Design and Prototypical Implementation of an Integrated Graph-Based Conceptual Data Model

Matthias Sedlmeier, Martin Gogolla

Abstract. The paper introduces a new, comprehensive integrated conceptual data model in a precise way. Central language features cover core modeling concepts as classification, membership, inheritance, interfaces, structured types, aliasing, aggregation, composition, constraints, clustering and relationships. Schema states are thought of as being realized as graphs with appropriate navigation options. A prototypical graph database implementation and accompanying examples are discussed.

Keywords. conceptual data model, graph based storage, graph database, conceptual modeling

1. Introduction

Information technology depicts a fundamental component of modern organizations in areas like administration, service and production. Well working electronic data processing is able to raise the efficiency of administration and business processes [11] by supporting employees accomplishing their tasks. Poorly designed information systems however may disturb work flows and lead to unmotivated labor [23]. The design of a good information system is challenging, because its quality depends heavily on the sound analysis of the requirements [33]. These are normally specified in cooperation with the representatives of all later end user groups to get a comprehensive impression of the needs.

An information system should be able to cover the information needs of a particular organization. Therefore, it must have the ability to store structured data, to calculate and derive new data and output data in a human readable way. One can assume, that the need of certain information is always linked to a specific task [18]. As information systems support specific tasks varying from organization to organization, their requirements also differ. The exact requirements are usually determined in the form of a data model holding a precise representation of the domain.

The ability of human intellect to abstract from reality establishes the fundament for designing data models. The process of abstraction implies the concentration on designated characteristics of a natural object and at the same time the abandonment of detailed examination of all recognizable attributes. Abstraction is hence an act of simplification or generalization. The design of a good data model demands an accurate analysis of the application domain in such an extent that the correct subset of all observed aspects is captured and represented appropriately.

To represent real domain aspects, one may use graphical language elements, which stand for objects, attributes and relationships and therefore receive a semantic denotation. Specific rules of combining and linking those graphical artifacts are defined by a corresponding syntax. In addition to the representation of natural considered objects, data models also describe ideal constructs such as *divorce*. Such standardization of a modeling language gives designers the ability to create models, which can be understood by anyone knowing how to interpret the language. The definition of a syntax also enables designers to apply formal model modifications as well as model transformations automatically. The possibility to create new models by transformation respectively translation is crucial for the design process of an information system, where each development stage claims models of different abstraction levels.

The controlled storage of data is usually done with the aid of a special software often denoted by *database system*. A database system consists of one component responsible for the physical data storage and further components dealing with data processing and access control [8]. The first component is also known as *database* and the second component is denoted as *database management system*. The design of a database as part of an information systems is divided into multiple phases, whose results end in corresponding documents. Depending on the current design stage, the domain artifacts are represented in different models, which vary in their way of expressing domain issues like structured types or relationships.

In the course of creating the requirements definition, one initially elaborates a conceptual model, which is also called a conceptual database schema. This model is typically worked out as an ER [7] or UML [31] diagram and serves as background for a logical database schema, which regards the concrete operating mode of the database model. The last step contains the translation of the logical database schema into a physical model.

The most frequently used database model is the relational one, which uses tables to represent the elements of the conceptual schema. But modeling domain aspects via tables often proves itself as complicated and insufficient, because tables do not allow the *direct* representation of domain facets. A solution to face these problems presents the design of an integrated semantic data model as a synthesis between conceptual, logical and physical model. For this approach, parts of the language features of the ER respectively EER [30] model and the UML class model are used. Moreover, a main memory resistant graph-based storage approach is chosen to make data records persistent. Abiteboul and Hull follow a similar idea in their work about the design of a semantic database model [1]. The introduced integrated semantic data model is not just another conceptual data modeling language. It is also considered as a type graph, from which database states can be directly derived by establishing instance graphs. A similar approach is used in [2], while [21] discusses search algorithms for conceptual graph databases.

These can also be found in medical computing [10], data warehousing [19], bioinformatics [12] and geographic information systems [24]. A comparison between graph databases is presented in [16] and [20], which lays the focus on performance. In [3] another comparison of current graph databases is provided. Graph databases are also subject of researches related to big data [25] and data mining [28].

The rest of this paper is structured in 5 sections. In section 2, some challenges are discussed. Section 3 introduces the designed integrated conceptual language called TEGeL. Section number 4 deals with the implemented database prototype called Lhasa

DB. And in section 5, a short evaluation is given. The paper closes with a conclusion and some ideas for future work in section 6.

2. Challenges

2.1. Requirements Analysis

The requirements definition is an important document generated in an early phase of the software development process. If an information system is designed, a lot of requirements are related to the question, which data must be stored in which way. Therefore, the first step contains a sound analysis of the domain. This task can be conducted by using graphical models prior to textual descriptions, which usually express the requirements in a more clearly and easy way. To cover every important aspect, everyone using the system later on, has to proclaim his information needs. In most cases these potential users are neither software developers nor do they have a technical background. Upon this, one must emanate from the fact, that all kinds of different careers have its own special requirements, which must be formulated in one comprehensible model. Therefore the chosen language must ensure, that everyone is able to understand its syntax and semantic. If this goal is reached, the conceptual model can be applied as a communication basis between developers and users [6].

As the practice shows, only 30 percent of all users are aware of their needs. Over 40 percent of their requirements are indeed known and action determining, but not directly accessible and though cannot be formulated. Another 30 percent of all requirements are subconscious and must be triggered from the outside [27]. This fact must be kept in mind when choosing a good conceptual language, which must be understood easily while leaving room for frequent changes. Requirements can also vary systematically due to the selected development process. If an agile approach is given, frequent increments of the requirements definition are part of the development principle. Therefore, a good conceptual model has to support agile design cycles.

As we can see, requirement changes are not avoidable. In practice, they are the order of the day and ultimately part of agile development principles. Under these circumstances it is necessary to ensure, that alteration costs are minimized by including all changes as early and straightforward as possible. Alteration costs can increase quickly: once the system is under design, the costs of changes raise by a factor of 3 and during its development by a factor of 7. If there are requirements changes while testing, the alteration costs increase by a factor of 50 and after delivery by a factor of 100 [26].

2.2. Problems of Relational Modeling

The data management component of an information system usually consists of a relational database, which is designed in three phases. The first phase includes the abstraction of the application domain in form of a conceptual model. This conceptual model is then transformed to a logical model, from which the physical model is derived. The conceptual model is often implemented as ER respectively UML diagram, which is able to represent facts like relationships directly. When it comes to relational databases, the conceptual model is used to derive a relational schema using tables with appropriate

columns. Further steps possibly contain normalization and synthesis. In the last phase this relational abstraction is taken and translated to corresponding SQL statements.

