions COUNT (number of element), SUM (summation), AVG (average), MIN (minimum), and MAX (maximum). For instance, the following expression filters development community members according to the number of posts they made:

```
$m : uri:dc-member
WITH COUNT($m->posted) > 500
```

Grouping. AIS offers a very general grouping operation, where we understand grouping as the creation of a new Info Item from a group of existing Info Items. A grouping operation is defined as a triple (S, E_G, I), where S is the base set, E_G is a set of grouping expressions, and I is an Info Item constructor function. All Info Items s in S with equivalent results on all expressions in E_G form a group g. The resulting set of a grouping operation contains new Info Items created by applying I to every g: $(S, E_G, I) = \{g: g = I(g) \wedge g \in E_G(S)/\sim\}$.

Assuming that D is the domain of all Info Items, an Info Item constructor function $I: \mathcal{P}(D) \to D$ maps a subset of Info Items to a new Info Item. An Info Item constructor can be defined (similar to an Info Item) as a quadruple (u, T, E_V, t), where u is the new item's URI, T is a set of n property Terms $\{t_1, \ldots, t_n\}$, E_V is a set of n expressions, and t is the item's Term. When applied to a group g, $I(g)$ results in a new Info Item of Term t with n properties, so that the i-th property is of Term $t_i \in T$ and is assigned with the evaluation of expression $e_i \in E_V$, where e_i is evaluated against group g.

$$(u, T, E, t) = (u, T, v: v(t_i) = e_i(g), t)$$

For instance, to query the average number of posts of DC members, we would define the grouping as: $\left(\text{uri:post}, E_V, (u, T, E_V, t)\right)$ with:

$$E_V = \left\{s \xleftarrow{\text{uri:posted}}\right\}$$
$$T = \{\text{uri:email,uri:numberOfPosts}\}$$
$$E_V = \left\{\left(g \xleftarrow{\text{uri:posted}}\right) \xrightarrow{\text{uri:email}}, \text{COUNT}(g)\right\} \ .$$

In statement syntax that is:

```
GROUP $s : uri:posts AS $g BY $s<-
uri:posted
TO ITEM {
 uri:email = $g<-uri:posted->uri:email,
 uri:numberOfPosts = COUNT($g)
}
```

Grouping can also group Info Items based on aggregates. For example, listing for each team size the ratio of heavy posters can be done with the expression:

```
GROUP $e : uri:employees AS $g BY
COUNT($e->uri:works-for<-uri:works-for)
TO ITEM {
 uri:sizeOfTeam =
  COUNT($e->uri:works-for<-uri:works-for),
 uri:heavyPosterRatio =
  COUNT($e : $g WITH
       COUNT($e->same-as->posted) > 500) /
  COUNT($e : $g WITH
       COUNT($e->same-as->posted) < 500)
}
```

A. Term Operations. Most of the described AIS operations rely on Terms. AIS organizes Terms in taxonomies to allow an explicit description of subsumption relationships between Terms. Formally, subsumption is a function τ(t) that results in a set

$$T = \{t_{sub}: t \text{ is super-type of } t_{sub}\} \cup \{t\}$$

with respect to the underlying taxonomy. Obviously, other operations have to deal with this resulting set

of types so that their result reflects the meaning of subsumption. For AIS, we allow subsumption for Term URIs representing Info Item sets, association traversal, and for the filtering on the type property of Info Items.

B. Life Cycle. As mentioned before, single Info Items are the granularity of data storage, but sets of Info Items reflect the granularity of processing data. Therefore, Info Items and Info Item sets are subject to different life cycles when being processed via query language statements.

Every Info Item, as the primary unit of storage, is generally considered to be persistent and can be identified among all Info Items in a workspace by its URI. Operations, except Info Item constructor functions, do not create new Info Items.

Info Item constructor functions are the only operations to create new Info Items. These newly created items are transient, i.e., they are not persistent by default, and will live only as long as the statement in which the transient items were created is processed. Transient Info Items do not necessarily have a unique identity. For that reason, transient Info Items cannot be mixed with persistent items in operations and sets. However, AIS can return transient Info Items to the client or persist transient Info Items with an Insert operation within the same statement.

