Combining Unstructured, Fully Structured and Semi-Structured Information in Semantic Wikis

Rolf Sint[1], Sebastian Schaffert[1], Stephanie Stroka[1] and Roland Ferstl[2]

[1] {firstname.surname}@salzburgresearch.at
Salzburg Research
Jakob Haringer Str. 5/3
5020 Salzburg
Austria

[2] roland.ferstl@siemens.com
Siemens AG (Siemens IT Solutions and Services)
Werner von Siemens-Platz 1
5020 Salzburg
Austria

Abstract. The growing impact of Semantic Wikis deduces the importance of finding a strategy to store textual articles, semantic metadata and management data. Due to their different characteristics, each data type requires a specialized storing system, as inappropriate storing reduces performance, robustness, flexibility and scalability. Hence, it is important to identify a sophisticated strategy for storing and synchronizing different types of data structures in a way they provide the best mix of the previously mentioned properties.

In this paper we compare fully structured, semi-structured and unstructured data and present their typical appliance. Moreover, we discuss how all data structures can be combined and stored for one application and consider three synchronization design alternatives to keep the distributed data storages consistent. Furthermore, we present the semantic wiki *KiWi*, which uses an RDF triplestore in combination with a relational database as basis for the persistence of data, and discuss its concrete implementation and design decisions.

1 Introduction

Although the promise of effective knowledge management has had the industry abuzz for well over a decade, the reality of available systems fails to meet the expectations. The EU-funded project *KiWi* - Knowledge in a Wiki project sets out to combine the wiki method of collaborative content creation with the technologies of the Semantic Web to bring knowledge management to the next level. Combining a Wiki with Semantic Web technologies results in three types of content:

Wiki Articles, which are basically unstructured textual content,
Management Data, like authors, creation dates and revisions, and

Semantic Metadata, which provide flexibility and spreading of the data about *KiWi*'s contents.

In this paper we explain the differences of data storage for these data types. We describe our choice of design and illustrate its usefulness for Semantic Social Software Applications. Furthermore, we explain how these three different approaches can be integrated in a single application, which is build with the Java Enterprise Edition (Java EE)[1] platform.

We present and discuss three different kinds of data: structured, unstructured and semi-structured. We discuss the weaknesses and strengths of each of them and describe that for a semantic social software application a combination of them brings advantages in form of an improved flexibility and performance. Furthermore, we describe several ways and designs how an application can implement the different ways of persisting data. The main challenge by developing an application which uses different data storages is to define a common interface for the access of data and to guarantee the synchronization of the different data storages.

Chapter 2 discusses the benefits, techniques and differences of structured, unstructured and semi-structured data. For this discussion examples for each paradigm are compared: An Apache Lucene full-text index for unstructured data, a relational database for fully structured data and an RDF triplestore for semi-structured data.

Chapter 3 describes several design patterns, which were used within *KiWi* to combine the three different approaches, focusing on the combination of relational databases and RDF triplestores. This chapter tries to answer the question which data should be stored where and discusses the decisions taken by the *KiWi* project.

Chapter 4 gives an overview of related work and chapter 5 summarizes the practical relevance of this approach.

2 Structured, Unstructured and Semi-Structured Data

In the semantic wiki *KiWi* we need all three kinds of data: structured, unstructured and semi-structured. This chapter presents and compares the different forms of data and gives examples and state-of-the-art techniques. Finally, a tabular overview of the different kinds of data structures is given.

2.1 Unstructured Data

According to[1], the term *unstructured* refers to the fact that no identifiable structure within this kind of data is available. Unstrucured data is also described as data, that cannot be stored in rows and columns in a relational database.

Storing data in an unstructured form without any defined data schema is a common way of filing information. An example for unstructured data is a document that is archived in a file folder. Other examples are videos and images.

[1] `http://java.sun.com/javaee/`

The advantage of unstructured data is, that no additional effort on its classification is necessary. A limitation of this kind of data is, that no controlled navigation within unstructured content is possible.