At the end of the pictured process, there are at least three models in different languages describing a single application domain. If any of these models is changed, all other models must be updated to prevent inconsistent representations. On the one hand, the correctness of the conceptual model is important, because it is the communication basis between all stakeholders. On the other hand, the derived models should always be synchronized with the conceptual model to represent the application domain accurately. It must be pointed out here, that it is always technically possible to manipulate every model in itself without having adopted those changes automatically to the remaining models. One explicitly must take care about the semantic cohesion between all models after every change. This synchronization contains continuous model transformation in different languages and is possibly lossy, if there is no direct mapping between the used language features. For this reason, a transformation may also not be reversible impeding the whole synchronization process.

As said before, the conceptual model plays an important role with regard to stakeholder communication. That is why it is created as an ER or UML diagram, which is much more comprehensible for technical lays than relational schemata. This is connected with the visual representation of domain facets and the kind of expressing relationships. In ER or UML models, these are illustrated by edges, which can be seen as the most direct semantic mapping possible.

The representation of relationships by tables falls far short of this simplicity, because one must use foreign keys and join tables to express the connection of two or more objects. This indirect description does not reflect the fact, that the connection of domain objects rather correspond to graph structures as it is clearly apparent in the conceptual model. This representation problem arises from the fact, that tables are flat and thus not able to express links and structures in a native way. Further problems occurring from this deficit of straightforward mapping are discussed now.

Domain objects usually have attributes for detailed specification. If those attributes are flat, they can be represented through suitable named table columns. Structured attributes however shape hierarchies, which can only be modeled indirectly using the relational join. This way must also be gone, if there are multiple attributes with an arbitrary maximum cardinality. Another problem depicts the Object-Relational Impedance Mismatch [5], which occurs using object-oriented business logic. Data structures, which were flattened for relational storage, now must be recovered to their structured versions. After processing, the data records are flattened again and so forth. This continuous translation induces additional computing costs. The mapping of relational data to object structures is often realized by a object-relational mapper, which can be added to an information system as further layer. Here, one must bear in mind, that using an object-relational mapper introduces an additional need for synchronization, because the business logic always works on a copy of the original data. Another problem is related to the concept of inheritance, which can be simulated with patterns like Class Table Inheritance, Concrete Table Inheritance or Single Table Inheritance [9], but which is not a native component and therefore offers no direct representation.

3. Introducing TEGeL

The controlled storage of data requires a formal model revealing which data must be saved, which connections exist and which constraints must be considered when database states are established. The following sections describe such a model as a *type graph*.

The central element of a type graph depicts the *entity type* as an abstraction of an object or idea. An entity type allows the instantiation of *entities*, which reside in a potential database state with a proper existence. Entity types can be categorized by *entity type classes*, which is expressed by a *classification relation*. An instantiation of entity type classes is not possible, hence they are marked as *virtual*.

Entity types are described in detail by *annotation types*, which represent attributes. Entity types and annotation types are connected by *membership relations*. Annotation types itself can equally be connected by membership relations, if structured attributes are modeled. Instantiated entity types, thus entities, hold accordingly instantiated annotation types called *annotations*. There are different *modes* of annotation types, which will be discussed later.

Besides the *reuse* of annotation types by multiple entity types, the inheritance of attributes is provided by introducing the concept of a *super entity type*. It equals an entity type except for the fact, that an instantiation is not possible. Super entity types may inherit from multiple other super entity types, which is realized by an *inheritance relation*. In addition to inheritance, one can use the concept of *entity type interfaces*, which also delivers extra attributes for entity respectively super entity types. The connection is established by an *implementation relation*. Entity type interfaces are not able to inherit from super entity types or to implement other entity type interfaces. The *clustering relation* enables the designer furthermore to categorize entity types, entity type classes, super entity types and entity type interfaces in named *clusters*.

The type graph does not only contains type information, but also rules for relations and constraints. The integrity rules are generally denoted as *integrity constraints* and decompose in *value constraints* for data type restrictions and *structural constraints* for structural requirements. Modeling relationships is also possible by using *relationship types* in combination with a special annotation type mode called *aggregation*. In a database state, relationship types become *relationships* interconnected with *aggregation annotations*.

The introduced type graph represents an integrated conceptual language, which will be called Tibet Entity Graph Language (TEGeL) from now on. Specific nodes and edges corresponding to the mentioned concepts above will be described and illustrated[1] in the following sections.

3.1. Classification and Membership

Figure 1 introduces the classification as well as the *default* membership concept for (reused) *annotation type nodes* and additionally shows comparable ER and UML models. The given schema models a kind of water sports world with two domain concepts, which are represented as entity type nodes: notably `Lido` and `SportsBath`. These *entity type nodes* are grouped together by an *entity type class node* named `SportsFacility` and share two annotation type nodes `Name` and `Location`, which represent *plain* attributes.

[1] All example schemata were created with the yEd Graph Editor.

These annotation type nodes are subject to integrity constraints as their possible instance values may only be elements of the string data type. The connection between entity type and annotation type nodes is generally established through a *membership edge*, which has different modes. In this case, the membership is qualified as *required* and *default*. Other modes are discussed later.

Sharing or reusing annotation type somehow breaks with the concept of encapsulation, but it provides an advantage towards integrity, because one can now assume, that an attribute with equal naming (irrespective of the local context) describes the same application domain artifact. The redundant definition of annotation types and their integrity constraints may thus be prevented.

Both entity types have distinct attributes named `NoSlides`, for the number of slides, `ChildrensPool`, which is set to true, if there is a pool for children and `StopWatch`, which is set to true, if there is a stopwatch to take the lap times. These attributes are also restricted when it comes to value assignment. There are two orders of value constraints: the data type is a *first order value restriction*, but it is possible to define *second order value restrictions* in form of simple conditions or regular expressions as it is apparent for the annotation type nodes `NoSlides` respectively `Name`. TEGeL is planned to support a subset of the ordinary atomic XML Schema [22] data types including *string*, *decimal*, *integer*, *float*, *boolean*, *date* and *time*. Sequences, sets and structured types can be directly modeled via TEGeL.