A set of Info Items, as the primary unit of processing, is generally transient. Every operation that results in a set of Info Items conceptually creates a new set. However, sets can be explicitly persisted for later retrieval, or re-use in other statements.

6 ARCHITECTURE

AIS provides an extension of the SAP HANA in-memory database engine. The existing HANA storage and query processing layer fits well the needs for scalability, speed, and built-in analytical processing capabilities.

Figure 4 outlines the conceptual architecture of AIS. At the front-end AIS exposes a RESTful client API. A client sends its statements as requests to this service. Inside AIS, a REST server receives this request, extracts the statement, and hands it over to the parser. The parser transforms the statement into an internal representation and a rule-based statement simplification procedure prepares it for execution. A complex AIS statement, which can be divided into an operational processing step and a result fetch step, usually involves multiple storage layer interactions before shipping the result sets back to the client.

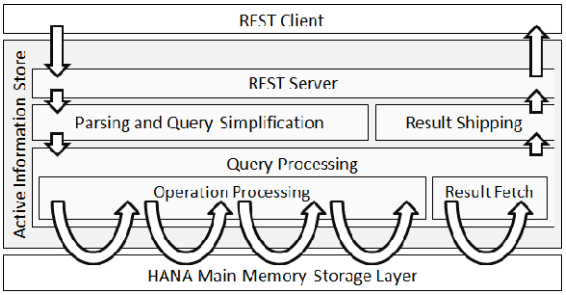

Figure 4: System architecture overview.

AIS does not preserve any state with regard to the client. For scalability over the number of concurrent clients, the processing of client requests can be easily distributed over multiple worker threads within a single machine and over multiple machines. The HANA storage layer already provides excellent scalability by being able to distribute data across multiple machines.

In the following subsections, we describe (1) the capability of the storage layer, (2) how we map the AIS data model to a fixed schema as required by the storage layer, and (3) how we process AIS statements on top, while utilizing as much of the existing storage layer capabilities as possible.

A. Storage Layer Capabilities. In the storage layer, data is stored in fixed-schema tables. Data can easily and highly efficiently be inserted, removed, modified, and queried. To retrieve data, the storage layer supports "Select Group Order Project" (SGOP) queries and join views to join tables on predefined join conditions over an arbitrary set of tables with potentially different join semantics.

A SGOP query (t, p, G, A, O) selects all rows that match the predicate p from table or join view t, groups the rows by all columns $c \in G$, creates aggregation columns $c \in A$, sorts the result by all columns $c \in O$, and finally projects to all columns $c \in G \cup A$. If A is empty, G will be a regular projection. If G is empty, the query will return all available columns. An aggregation column $a(c)$ aggregates a column c of t with the aggregation function a (sum, minimum, etc.). Predicate p is a logical expression of arbitrarily nested conjunctions and disjunctions of conditions of the form $R(c, v)$, where R is a relation (equal, less, greater, in, etc.), c is a column of t, and v is a value literal.

B. Mapping to the Storage Layer. We need to provide the required flexibility in the AIS data

model while relying on an underlying schema-fixed storage layer, i.e. we need to map Info Items to a fixed schema. The possible mapping alternatives are determined by the AIS requirements on the one hand and the storage layer capabilities on the other hand:

Logical Type Flexibility. We want to create, remove and change logical types at any time easily and efficiently. In addition, the set of logical types in the data is unknown at deployment time. The mapping has to "soft-code" all logical types, i.e., treat the logical type information as regular data in the storage layer.

Technical Type Preservation. We want to efficiently utilize the type-specific operations of the storage layer, such as value comparisons or aggregations. The mapping has to preserve the technical value types of property values. Otherwise, the execution of such an operation would include a cast on every value it operates on.

Existence Representation. We want to store data space-efficiently and not waste memory resources. The mapping has to avoid an explicit representation for non-existing values. In particular, Null values are not a solution for non-existence, since they are not stored more efficiently than regular values. Only existing values should be explicitly stored.

The basic dimensions for a mapping are (1) the entity orientation, i.e., horizontal or vertical, (2) the level of soft coding, i.e., how much typing is treated as data or is mapped to storage layer elements, (3) entity abstraction and decomposition, i.e., if entities are generalized and decomposed into chunks on the storage layer. Regarding these building blocks, the following considerations justify the mapping we use for AIS data.