A common technology to search in unstructured text documents is full-text search. The advantage of full-text search is, that it is completely decoupled from the data. This makes it very flexible, because it can be used on every kind of textual data, even if no schema or structure is defined. One limitation of full-text search is that it cannot be used to search for pictures or videos.

Full-Text Search can be optimized by generating a full-text index, that increases the performance of a full-text search query. A famous full-text search engine library is *Apache Lucene*[2] . Other examples are MySql[3] and Postgres indixes[4].

2.2 Fully Structured Data

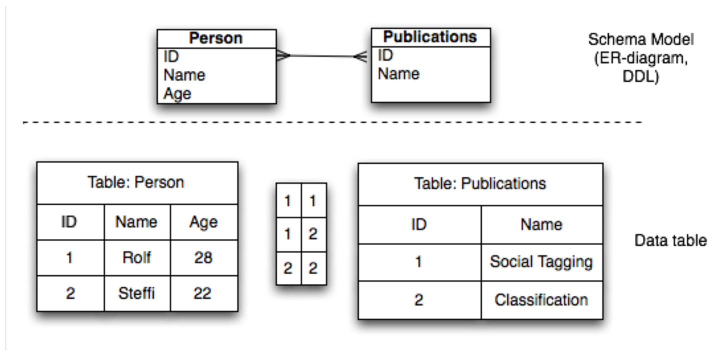

Table: Person

ID	Name	Age
1	Rolf	28
2	Steffi	22

1	1
1	2
2	2

Table: Publications

ID	Name
1	Social Tagging
2	Classification

Fig. 1: Sample Table in a Relational Database System

Fully structured data follows a predefined schema. *"An instance of such a schema is some data that conforms to this specification,"*[2]. A typical example for fully structured data is a *relational database system*. Designing a database schema is an elaborate process, because a schema has to be defined before the content is created and the database is populated. The schema defines the type and structure of data and its relations. Figure 1 illustrates an Entity Relationship diagram (ER-diagram) and its concrete tables within a RDBMS (relational database management system).

"The well-defined schema of fully structured data enables efficient data processing and an improved storage and navigation of content,"[2, page 122]. The

[2] `http://lucene.apache.org`
[3] `http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html`
[4] `http://www.postgresql.org/docs/7.4/static/indexes.html`

cost for high performance and navigation is flexibility and scalability. It is difficult to subsequently extend a previously defined database schema that already contains content. For example, it is not possible to extend a single table row with a new attribute without creating another table column. This is unprofitable for tables that contain thousands of other rows that do not need another attribute.

An advantage of relational database applications are the existing tools and web frameworks, which support the development of database-focused applications. For instance, *Hibernate*[5] and *Oracle TopLink*[6] are *Object/Relational* (O/R) *Mapping* frameworks, which map classes and objects to relational database tables and rows. Moreover, there exist several practical tools for maintenance, management and administration of relational database systems.

2.3 Semi-Structured Data

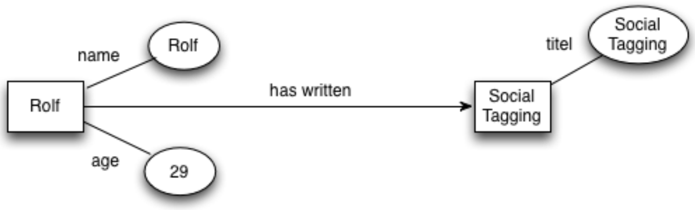

Fig. 2: Sample RDF Graph

Semi-structured data is often explained as "*...schemaless or self-describing, terms that indicate that there is no separate description of the type or structure of the data*"[2, page 11]. Semi-structured data does not require a schema definition. This does not mean that the definition of a schema is not possible, it is rather optional. The instances do also exist in the case that the schema changes. Furthermore, a schema can also be defined according to already existing instances (posteriori). The types of semi-structured data instances may be defined for a part of the data and it is also possible that a data instance has more than one type[2].

One of the strengths of semi-structured data is *"... the ability to accommodate variations in structure"*[2, page 12]. This means that data may be created according to a specification or *close* to a type. For instance, fields can be duplicated, data can be lacking or there may exist minor changes[2]. Figure 2 illustrates a graph representation of semistructured data. Figure 4 illustrates the same schema as in Figure 3, with the difference that the instance model has an additional property, which is not defined in the schema model.