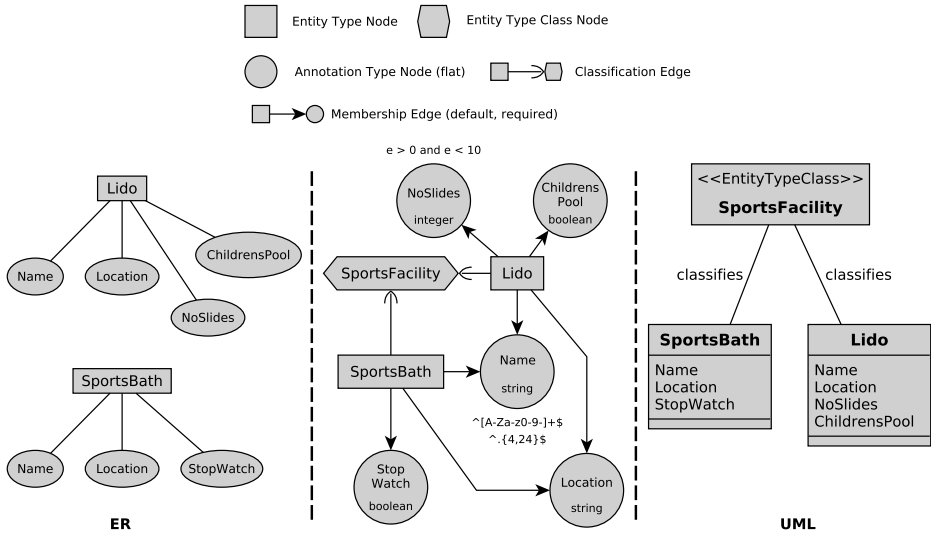

Figure 1. Classification and Membership

Entity type nodes are drawn as rectangles with sharp edges, whereas entity type class nodes are represented by hexagons. The displayed annotation type nodes are in plain mode, so they are drawn as circles. Each node must at least have a unique identifier and may carry additional information, called *decorations*. In the given example, the indication of first and second order value restrictions can be considered as decorations as well as the data type indication. Custom decorations may also be sensible, but will not

be discussed here. As one can see later, also edges hold decorations, which are called *modifiers*.

The membership relation between both entity types and their annotation type nodes can be directly drawn as edges with normal arrow heads directed from the entity types towards their attributes. The classification edges towards the entity type class node `SportsFacility` are also solid, but show open concave arrow heads.

3.2. Inheritance, Interfaces and Structured Types

Figure 2 again models a kind of sports world and introduces the concept of inheritance. The already known entity type nodes `SportsBath` and `Lido` inherit all annotation type nodes of the super entity type `RecreationFacility`, namely `Name`, `Location` and `Admission`. The first two attributes are already familiar from the above example schema, where they were modeled as *reused* annotation type nodes. A *super entity type node* is drawn as a trapezoid connecting entity type nodes by *inheritance edges* with a white triangle arrow head. The third annotation type node is a structured attribute called `Admission` holding two plain annotation type nodes `AdmissionType` and `Amount`. Annotation types in *structured mode* are drawn in rectangles with round corners and have no proper data types as they themselves build new ones.

Next to the `Admission` annotation type node, there is another structured attribute, which forms a new data type representing a `Pool`. As we can see, *nested* structured types can be composed of other structured types forming a hierarchy. Regarding the membership between `SportsBath` and `Dimension`, one can also see, that structured annotation type nodes can be reused, even if they are part of a substructure. Furthermore, we see, that membership edges can be *optional* if they are drawn in a dashed way. If annotation types have an optional membership, they are allowed to be absent when instances are built.

Another point is the indication of a *membership cardinality*, like it is shown for the attribute `Admission`. Membership cardinalities provide a mechanism to control the quantity of established attribute instances and induce a *multi membership*. Together with the indication of optionality, it represents another way of defining integrity constraints.

Additionally, functional dependencies can be established by declaring memberships as *unique* as it is done with the plain annotation type node `GeoCoordinates`. If a multiple attribute membership edge (one with cardinality indication) is modified as unique, the whole collection of attributes instances has to be unique concerning the element values as well as their order (if it is not a set). Multiple attributes can also be attached as *distinct*, which means, that every element within the collection must be unique.

A further important concept depicts the entity type interface. It is applied to be sure, that entity types hold certain attributes, which are necessary for the processing of derived instances. Expressing interfaces by modeling corresponding entity super types is, although technically possible, semantically misleading, because a super entity type usually represents the same domain object on a higher abstraction layer. This is also the reason, why entity types interfaces neither implement other entity type interfaces nor inherit from any super entity types. Conversely, super entity types possibly fulfill interfaces as it is shown on the example schema, where `RecreationFacility` is connected to entity type interface `Visitable` by an *implementation edge* with a dot arrow head. The *entity type interface node* itself is drawn as an upside down trapezoid.

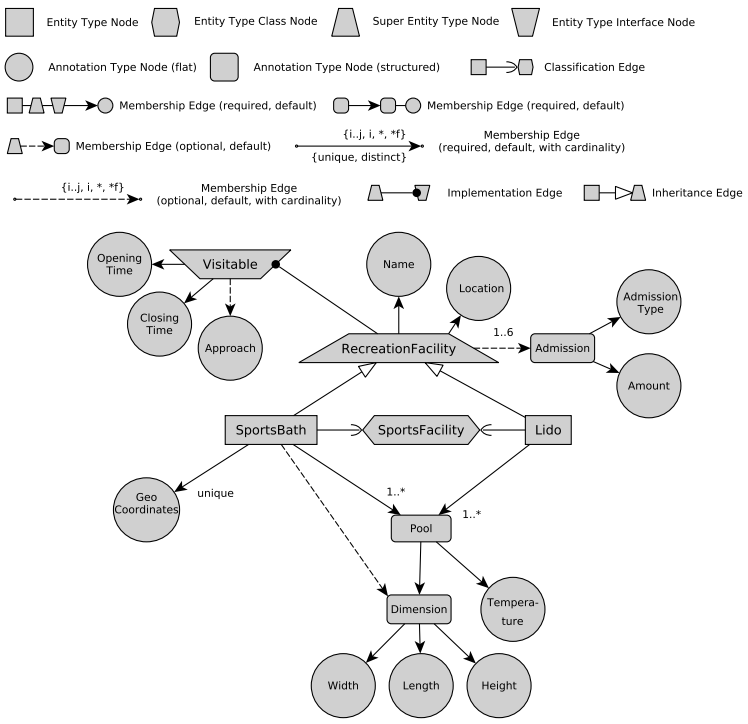

Figure 2. Inheritance, Interfaces and Structured Types

3.3. Aliasing, Constraint Groups, Aggregation and Clustering

Figure 3 introduces *alias annotation type nodes*, which are drawn as diamonds. By means of those attributes, one can use existing annotation type nodes while renaming them. The example shows a possible application for this linguistic feature. Both alias annotation type nodes `BeginDate` and `EndDate` only have a slightly different semantic notion. Here, this feature realizes a fine specialization upon the `Date` attribute, whose values additionally must comply with the condition $e > 01/01/1980$, where e is a possible concrete date.