(1) Irregularly structured Info Items have a varying and unpredictable number of properties. To avoid the explicit representation of non-existing properties, we chose a vertical over a horizontal orientation.

(2) The required logical type flexibility and technical type preservation already define the level of soft coding. Hence, we represent all logical types as data and store all values in columns with their corresponding technical type. Additionally, we partition the value table by technical type to avoid Null values for non-existing combinations of property and technical type.

(3) Entity abstraction and decomposition are good means to make an existing well-defined schema more flexible. For an application, entity chunks can be tailored based on the application's schema and workload. AIS is an application-invariant repository, where schema and workload are not known in advance. Hence, we do not incorporate any entity abstraction and decomposition into our storage layer mapping.

To summarize, we selected a vertical schema with partitioned value tables. The storage layer schema encompasses the tables shown in Figure 5.

Info Item. This central table keeps the header information of every Info Item *including* the item's URI, its Term, and a unique identifier used to reference the item in the other tables. This table also keeps the basic attributes of each Term, including the Term's unique identifier, URI, the identifier of the Term's super term, and the technical type of the Term's instances. This table is used for filtering based on basic Info Item attributes, in particular its semantic type, before accessing the value tables.

The Term relationship and the Super Term relationship are semantically different, but since all technical types are hard-coded, we can re-use the term column to store the super term relation of Terms. The column for the technical type remains unused for Info Items but, if needed, is leveraged to indicate the technical type if the Info Item is representing a Term.

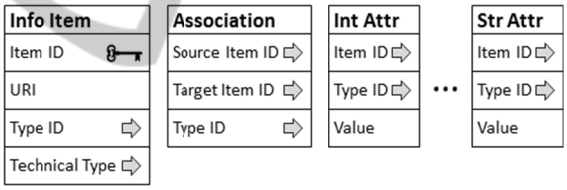

Figure 5: Storage layer schema.

Association. The second major table of the AIS relational schema records unidirectional associations between Info Items, including the identifiers of the source item, the target item, and the Term (i.e. semantic type) of the association.

Attribute. Finally, the third table stores the attributes of the Info Items. Using a separate table for each technical type preserves the types of the individual columns and enables the AIS execution engine to fully utilize the selection and aggregation capabilities of the HANA storage layer. Beside the actual value, the table stores identifiers of the attribute's Term and of the Info Item it belongs to.

With this mapping, we maintain the flexibility required for the AIS data model, and are able to utilize as much of the storage layer's capabilities as possible.

C. Statement Processing Overview. Having a powerful storage layer at hand, the statement

processing engine of AIS remains lean and does not replicate any of the capabilities of the storage layer. Thus, our processing model tries to delegate as much of the query processing to the powerful storage layer as possible, and the process-sing model is designed to leave the payload data in the storage layer until Info Items have to be shipped to the client. Since the processing of Load, Insert, Update, and Delete operations is straight-forward, we focus here on the Retrieval operation.

Since the AIS query language is more expressive than the query language of the underlying storage layer an AIS statement, in general, results in multiple storage layer interactions (Figure 6). After parsing, AIS represents a Retrieval operation as a statement tree. Every node in this statement tree presents one of the operations of the AIS query language as used in the specific Retrieval operation. Assuming that its sub-expressions have been processed, an operation can be processed either without, with one, or with multiple storage layer interactions.

In order to optimize the number of storage layer interactions, the AIS statement simplifier removes all idempotent operations, e.g., multiple negations, and processes all operations that can be executed without any storage layer interaction. Also, the simplifier integrates all traversal operations on attributes into their parent operation in the statement tree.

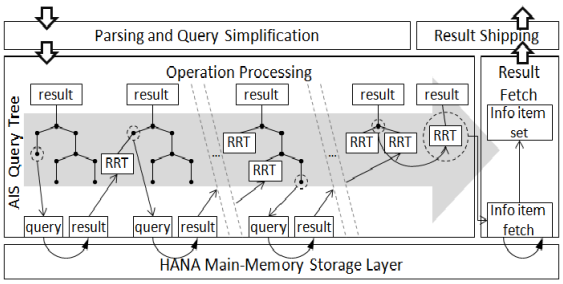

Figure 6: Statement processing model.