[5] `http://www.hibernate.org/`
[6] `http://www.oracle.com/technology/products/ias/toplink/index.html`

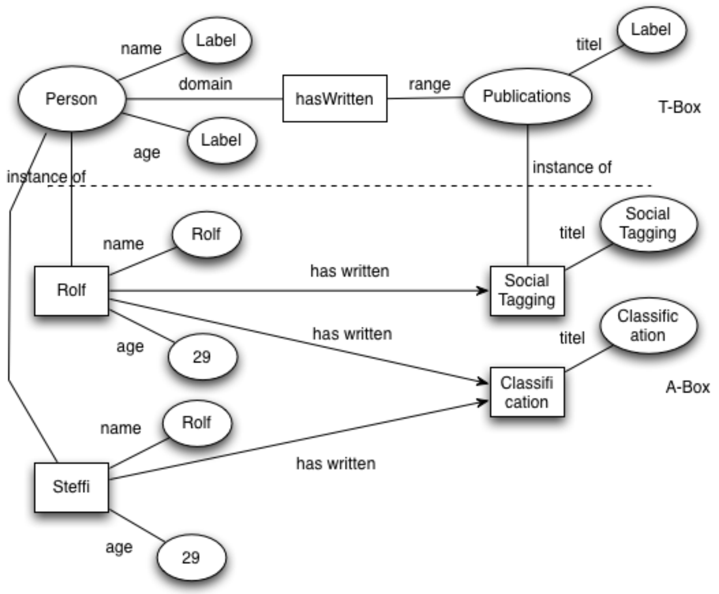

Fig. 3: RDF Schema (RDFS) and two instances

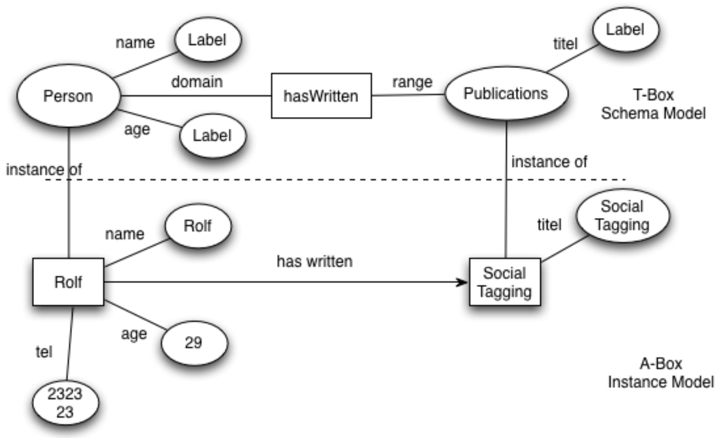

Fig. 4: RDFS and a flexible instance

A typical example of semi-structured data is XML, which is a language for data representation and exchange on the web. In XML data can be directly encoded and a *Document Type Definition* (DTD) or *XML Schema* (XMLS) may define the structure of the XML document[2].

In the research fields of the Semantic Web, knowledge is encoded in Resource Description Framework (RDF) triples[3], which store data in the form of subject, predicate and object nodes. The RDF Schema (RDFS)[4] vocabulary definition language allows the definition of classes and properties. In the World Wide Web RDF is used as a language that provides metadata to web resources.

2.4 Transformation of Data

In *KiWi*, data sometimes needs to be transformed from one structure into another. For instance, fully structured data is converted into unstructured data when a user generates a PDF out of a wiki article and its management data like author, creation date and so forth. It is also possible to convert data from a database into semi-structured data, like an RDF graph. Several modern web applications use RSS feeds , which are generated by reading data of a relational database and provide it in RDF format.

On the contrary, it is more complex to transform unstructured information into semi- or fully structured information. *KiWi* structures textual content with techniques of information extraction and natural language processing. Tags, which describe the content of a text, are automatically extracted out of a wiki article. In this way the unstructured data can be converted into semi-structured data.