So far, one can only make statements about the required or optional existence of attributes. The introduction of *constraint group annotation types* allows the definition of more precise conditions. Assume that information about sporting events has to be stored. The example schema shows a possible variant by modeling an entity type node called `SportingEvent` with a plain attribute `Title`. There is also a member called `Venue` representing a constraint group annotation type node, which is connected by a *restriction membership edge* with an opened convex arrow head. This constellation demands, that, when a database state is constituted, there can either be a connection to the aggregation annotation node (which will be discussed later) `SportsHall` or to `Stadium` as the inbound edge modifier `XOR` indicates. The constraint group annotation type node itself is drawn as a parallelogram. Besides the logical operators `OR`, `AND` and `XOR` also relational operators like $>, <, =, >=, <=$ and $!=$ can be used, if the member data types provide support.

Till now, the presented concepts do not allow to correlate entity types. This serious restraint is cured with the introduction of *aggregation annotation type nodes*. In the field of data modeling, relationships are often denoted as associations and usually allow the connection between one, two or more objects. Thereby, the exact semantic of those associations is not fixed. In practice, one must frequently express, that objects have a whole-part connection to model hierarchical structures or to formulate existence conditions. For the first case, the *aggregation* can be applied representing a specialization of an association. This relation is usually directed, so that there exists a whole, which is here called *aggregate* and subordinated parts named *components*. There is no existential binding between the aggregate and its components. If the aggregate's life cycle terminates, the components are released from the relation and persist.

For the second case, the concept of *composition* comes into question, which enables the designer to connect entity types existentially. Compared to aggregation, the existence of all components are annihilated, if the aggregate is deleted. The composition can be also seen as a specialization of the association. Following the naming conventions of UML, the aggregation mode is either *shared* for the representation of aggregation or *composite* for compositions.

In the previous example schema, the pool artifact was designed as a structured annotation type node, which made it an integral part of a lido respectively sports bath. If we want to model the pool as an independent concept, the composition comes into play and the pool is then modeled as an entity type node. We express the fact, that a pool should be bound to the existence of its lido or sports bath and won't be shared between multiple aggregates. The entity type `Pool` is connected to `Lido` by an aggregation annotation type node, which itself is linked to the `Lido` entity type node through a specialized *composition membership edge*, which shows a black diamond as arrow head. The edge direction is knowingly reversed in contrast to the UML notation. The cardinality indication shows, how many pool instances may be composed. The `SportsBath` entity type is also connected to the pool artifact by another aggregation node.

Additionally, one can determine a shared aggregation between `Lido` and `RubberMonster`, which is also established by an aggregation node through a *shared membership edge* drawn with a white diamond arrow head. This connection depicts, that a lido may offer a rubber monster, which is not existentially bounded. All aggregation annotation type nodes are linked to corresponding entity type nodes by `aggregation edges` equipped with crow arrow heads.

Monomorphic aggregation respectively composition is very inflexible, because it only allows to connect instances holding exactly the specified type. For this reason, TEGeL permits directing aggregation edges not only to entity types, but also to all other concepts except for annotation types. This feature is here called *polymorphic aggregation*. Aggregation annotation types also have identifiers, which can be seen as *target role names* for the attached entity types offering an additional navigation possibility.

Another language feature are *cluster nodes*, which enable designers to categorize entity types, super entity types, entity type interfaces as well as other clusters. A very limited example is given in the example schema. The `Attraction` cluster node connects the entity type node `SportingEvent` as well as `InflatableMonster` and the entity type interface node `Visitable`, which means, that every other type implementing this interface is also part of the cluster. In this case, `Lido` as well as `SportsBath` are transitive elements of `Attraction` via the `RecreationFacility` super type node.

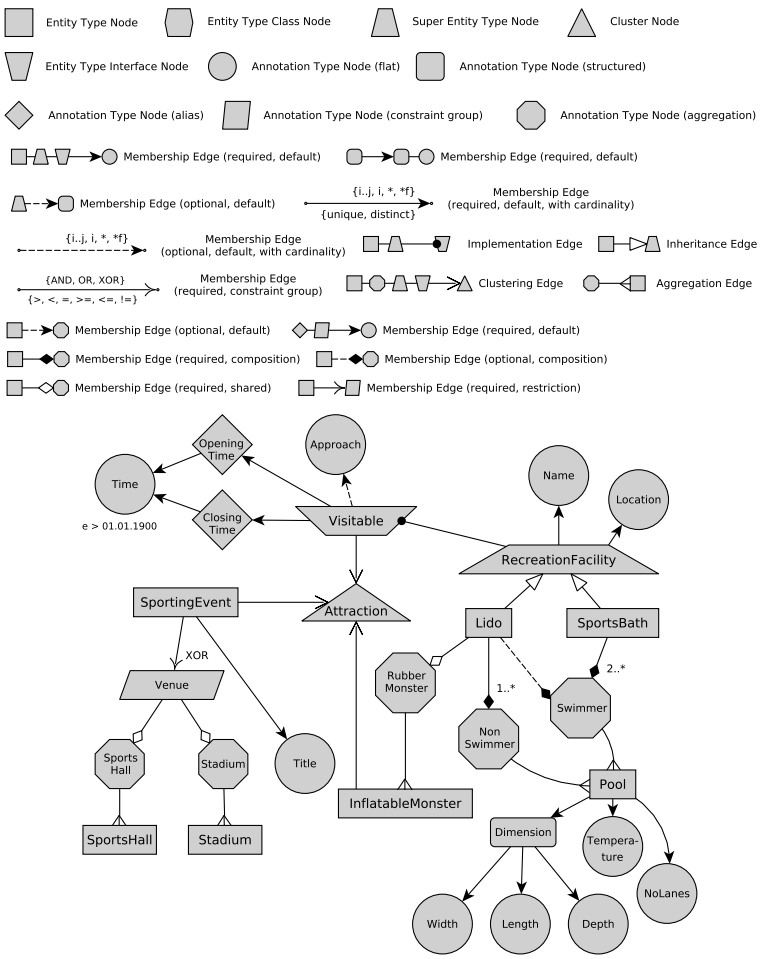

Figure 3. Aliasing, Constraint Groups, Aggregation and Clustering

3.4. Relationships and Hyper Relationships

Relationship types enable the designer to establish associations between entity types respectively relationship types. Relationship types are handled as specialization of entity types. In the context of a schema definition, one can specify which entity types respectively relationships types are connected and which quantitative participation constraints they must comply. Therefore the membership cardinality as well as the *participation cardinality* is used.