The AIS statement simplifier is based on an extensible rule set. The simplification algorithm traverses the statement tree top-down and then bottom-up again. At each node of the tree, it rewrites the tree according to the matching rules. Each rule itself defines whether it is applied during the top-down or bottom-up run, and whether the simplifier should try to apply the rule repeatedly to the same node. After the simplification step, the statement tree is ready for the actual operation processing.

D. Operation Processing. After parsing and simplification, AIS processes all operations of the statement tree bottom-up, as illustrated in Figure 6. The result of each operation is a Retrieval Reference Table (RRT). An RRT is a set or list of Info Item identifiers or values, respectively. Additionally, an RRT can contain a set of Info Item identifiers as back references for each of its entries. After an operation has been processed, it is replaced with its resulting RRT in the statement tree. The RRT then forms the input to the operations higher up in the tree. Eventually, the statement tree collapses into a single RRT, which will be handed over to the result fetch step.

Back references in RRTs are an important mechanism for mass data processing. Many of the AIS operations comprise *Foreach* semantics. If these operations contain sub-operations that require separate processing, the sub-operation would have to be processed iteratively. With back references, an operation keeps track of which elements of its output resulted from which elements of its input. By this, the iteration can be avoided and the sub-operation can be processed in a single step. Nevertheless, this approach trades space for time, and is therefore bound by the available resources.

A Retrieval operation processes results in the form of an RRT and may involve a number of interactions with the storage layer, depending on the operation being processed.

Set Algebra. Set Algebra operations are processed in AIS without any storage layer interaction. Two input RRTs are directly united, intersected or subtracted.

Traversal. Traversal operations require querying the association table. The traversal (S, T) will result in a storage layer query $(\text{Association}, p, O, \emptyset, O)$, with

$$p = (\text{sourceItemID} \in S \wedge \text{termID} \in T)$$

$$\text{and } O = \{\text{targetItemID}, \text{sourceItemID}\}$$

Transitive traversal is realized iteratively. In every iteration, we set S to the Info Item identifiers fetched in the previous iteration and execute the storage layer query again. More specifically, the predicate p is adjusted for each iteration to avoid traversing the same associations multiple times. Assume S as the set of Info Item identifiers fetched in the previous step, and R as the set of Info Item identifiers retrieved in all iterations before the previous one, then p is set to

$$(\text{sourceItemID} \in S \wedge \text{targetItemID} \notin R \wedge \text{termID} \in T)$$

If the storage layer query has an empty result, we have reached the fixed point. Theoretically, a transitive traversal can result in a very large set of Info Items, in an extreme case encompassing the

complete workspace. Hence, processing a traversal operation can become very time and space consuming. Practically, however, we have not encountered this problem for our use cases so far.

The traversal operation is also defined for attributes. However, we always integrate the processing of an attribute traversal in its nesting operation.

Filter. The processing of a filter operation depends on its condition function, and we distinguish two cases: (1) the condition function consists only of logical operators and property-value comparisons, or (2) it is a more complex expression.

In the first case, the statement simplifier has transformed the condition function into disjunctive normal form. The atoms are property-value comparisons $R(t, v)$, each stating that a property of Term t has to be in relation R with value v, e.g., the property salary has to be greater than 50k. Although the storage layer is able to process predicates of this class directly, the processing is more complex because of the vertical schema. Therefore, we directly push down disjunctions, but resolve conjunctions by executing them as disjunctions on the storage layer and post-processing the conjunction within the AIS context. For instance, the predicate `uri:a>5 AND uri:b=2` is executed as

$$(\text{termID} = id_a \wedge \text{value} > 5) \vee (\text{termID} = id_b \wedge \text{value} = 2)$$

We partition attributes in the storage layer by their technical type, and therefore have to execute comparisons of attributes of different technical types in separate storage layer queries. Assuming a condition function involves N conjunctions and M technical types, we have to execute $N \cdot M$ storage layer queries. In practice, N and M appear to be rather small, so that this approach is feasible.