2.5 Comparison and relevance for an application

It can be summarized, that the high degree of typing enables a better performance and less flexibility.

Serge Abiteboul, Peter Buneman and Dan Suciu define several reasons why defining a structure is good for[2]:

– to optimize query evaluation,
– to improve storage,
– to construct indexes,
– to describe the database content to the user and facilitate query formulation,
– to proscribe certain updates, and
– to support strongly typed languages.

Table 1 gives an overview over the strengths and weaknesses of the different storing structures in technology fields that may be important in practice.

2.6 Conceptual Federation of Relational Databases and Triplestores

To know how to combine a relational database and a triplestore we have to consider what data is stored where. Therefore, we review the strengths and

	Unstructured	**Fully Structured**	**Semi-Structured**
Technology	Character and binary data	Relational database tables	XML/RDF
Transaction Management	No transaction management, no concurrency	Matured transaction management, various concurrency techniques	Transaction management adapted from RDBMS, not matured
Version Management	Versioned as a whole	Versioning over tuples, rows, tables, etc.	Not very common, versioning over triples or graphs is possible
Flexibility	Very flexible, absence of schema	Schema-dependent, rigorous schema	Flexible, tolerant schema
Scalability	Very scalable	Scaling DB schema is difficult	Schema scaling is simple
Robustness	-	Very robust, enhancements since 30 years	New technology, not widely spread
Query-Performance	Only textual queries possible	Structured Query allows complex joins	Queries over anonymous nodes are possible

Table 1: Comparison of unstructured, fully structured and semi-structured content

weaknesses of different data structures and discuss the demand of structure characteristics for specific data sets. A relational database stores fully structured data, which necessarily have a predefined schema. Relational databases provide the application with a high query-performance and fast joins. Vulnerabilities are rare since more than 30 years of research, development and improvement eliminated most of them and increased the robustness.

Semi-structured data like RDF data does not have to predefine a schema and is very scalable and flexible. Furthermore, RDF and OWL[7] allow the definition of logical rules and many applications implement an inference layer that infers new triples by reasoning over the existing data set.

Thus, data that has a predefined schema, that is sensitive and that is often queried should be stored in a relational database. Data that is added to the application lately (e.g. data for extensions or plug-ins) and data that might be important for reasoning should be stored in the triplestore. Figure 5 provides a quick overview over the division into relational database data and triplestore data. As one can see, the data sets are partially overlapping.

[7] `http://www.w3.org/TR/owl-features/`

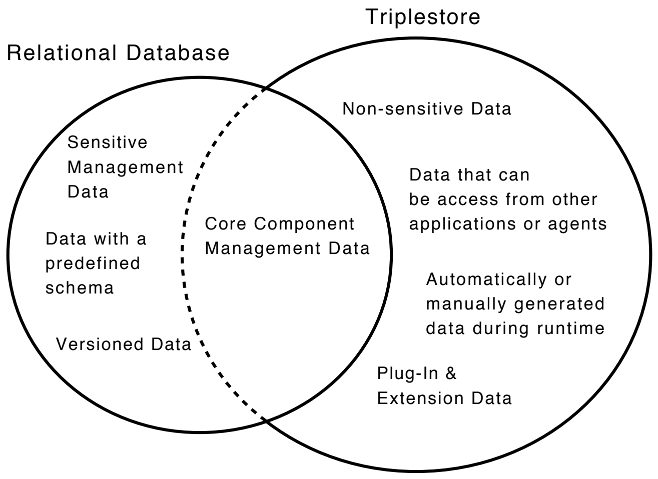
Triplestore
Relational Database
Non-sensitive Data
Sensitive Management Data
Data that can be access from other applications or agents
Data with a predefined schema
Core Component Management Data
Automatically or manually generated data during runtime
Versioned Data
Plug-In & Extension Data

Fig. 5: Overlapping data sets stored in the triplestore and in the relational database

3 Data representation in *KiWi*

Combining structured and unstructured data is an often applied strategy in web applications to achieve the advantages of both persistence types. The employment of all three alternatives, however, is uncommon.