If cardinalities are indicated, one must differ, if relationships or entities are counted. The participation cardinality restricts, how often an entity takes part in a specific relationship. Conversely, the membership cardinality counts how many entity instances participate in an aggregation. So, one can basically count (a) the number of participations of entities in relationships (participation cardinality) or (b) the entities participating in an aggregation (membership cardinality).

Besides the indication of cardinalities, relationships have an arity defining, how many entity or relationship types are participating at all. Usually, relationships are binary, but they can basically have any arity greater than zero. Additionally, relationships are distinguished between their orders. First order relationships only connect entity types, while higher order relationships also connect first order relationships. In this paper, these higher order relationships described in [4] are named *hyper relationships*.

Usually, three kinds of binary relationships are distinguished. Consider *A* and *B* as entity sets. The 1:1 relationship describes a *one-to-one* mapping between an element of *A* and an element of *B*. The 1:n relationship specifies a *one-to-many* mapping, where one element of *A* has a connection to *n* elements of *B*, but an element from *B* relates only to one element of *A*. Finally, there is the n:m relationship, where *n* elements of *A* can be connected with *n* elements of *B* and vice versa.

Figure 4 shows three binary first order relationship nodes called `Membership`, `Offer` and `Execution` as well as one binary hyper relationship node named `Training` and finally one ternary first order relationship node denoted as `Contact`.

The relationship type `Membership` models the fact, that an athlete is a member in a gym. Given the membership and participation cardinalities, we can make the following statements: Firstly, one `Gym` instance has between 100 and 1000 members. Secondly, one athlete has any number of gyms, at which he is enrolled. Thirdly, one `Membership` instance holds an arbitrary number of gyms and fourthly, one `Membership` instance holds an arbitrary number of athletes in *fixed mode*. The fixed mode means, that for every new `Athlete` instance a new `Membership` instance is created to preserve ambiguity. Therefore, the left side membership cardinality of `Athlete` is internally always one and its participation cardinality is arbitrary. The right side membership cardinality of `Gym` is arbitrary, while its participation cardinality lies between 100 and 1000. This kind of representation is called the *compact representation of relationships*, as it doesn't establish a relationship instance for every athlete-gym tuple.

Another connection is established by the *Contact* relationship type. This relation models the fact, that an athlete acts as a contact person for one to three sports in one to two cities. An athlete can act as contact person only once, whereas sports and cities can be included any times. A relationship node is drawn as a combination of an oblong diamond overlaid by a rectangle.

4. Prototypical Implementation

This section deals with the design of the implemented prototype database, which is called *Lhasa DB*. The main aspects of this concept are represented by the *indoor* and *outdoor* metaphors.

As Lhasa DB persecutes a main memory approach, it is important to realize, that entities always reside in the main memory, no matter if they are actually *stored*. That is the reason for creating a virtual distinction between entities, which are indeed in the main memory, but not officially stored (thus *transient*) and those entities, which are marked as stored and thus officially *persistent* (and still in the main memory). In the given context, persistent means, that entities have passed through the regular *storing process*, which brings them from outdoor to indoor. This event is called *entity transition* or *entity crossing*.

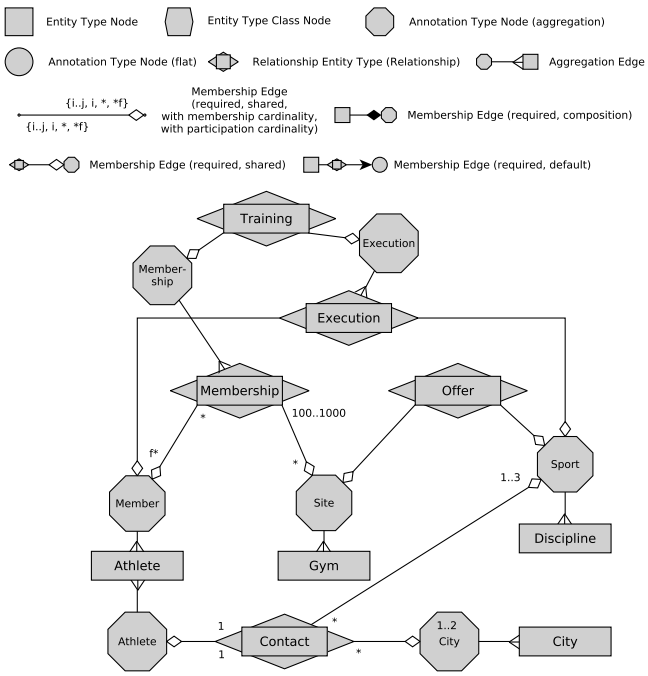

Figure 4. Relationships and Hyper Relationships

Entities, which do not carry the status stored are in a virtual region called *outdoor* or *outdoor sphere*. Entities carrying the stored status are in a virtual region called *indoor* or *indoor sphere*. The entity transition is triggered by a storing process letting an outdoor entity cross the *sphere frontier*. Entities before this process are transient, entities after this operation are considered persistent. Therefore, the common definition of *persistent* is changed not demanding a permanent storage outside of the main memory any more.

The latter aspect is realized by the *materialization process*, which backs up the persistent database state in the main memory to a permanent storage device. This procedure can be synchronized with the main memory storing process by enabling the *write-through* mode.

The crossing of an entity depends on whether it complies to the given schema. Therefore, the storing process initially checks, whether all constraints are fulfilled and the entity is composed correctly, that means, if a correct graph structure with all necessary nodes and edges is given. If that is the case, the entity and its annotations transit to the indoor sphere, which holds the current database state. This database state is represented through the *instance space* consisting of *instance graphs*, which are derived from the type graph and are accessible through entity and annotation indices. The crossing process also contains the *unification operation* stating whether there are value occurrences, which were stored before. If that is the case, these values are referenced instead to save main memory space and to fasten look-ups.