All queries order their results by Info Item identifiers. AIS then merges the result sets using interleaved scans. The merge results in a stream of (i, T) pairs, where i is an Info Item identifier, and T a set of Term identifiers. After that the stream is filtered to check whether every Info Item has matches for every property Term of one conjunction. In our example, we would check that every Info Item had matches for `uri:a` and `uri:b`.

In the second case, if a condition function is more complex and involves other operations, e.g., a traversal operation, all nested operations will be child nodes of the filter operation in the statement tree. The sub-operations can be seen as a function $f: S \rightarrow S_D$, where S is the set that needs to be filtered and S_D is the set the property-value comparisons are defined on. For example, in the filter operation:

```
$e : uri:employee
WITH COUNT($e->same-as->posted) > 0
```

S is the set of employees and S_D is the set of count values. The sub-operations are two traversals and an aggregation function. The processing of these sub-operations results in an RRT with back references to the Info Items in S (employees in our example). Then, we regularly process the actual filter (in our example: count value greater zero) on this RRT as described in the first case, and unite all remaining back references to get the final result.

Quantification. Quantification relies on filter processing. The existential quantifier is weaved into the filter processing to exploit early-out opportunities. Here, we return *true* as soon as we see an item successfully passing the post filter predicate q. For the universal quantifier, we use an interleaved scan of S and the filter result S', both already ordered by item id, and return false as soon as we have an item in S that is not in S'.

Aggregation Functions. A group of aggregation functions $f_1, \ldots, f_n$ applied to attribute traversal operations $(S, T_1), \ldots, (S, T_n)$ can be executed directly in the storage layer. Assuming all attribute Terms $t \in T = \bigcup_{i=1}^{n} T_i$ have value type vt, AIS executes the query $(\text{table}_{\text{vt}}, p, \{\text{termID}\}, A, \{\text{termID}\})$ with

$$p = (\text{itemID} \in S \wedge \text{termID} \in T)$$
$$\text{and } A = \{f_1'(\text{value}), \ldots, f_n'(\text{value})\}$$

where f_i' is the corresponding aggregation function at the storage layer for f_i. If all attribute Terms $t \in T$ are not of the same value type, then one storage layer query per involved value type is executed.

Grouping. The grouping operation consists of the grouping expressions (the set E_G) and the creation of new Info Items. Each grouping expression results in an RRT of the grouping values that have resulted from the expression and back references to the Info Items to be grouped. Each RRT is sorted by the Info Item identifiers. With interleaved scans, AIS merges all RRTs into a single RRT, which now consists of the grouping values of all grouping expressions plus the back references. Now, AIS determines the actual groups of Info Item identifiers. All back references that belong to equal grouping values form one group.

The item constructor creates a new Info Item for each group of Info Item identifiers. Each group is processed separately. Given an item constructor function (u, T, E_V, t), we process each value expression in E_V on the Info Item identifiers in the group, and assign the resulting value to the corresponding property of the new item.

All grouping expressions E_G, and likewise all value expressions E_V, have to be executed on the same set, the set of Info Items to group, and a group of Info Items, respectively. For that, we leverage a number of shortcuts and optimizations. First, we group traversals on the same source set for execution. Second, we similarly group aggregation functions on traversals. Third, for expressions with identical sub-operations on the same source set, we execute the sub-operation only once and re-use the resulting RRT. Fourth, if a value expression is identical to a grouping expression, we take the result directly from the RRT used to determine the groups.

E. Result Fetch. Finally the result fetch retrieves the actual payload data of the retrieval. Assuming a Retrieval operation $(\mathrm{e}, \mathrm{u}, \mathrm{T})$ whose query expression e has been processed to an RRT S (a set of Info Item identifiers), AIS now fetches all rows from the property table whose item id is in S and whose Term id is in T, by executing for every value type vt of the property Terms in T the query $(\mathrm{table}_{\mathrm{vt}}, \mathrm{p}, \{\mathrm{itemID}, \mathrm{termID}\}, \emptyset, \{\mathrm{itemID}\})$ with

$$p = (\mathrm{itemID} \in S \wedge \mathrm{termID} \in T) \quad .$$

Two more storage layer queries are necessary: (1) the Info Item table is queried for the header data of the items, and (2) a join view over the association table and the Info Items table is queried for associations. All of these queries order their result by Info Item id. With interleaved scans, AIS finally merges the values into Info Items.