KiWi is a platform for Semantic Social Software applications, implemented with Java EE technologies. We decided to store data in a semi-structured form, because we wanted to attain a better flexibility and scalability than provided by the structured form. We also wanted to store data in a robust database with good query and join performance. We have to control a big amount of textual content, which needs to be queried for keywords.

Hence, we decided to combine unstructured, structured and semi-structured data storage and segmented the data into long textual content (*unstructured*), core component data (*fully structured*) and flexible data (*semi-structured*). For a better clarity, Table 2 visualizes the segmentation. The sets of fully structured and semi-structured data are overlapping, because we represent the non-sensitive core data additionally in the triplestore to get a complete data set that can be provided to other Semantic Web Applications (e.g. Linked Data[8]).

3.1 Three possible Levels of Synchronization

Applications that store data in a triplestore as well as in a relational database have to implement a synchronization mechanism to keep information consistent. Such a synchronization mechanism can be implemented on different layers of an application.

[8] http://linkeddata.org

	Content Type	**Example**
Unstructured	Textual Content	Wiki Articles, Blog Pages
Fully Structured	Sensitive Content & System Maintenance Data Core Component Data	ContentItem, User data
Semi-Structured	Non-sensitive Core Component Data, Flexible Content & Individual Data	ContentItem-extending Data, Use Case Data

Table 2: Persistence alternatives and apportioned content

Database Layer Synchronization on the database layer is implemented by forcing a data storage (e.g. database) to update another data storage (e.g. triplestore) when a data item changed. For instance, every time an application writes on a database, the according operation could be executed on the triplestore, which might be hold in the database. This could be implemented using database triggers or Java EE persistence interceptors. Another possibility is that the triplestore is generated automatically from the entries within the database. Hence, the triplestore could be updated regularly. In both variants the database is defined as master and the triplestore is defined as slave. This design is illustrated in Figure 6.

This design benefits from high performance and good integration of relational databases into existing software technology stacks (e.g. Java EE). Furthermore, functions provided by a triplestore, like reasoning, are possible, because the data also exists in a semi-structured form. The disadvantage is that this design does not offer the flexibility of semi-structured data, and that the application has read only access to one data storage.

An alternative design is a bi-directional trigger synchronisation between relational database and triplestore. The triplestore, as well as the relational database can update each other with database triggers. The advantage of this design is that it allows writing access to both data storages. This design is illustrated in Figure 7. The limitation is, that some updates on the triplestore cannot be processed on the database and must be forbidden to keep consistency. Therefore, this design does not support the full flexibility of semi-structured data, too.

O/R Mapping Tool O/R mapping tools provide another layer for synchronisation. This design is illustrated in Figure 8. For instance, the Java Persistence API (JPA)[9] could be extended to persist Java objects in the database as well as in the triplestore. This encloses the translation of JpaQL (JavaPersistenceApiQueryLanguage)[10] queries into triplestore queries. This approach decouples the

[9] http://java.sun.com/developer/technicalArticles/J2EE/jpa/
[10] http://java.sun.com/javaee/5/docs/tutorial/doc/bnbtg.html

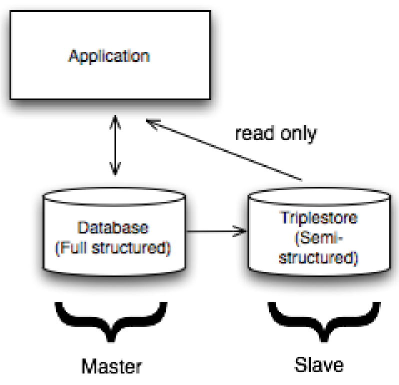

Fig. 6: Database defined as master and triplestore defined as slave

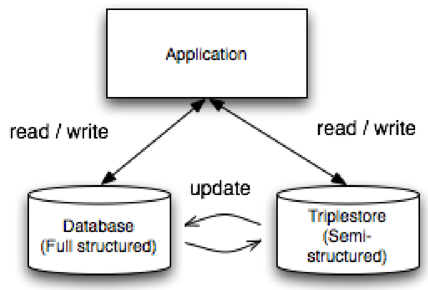

Fig. 7: Triplestore and database update each other

persistence layer from the application layer, and, therefore, provides the flexibility of semi-structured data. Thus, additional attributes of an object may be defined during the runtime of an application and persisted in a triplestore. This may be realized using Aspect Oriented Programming (AOP)[11] techniques or dynamic languages like Groovy[12]. With this approach, distributed queries over several datasources could be realized.