Instance nodes usually are connected in a *bidirectional* manner, that is a node n_1 has a directed outgoing reference to n_2 and n_2 an ingoing reference from node n_1. So usually, one will spot two edges realized by trivial memory references. But that is not the

case when it comes to unification or *shallow entities*, which are discussed later. Here, a *unidirectional* connection is used.

Assume, one wants to store a pool entity, whose type information is given in schema 3. In the first step, one must initially establish a living entity instance in the outdoor sphere. Therefore, the *prime* operation is used, which takes the desired entity type identifier as a parameter. After this is done, attribute values can be set accordingly to the schema information by instantiating corresponding annotations. Lhasa DB provides *meso operations* named *set*, *use* and *unset* to handle attribute values. The prime operations is one of the five Lhasa DB *PiCRUD macro operations* prime, *create*, *retrieve*, *update*, *destroy*, which will be discussed in detail later.

If an attribute value is assigned, usually an annotation node is build and provided with its value. In this case, the entity and the annotation node have a bidirectional connection. Both nodes are still in the outdoor sphere and thus not part of a potential indoor database state. When it comes to unification, the described process shapes differently. Here, the set operation checks, whether there is already an annotation with the same type and value, which was stored before and thus can be detected indoor in the instance space. Under these circumstances, the indoor annotation is referenced by a unidirectional connection coming from the outdoor entity. This reference must be unidirectional and it must be directed from the outdoor sphere into the indoor sphere to not alter the current database state. This kind of unification is called *instant unification*. So, outdoor structures are fully connected, if the involved nodes are transient (not stored), else unidirectional connections are used. Persistent (stored) indoor structures are always fully connected.

After all attributes are set, the established entity itself has to be integrated into the database state. Therefore, one uses the create operation, which triggers the storing process. This process initially performs integrity checks, unifies those attributes, which were not unified before (*deferred unification*) and expresses the state integration as a series of Lhasa DB *micro operations*, which realize instance space index manipulations as well as edge reallocations. The integration of the outdoor graph structure into the database state is here described as the *weaving process*. So the create operation triggers the storing process, which overall includes (a) a validation process, (b) a final unification process, (c) a weaving process and at least (d) a materialization process, which was mentioned above.

So far, the first two macro operations are covered. The next operation to be mentioned is called retrieve. This operation reads a stored entity and makes it available to the outdoor sphere, where it can be manipulated by the database user applying the mentioned meso operations. These actions then may end up in calling the PiCRUD update operation, which again triggers the store operation.

If an entity is brought from indoor to outdoor again, there are two main challenges, which must be addressed: the integrity of the database state and the synchronization of parallel access, when the entity is updated. That is the reason for introducing the *shallow entity*, which only exists outdoor and holds a pointer to its indoor counterpart. If an entity is to be retrieved, initially a shallow copy is created. Secondly, all annotations of the original entity are also linked to the shallow copy in a unidirectional manner. Finally, this construct is returned to the outdoor sphere and then accessible by the user. Changes, which are made on the shallow copy, do not immediately change the indoor database state and are deferred until the update operation again initiates the storing process, which is synchronized by the Lhasa DB *Transition Dispatcher* and *Concurrency Controller*. If a shallow entity is stored, it is treated as a new entity except for the fact, that all its new

annotation nodes are reallocated to the existing original entity, whose previous annotation nodes are then detached. The shallow entity is then expunged. The last operation *destroy* deletes the desired entity by removing it from the instance space index. Its annotation nodes reside, if they are used by other instances.

Figure 5 shows the unification before and after the storing process. Assume $x2$ as an outdoor entity instance of type X and $a2$ as an outdoor annotation type node of type A. Node $x2$ and $a2$ are connected in a bidirectional way, as both nodes are outdoor and full graph navigation possibilities are given. Entity $x2$ has another attribute node b of type B, which resides indoor. This annotation types was attached in a unidirectional way during the instant unification process. One can navigate from the outdoor entity to this indoor attribute, but one cannot navigate from b to $x2$ as long as it is not stored. Creating a bidirectional connection between indoor and outdoor nodes is illegal, because alteration of b would change the database state. As the figure illustrates, b was originally brought to the indoor sphere by the stored entity $x1$, which fully references another attribute node $a1$. After the storing process, the instance space experienced an alteration and $x2$ is now part of the database state. It is now save to fully attach both nodes $x2$ and b.

As it is important to check, whether operations change the internal state, all Lhasa DB operations are classified as either *state invasive* or *state retaining*. The meso operations set, use and unset are state retaining because they never change an indoor attribute directly. For instance, a set operation for assigning a new value to a stored unidirectional linked annotation node always causes the instantiation of a new attribute instance with the same type. The set operation only changes the attribute value directly, if it resides in the outdoor sphere. The macro operations PiCRUD are partly state invasive and partly state retaining, the same applies to the micro operations.

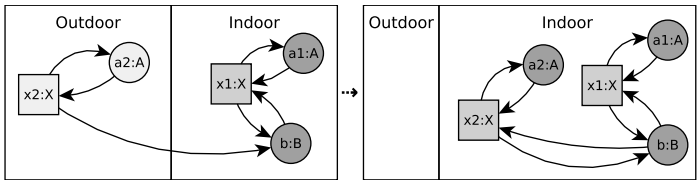

Figure 5. Unification before and after the Storing Process

5. BibTeX Experiment

5.1. The Scenario

The following experiment gives an impression about time and space requirement for establishing a database state. Based on a BibTeX example, a TEGeL schema is build and instantiated with given example data. The example schema is given in figure 6 and the example records consists of 205 `Pub`, 491 `Keyword` and 152 `Author` entities. Furthermore, it involves 1519 `PubKeyword` as well as 594 `PubAuthor` relationship nodes.

Based on these numbers and the given schema, one can calculate the hypothetical quantity of all instance nodes by counting all entity, attribute and relationship nodes required to establish a valid state. This calculation does neither consider the compact representation of relationships nor the unification.