7 RELATED WORK

AIS clearly falls into the category of NoSQL databases. The term NoSQL covers a wide range of data stores, which vary considerably in the data model used and the query and interaction capabilities offered.

In terms of data models used in NoSQL databases, key-value based, hierarchical, and graph-based data models can be distinguished. The group of key-value stores includes document stores (for instance CouchDB (CouchDB, 2012)), pure key-value stores (for instance Amazon S3 (Amazon S3, 2012)), and wide column stores (such as Bigtable (Chang et al., 2008), or Cassandra (Lakshman et al., 2009)). Generally, a key-value based data model associates one to three hierarchical keys with non-typed byte arrays. Although very general and extremely flexible, the key-value concept does not offer any inherent higher means to type and associate entities. Approaches to work around these conceptual shortcomings rely on interpretation by the client application and are transparent to the data store, including its query mechanisms.

Document stores typically follow a hybrid approach nesting two data models. Being primarily key-value stores, document stores additionally structure values using a hierarchical data model such as XML (Extensible Markup Language, 2008) or JSON (Crockford, 2006). A tree of flat objects allows modeling an entity's structure and preserving its integrity as a unit. Furthermore, it is possible to hierarchically associate entities by nesting them.

Graph-based data models are more general and allow arbitrary associations. Probably the most general graph-based data model is RDF (RDF/XML Syntax Specification, 2004). RDF stores statements about entities as subject-predicate-object triples, which form a labeled graph. Consequently, when stored in RDF, entities are decomposed into statements. The entity's integrity as a unit is lost and must be reassembled during retrieval. RDF is probably the most general data model. However, RDF is too general if the representation and management of composite data objects is required.

The AIS data model resembles a graph of plain objects. Plain objects graphs are a very flexible representation of data, where values and links can be easily added and removed, and the entity integrity is preserved at the same time. Comparable to AIS, Neo4J (Neo4J, 2012) is a graph database with a data model incorporating only a very weak type system. Besides missing support for taxonomy and subsumption, semantic attribute types in Neo4J do not imply any technical value type. Weak technical typing allows more flexibility at the price of required type casts. AIS avoids type casting for better aggregation performance. Freebase (Bollacker et al., 2008) is a public graph database operated by Google. Freebase offers an advanced type system without hard-coded schema but type-specific metadata defined (at runtime) before data can be stored. The Freebase concept lacks taxonomy and subsumption support.

Regarding query and interaction capabilities, everything from simple APIs to proper query languages can be found in the zoo of NoSQL databases. APIs are offered by the majority of the key-value stores. These APIs are typically very limited in their expressiveness – comparable to the internal record access methods of a standard relational database system.

Query languages usually come with the more advanced hierarchical and graph-based data models.

XQuery (XML Query Language, 2007), SPARQL (SPARQL, 2008), Gremlin (Gremlin, 2012), MQL, and Cypher (Cypher Query Language, 2012) are the most important ones to mention here. XQuery is a powerful language to work with XML. It strictly builds on the hierarchical nature of XML, which makes it not a good fit for graph-structured data.

SPARQL is designed to query RDF data sets. A SPARQL query matches a graph pattern to a labeled graph. Like RDF, SPARQL has no notion of an entity as a unit. To query an entity, all properties of this entity have to be known in advance. Further, SPARQL lacks means to aggregate values or objects, which makes it inappropriate for most graph analytics tasks.

Gremlin is a language for graph querying, analysis, and manipulation. Since Gremlin heavily builds on graph traversals it is a powerful language to operate with graphs. However, Gremlin is a programming language rather than a query language. There is no inherent language support for set operations, aggregation, or consolidation.

MQL is the query language of Freebase. It strictly follows the query-by-example paradigm. Thereby, it allows a relatively simple composition of powerful queries. Nevertheless, by that simplicity it narrows the granularity of client interaction to single queries per roundtrip.

Cypher is the query language of Neo4j. With an SQL-like syntax, Cypher offers filtered traversal from a given node set and projects out the value of interest found along the traversal. It supports ordering an aggregation. Cypher is the graph query language closest to WIPE. However, it lacks WIPE's support for complex retrieval and manipulation statements.