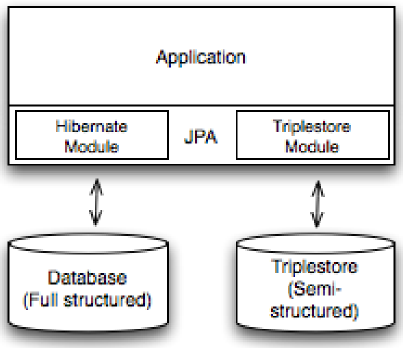

Fig. 8: Extension of the JPA with a triplestore module to guarantee consistency

[11] http://www.eclipse.org/aspectj/
[12] http://groovy.codehaus.org/

Application Layer / Middleware Layer Another alternative to guarantee the synchronisation of data is to implement it in the middleware or application layer. This layer could use normal JpaQL queries for the database as well as SPARQL commands to query the triplestore. This design is illustrated in Figure 9. A different alternative is to provide a general purpose query language for both data stores. In this way, distributed reasoning over the triplestore, as well as over the data in the relational database system could be enabled. This design is illustrated in Figure 10.

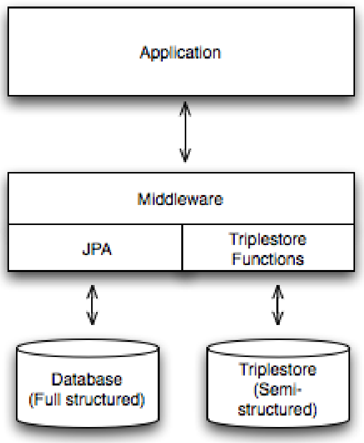

Fig. 9: Middleware layer which handles the persistence of data

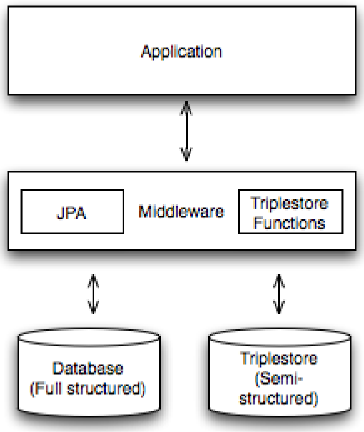

Fig. 10: General purpose query language

3.2 Integration of a triplestore in the Java EE stack

We decided to choose the Application Layer for synchronization, because it grants us flexibility to improve weaknesses and to enforce the strengths of each data structure type. First, we will give you an overview over the triplestore position inside of *KiWi*.

Figure 11 illustrates the overall structure of *KiWi*. The combination of triplestore and relational database can be found in the *Persistence* and *Data Model* layers. As an RDF triplestore *KiWi* currently uses *Sesame2*[13]. The relational database connection is enabled through Hibernate with JPA. Storage configurations for relational database and triplestore can be applied with Java annotations.