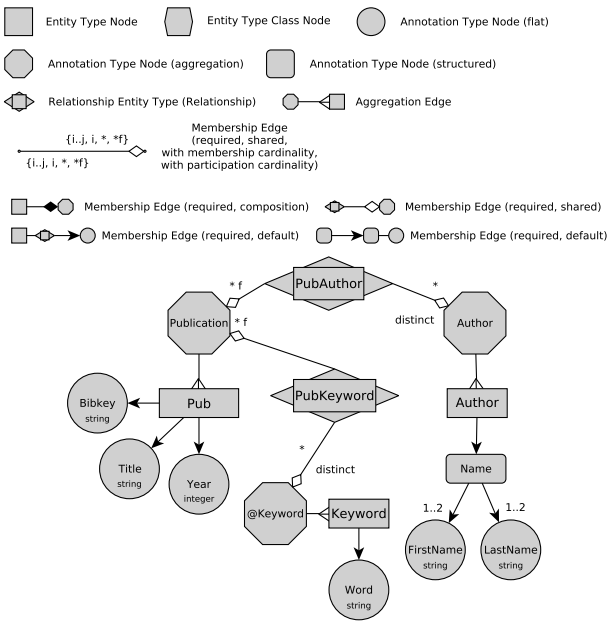

Figure 6. TEGeL BibTeX Example

So, one would completely establish 820 nodes for all `Pub`, 982 nodes for all `Keyword` and 925 nodes for all `Author` entities. Additionally, one must count 4557 nodes for all `PubKeyword` as well as 1782 nodes for all `PubAuthor` instances. By determining these figures, one must keep in mind, that multiple attributes always require an additional multi node and that relationships are basically modeled by additional aggregation nodes, which increase the final number of nodes seriously. After all, this state has an overall count of 9066 nodes.

In practice, Lhasa DB is able to reduce this amount of nodes up to 44,6 percent. Firstly, the prototypic implementation chooses a compact representation of relationships. That means, not every relationship in the sample also ends up in a de facto relationship node instance. This lowers the node quantity to 6070 units. Secondly, Lhasa DB uses unification to prevent identical values being stored, which cuts down the node number to finally 4040. Additionally, the overall edge count is also reduced while preserving the data integrity.

5.2. Measurements

The resource requirement of a database is essential for its application. That is the reason, why detailed measurements are performed and discussed in this section. The measuring problem is the complete composition of the database state along the sample data. Memory requirements are measured in kilobyte (KB) and the computing time in seconds (s). Additionally, the number of nodes and edges are counted.

The measurements are based on a comparison of overall nine different treatment scenarios. In the first scenario, the unification is completely disabled. The second scenario uses unification, but only for plain nodes and only in deferred mode. The third scenario uses instant as well as deferred unification exclusively for plain nodes. Subsequently,

the fourth scenario uses unification exclusively for structured types and so forth. Table 1 gives a complete overview.

Table 1. Measurement Scenarios

Acronym	Meaning
NU	no unification
PDU	exclusive deferred unification of plain nodes
PIDU	exclusive instant and deferred unification of plain nodes
SDU	exclusive deferred unification of structured nodes
ADU	exclusive deferred unification of aggregation nodes
AIDU	exclusive instant and deferred unification of aggregation nodes
MIDU	exclusive instant and deferred unification of multi nodes
CDU	complete unification (all nodes, only deferred)
CIDU	complete unification (all nodes, instant and deferred)

The measurements only deal with the constitution of the database state. The import of the sample data from a text file as well as the data preparation happens before. Based on the fact, that a profiler may tamper the test results, the loading time as well as the memory usage is taken manually by means of the process information. Every test series contains 100 values for every data point, for which the average as well as the standard deviation is calculated to estimate the sample quality. A measure is considered valid, if the standard deviation lies under 5 percent.

The measures are performed on a virtual Linux machine (VirtualBox 4.0.18 with Extension Pack on Ubuntu 12.10, GNU/Linux 3.5.0-17-generic x86_64) with 512 MB main memory and a 64 bit Intel Core 2 Duo CPU (T9550, 6 MB Cache, 2.66 GHz, 1066 MHz FSB). The prototype is implemented in Ruby 2.0.0p0, which uses the YARV virtual machine. The Linux virtual machine is restarted after every pass to prevent side effects induced by cache contents. Additionally, the system workload is kept minimal on the host as well as on the guest system to not slow down the ruby execution speed.

Before discussing the results, one must be aware of the fact, that the memory and loading time results directly correlate with the nature of the sample data. So, all beneath statements cannot be generalized. Besides, there is no constraint checking done yet.

Figure 7 shows, that the number of nodes varies between 6070 and 4040 and the number of edges between 5299 and 7335. This difference can be explained by the different unification scenarios, which realize varying compression rates. The element count is laid on the left y axis. The node count is represented by a solid line and the edge count by a dashed one. The first scenario NU does no unification, so the state consists of the maximum node and edge quantity. The same applies to the second scenario SDU, because there are no equal author names and thus no unification is possible. The third scenario PIDU contains a reduction to 5860 nodes, what can be explained by the fact, that some authors have the same first or last name. Also the plain year annotation type can be unified in multiple cases. Based on the fact, that no structured nodes are unified, the edge count stays the same at 7335.

The fourth scenario PDU only differs in the time when the unification is done and thus establishes an equivalent state. The MDU scenario however shows a clear reduction of the node and edge quantity. The unification of multi nodes related to the `FirstName`, `LastName` as well as to the multi aggregation nodes saves a lot of space. All in all, the

node count is reduced to 5564 and the edge count to 6632. The same effect shows up in the following scenarios AIDU and ADU, where more aggregation nodes can be unified and thus the count downsizes again to 4395 nodes and 5560 edges. The last two scenarios CIDU and CU use the complete unification, which reduces the node number to finally 4040 and the edge count to 5299. Thus, the node compression rate lies at 33.4 and the edge compression factor at 27.9 percent.