8 CONCLUSIONS

We presented the challenges faced by existing and emerging applications that require a schema-flexible DBMS that: (1) needs to handle data of different schemas and with different degrees of structure without upfront expensive data integration, (2) has to support changes in the data schema during data query and processing efficiently, but (3) still needs to incorporate the notion of data schemas and a proper type system, so that types can be used to manipulate and retrieve data.

We described AIS as SAP's answer to the challenges of these new kinds of business applications. In this paper we provided a system overview, and described our data model and data query and manipulation language. We sketched the system's architecture and how its statement processing is implemented. A description of experimental results, in particular performance experiments, are planned for upcoming publications.

Finally, we discussed existing approaches related to our data model and data query language. We pointed out that existing data models either are structurally too weak or lack an adequate type system. For query languages, we argued that the discussed approaches are either not expressive enough or too specialized in data models or operations to be suitable for our target applications.

We are currently using AIS in Semantic Business Applications where we represent user and application context in the form of semantic networks, our Business Network Services platform, where we use AIS to efficiently represent and analyze company relationship networks, and in Sentiment Analysis applications, where we combine and analyze internal structured business data with external irregularly structured social media data.

Ongoing work includes the implementation of dedicated physical storage structures and graph specific operations, e.g., graph traversal operators, directly inside the HANA database engine to further optimize query performance.

REFERENCES

Amazon S3. 2012. http://aws.amazon.com/de/s3/.

Apache CouchDB. 2012. http://couchdb.apache.org/.

Aulbach, S. et al., 2008. Multi-tenant Databases for Software as a Service: Schema-mapping Techniques. In *SIGMOD*.

Bollacker, K. et al., 2008. Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge. In *SIGMOD*.

Chang F. et al., 2008. Bigtable: A Distributed Storage System for Structured Data. In *ACM Transactions on Computer Systems, 26(2)*.

Cohen, J. et al., 2009. MAD Skills: New Analysis Practices for Big Data. In VLDB Endowment*, 2(2)*.

Crockford, D., 2006. The Application/JSON Media Type for JavaScript Object Notation (JSON). *RFC 4627, http://tools.ietf.org/pdf/rfc4627*.

Cypher Query Language for Neo4j. 2012. http://docs.-neo4j.org/chunked/stable/cypher-query-lang.html.

Das Sarma, A., at al., 2008. Bootstrapping pay-as-you-go data integration systems. In *SIGMOD*.

Färber, F. et al., 2011. SAP HANA Database – Data Management for Modern Business Applications. *SIGMOD Record, 40(4).*

Franklin, M. J. et al., 2005. From Databases to Dataspaces: A New Abstraction for Information

Management. In *SIGMOD Record, 34(4)*.
Gremlin – A Graph-based Programming Language http://github.com/-tinkerpop/gremlin/wiki.
Gupta, U. et al., 2012. SAP HANA Overview and Roadmap. *SAP Community Network*, http://www.sdn.sap.com/irj/scn/index?rid=/library/uuid/6015ec1d-7f7d-2e10-06b8-edfa52a4c981.
Lakshman, A., Malik, P., 2009. Cassandra: Structured Storage System on a P2P Network. In *PODC*.
Morris, M. R. et al., 2010. Search on Surfaces: Exploring the Potential of Interactive Tabletops for Collaborative Search Tasks. In *IPM, 46(6)*
Neo4J. http://neo4j.org/. 2012.
Perlin, K., et al., 1993. Pad: An Alternative Approach to the Computer Interface. *In SIGGRAPH.*
Shen, W. et al., 2008. Toward Best-effort Information Extraction. In *SIGMOD*.
An XML Query Language. W3C, January 23rd, 2007. http://www.w3.org/TR/xquery/ .
Extensible Markup Language (XML). W3C, Nov 26th, 2008. http://www.w3.org/TR/xml/.
RDF/XML Syntax Specification (Revised). W3C, Feb 10th, 2004. http://www.w3.org/TR/REC-rdf-syntax/.
SPARQL Query Language for RDF. W3C, Jan 15th, 2008. http://www.w3.org/TR/rdf-sparql-query/.
Werner, H. et al., 2011. MOAW: An Agile Visual Modeling and Exploration Tool for Irregularly Structured Data. In *BTW*.