[13] http://www.openrdf.org/

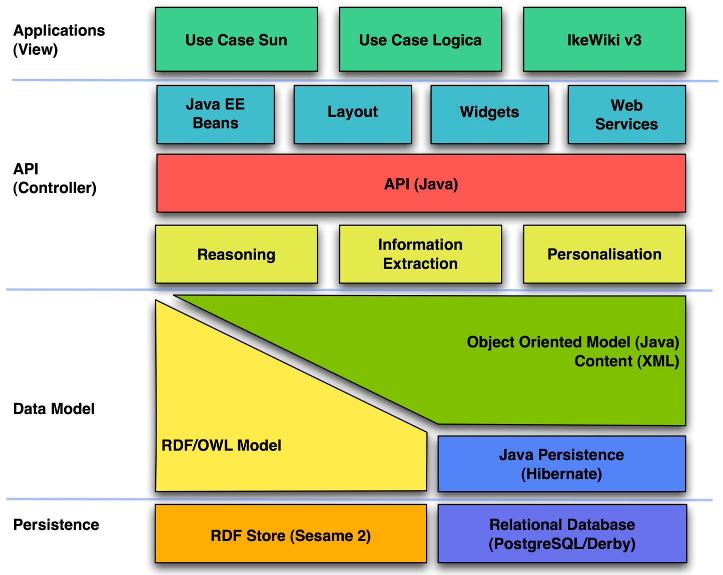

Fig. 11: KiWi's overall structure, adapted from[5]

Transactional synchronization As Table 1 illustrated, transaction management for unstructured and semi-structured data is not very common or matured. Though, storing data in those federated, heterogeneous databases needs to be controlled to avoid states of inconsistency. A global transaction management is the easiest way to administer all three data structure types in terms of their transactions. *JBoss Seam*[6], *Hibernate/JPA*[7], and *Enterprise Java Beans* (EJB)[8] provide us with diverse techniques to control transactions programmatically and declaratively, for example:

Java Transaction API, also called JTA[14] specifies standard Java interfaces for Java Enterprise Applications implemented by the application server[9].

Seam Transactions extend JTA UserTransactions with useful functionality, for example the registration of a synchronization implementation[6].

EntityManager Transactions are provided by Hibernate/JPA for programmatic transaction management to start and stop transactions explicitly[10].

Programmatic transaction processing requires the definition of a start and end time for the transaction. It allows flexible pre- and post-treatment of the application when the transaction ends. Declarative transaction processing, on the other hand, is simpler than programmatic transaction management, because the transaction start and end time is managed by the container[11]. To control the behaviour before and after a transaction ends in applications using declarative

[14] http://java.sun.com/javaee/technologies/jta/index.jsp

transaction processing, a synchronization implementation can be registered[9]. In *KiWi* we use the *before-completion phase* to synchronize the relational database state with the triplestore state. Thus, updates to both databases will be executed simultaneously at the end of a transaction. Figure 12 illustrates the process. If an update fails, the whole transaction including changes on both databases will be rolled back.

A more detailed description of the transaction models in Java Enterprise Applications, the concurrency problems that triplestores must consider and the database synchronization is given in [12].

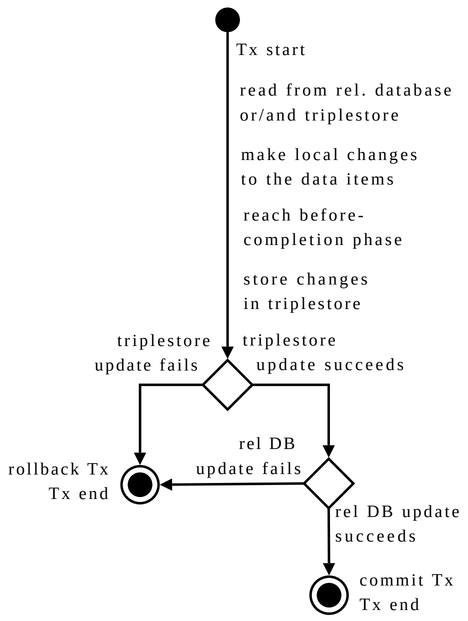
Tx start
read from rel. database or/and triplestore
make local changes to the data items
reach before-completion phase
store changes in triplestore
triplestore update fails
triplestore update succeeds
rel DB update fails
rollback Tx
Tx end
rel DB update succeeds
commit Tx
Tx end

Fig. 12: Transactional synchronization process

Data Versioning Versioning of unstructured, semi-structured and fully structured data is an important core functionality of *KiWi*. RDF triple versioning is uncommon and few well-established RDF repositories allow versioning. Sesame2 puts RDF triples internally under version control, but it does not enable undo or redo functions.

With the chosen transaction strategy we can easily implement version-control of unstructured, semi-structured and fully structured data. At the end of a transaction, updates for all kinds of data are creates and stored as revisioning and update tables in the relational database. This design was chosen to collect all versioning data in a robust database, to enable easy querying, and, consequently, to allow fast undo and redo functionality for all kinds of data. Versioning data has a pre-defined schema that will not be changed in the future.

Query & Reasoning With the chosen level of synchronization it is possible to create a query language for all kinds of data. *KiWi* enables this global querying that interprets to *SQL* and *SPARQL*[15] queries. Furthermore, reasoning is not limited to the RDF repository anymore. The interested reader is referred to [13] for a more detailed discussion about this issue.

4 Related Work

In the following we provide an overview over implementations of semi-structured data into existing application stacks. *Elmo*[14] is a Java library for Semantic Web applications that maps Java classes to RDFS/OWL classes. Another implementation of a server which offers access to different representations of data is *Virtuoso*, *"... which is a database engine hybrid that combines the functionality of a traditional RDBMS, ORDBMS, virtual database, RDF, XML, free-text, Web Application Server and File Server functionality in a single server product"*[15].

5 Conclusion

The main advantage of fully structured data is the strong typing which enables high performance and efficiency. On the other hand, unstructured and semi-structured data allow a higher degree of flexibility. In this paper we compared unstructured, semi-structured and fully structured information and discussed an application design which combines all three types of data, based on a relational database system combined with an RDF triplestore. We illustrated this design on the concrete implementation of the semantic wiki *KiWi*. We saw that a challenge for such an application is to avoid states of inconsistency and present three different layers where a synchronisation of data within an application could be implemented:

1 On a low level database layer,
2 On the he O/R mapping layer, and
3 On the application layer.

In *KiWi* the synchronisation of data is implemented on the application layer because it offers database independence and enables the implementation of a common query language for all different data stores.

[15] http://www.w3.org/TR/rdf-sparql-query/

References

1. Blumberg, R., Atre, S.: The Problem with Unstructured Data. `http://www.dmreview.com/issues/20030201/6287-1.html` (19.02.2009) (2003)
2. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: from relations to semistructured data and XML. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA (1999)
3. Manola, F., Miller, E.: Resource Description Framework (RDF):Concepts and Abstract Syntax. `http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/` (19.02.2009) (2004)
4. Brickley, D., Guha, R.: RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/` (19.02.2009) (2004)
5. Schaffert, S., Sint, R., Grünwald, S., Stroka, S.: The KiWi Architecture. (2008)
6. Allen, D.: Seam in Action. Manning Publications Co. Greenwich, CT, USA (2008)
7. Bauer, C.: Java Persistence with Hibernate. Manning Publications Co. Greenwich, CT, USA (2006)
8. DeMichiel, L., Keith, M.: JSR 220: Enterprise JavaBeansTM,Version 3.0. `http://java.sun.com/products/ejb/docs.html` (20.02.2009) (2006)
9. Cheung, S., Matena, V.: Java Transaction API (JTA). `http://java.sun.com/javaee/technologies/jta/index.jsphttp://java.sun.com/javaee/technologies/jta/index.jsp` (19.02.2009) (2002)
10. : javax.persistence.EntityTransaction Interface JavaDoc. `http://java.sun.com/javaee/5/docs/api/javax/persistence/EntityTransaction.html` (11.02.2009) (unknown)
11. Connolly, T., Begg, C.: Database Systems: A Practical Approach to Design, Implementation, and Management. Addison Wesley Publishing Company (2005)
12. Stroka, S.: Transaction Management in Federated, Heterogeneous Database Systems for Semantic Social Software Applications. (2009)
13. Francois Bry, Michael Eckert, J.K., Weiand, K.: What the User interacts with: Reflections On Conceptual Models For Semantic Wikis. (2009)
14. Leigh, J.: Elmo User Guide. `http://www.openrdf.org/doc/elmo/1.4/user-guide/index.html` (19.02.2009) (2008)
15. Virtuoso: Virtuoso Universal Server. `http://virtuoso.openlinksw.com` (2009)