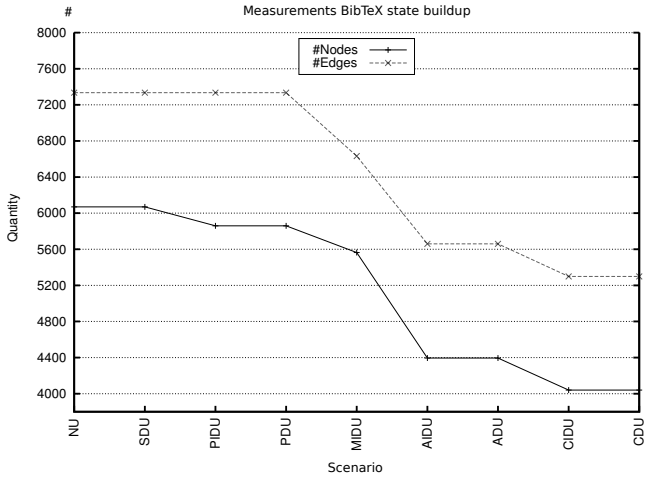

Figure 7. Node and Edge Quantity Measurements

Figure 8 shows the required main memory as well as the loading time. The scenarios are sorted by the state loading time in ascending order. The memory usage is laid on the left y axis and is represented by a solid line. The loading time is laid on the right y axis and drawn as a dashed line. The main memory requirements vary between 9397.12 and 7901.52 KB, whereas the loading times differ roughly in a two tenth seconds range and reside between 1.1589 and 1.2910 seconds. Contrary to the above discussion, the distinction between instant and deferred unification plays in important role regarding the loading time.

The first scenario AIDU exhibits the least loading time with 1.1589 s. This scenario enables the instant unification of aggregation nodes in contrast to the ADU scenario (1.2025 s), where only deferred unification is active. So, in the latter scenario, the maximum of aggregation nodes are initially instantiated and then partly dropped, when unification occurs.

The second scenario PIDU only unifies plain nodes, what leads to a heavy increase of memory usage with 9255.76 KB. This also explains the raised loading time, because the maximum node count must be build for remaining node types. The same applies to the following scenario PDU. The fourth scenario NU neglects the unification at all and so the maximum number of nodes and edges must be established. The loading time increases to 1.1693 seconds and the memory usage reaches the maximum measured at 9397,12 KB. As the following scenario SDU virtually offers no unification candidates, the values resemble strongly with 1.1703 s and 9383 KB. The sixth scenario MDU marks the beginning of a significant change concerning loading time. In this scenario, the multi nodes are only unified in deferred mode. So the maximum of redundant multi nodes are

instantiated. On the one hand, this creates a memory overhead, because much more ruby objects must be initiated, and on the other hand, it causes additional computation time, because a lot of nodes must be reallocated respectively deleted during the unification process. This phenomenon can be also observed in the following example ADU.

The last but one scenario CIDU unifies all nodes types immediately, if possible. Against the prospects of the authors, the loading time increases to 1.2508 clearly. One possible explanation lies in the fact, that a lot of attribute index requests are performed, what annihilates the advantages of reduced node instantiations. The longest loading time (1.2910 s) in the last scenario CDU can be justified with the much higher reorganization costs of deferred unification. The last scenario also takes a bit more memory, which cannot be explained so far.

The results support the hypothesis, that the unification has a negative loading time impact. This effect gets even worse, if the unification happens deferred. In this case, a reduction of main memory requirements leads to a clearly increased loading time and vice versa.

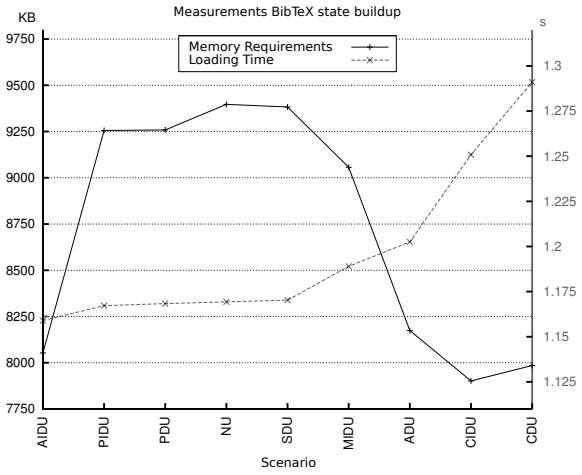

Figure 8. Time and Space Measurements

6. Conclusion

TEGeL complies with the requirements to be used for a conceptual and simultaneously for a logical and physical model, which simplifies the data modeling process. Furthermore, it also introduces concepts of object-oriented software engineering, like inheritance or interfaces. It is possible to express integrity constraints and the introduction of polymorphic aggregation leads to very flexible definition of entity type relations. Additionally, structured types, entity compositions and relationships of any kind can be modeled. TEGeL improves the semantic coherence, because every schema element has a unique name. Moreover, designers gain the possibility to reuse attributes. The direct expression of generalization, structured types, interfaces, aggregation, classes and clusters boosts semantics and eases the domain representation.

Lhasa DB uses TEGeL type graphs as logical schemata and establishes database states directly by deriving instance graphs. This procedure is described in detail above. Additionally, this paper explained the unification as a possibility to compress database states and to fasten index look-ups. We assume that unification generally leads to a very compact data representation, as it builds on the fact, that attribute values are discrete and often equal.

The results of the BibTeX experiment show, that a node takes up nearly 2 KB memory in average. This value must be seriously cut down to make Lhasa DB scalable. Current computer systems provide up to 2 TB of main memory[2] and could theoretically handle up to one billion nodes depending on the data quality. Larger memory requirements can be handled by distributed in-memory operation as it is discussed in [15]. Concerning performance, one must consider effective indexing methods as discussed in [13] and [14].

In addition to further implementation work, TEGeL may be extended to support *Role Based Access Control*. Information about roles and permitted operations can be attached to the membership edges realizing a very fine access control on the level of single attributes. A graph-based approach is introduced in [17]. When it comes to multi user access, a stable transaction concept must be elaborated. Additional, the specific retrieval of data can be reduced to the induced subgraph isomorphism problem by formulating query conditions as search graphs. In this regard [29], [32], [35] as well as [34] can be considered.

Furthermore, the materialization process must be designed in detail and an efficient synchronization algorithm must be developed. In this context, one may think about additional data compression with *gzip*[3] or *value only unification* of nodes irrespective of their types. Looking at fast SSD devices for materialization can also be an interesting investigation area as the partial swapping of main memory data to fast hard disks provides an additional possibility to extend storage capacity.

References

[1] Abiteboul, S., Hull, R.: Ifo: a formal semantic database model. In: Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems. pp. 119–132. PODS '84, ACM, New York, NY, USA (1984), http://doi.acm.org/10.1145/588011.588029

[2] Andries, M., Gemis, M., Paredaens, J., Thyssens, I., den Bussche, J.V.: Concepts for Graph-Oriented Object Manipulation. In: Pirotte, A., Delobel, C., Gottlob, G. (eds.) EDBT. Lecture Notes in Computer Science, vol. 580, pp. 21–38. Springer (1992)

[3] Angles, R.: A Comparison of Current Graph Database Models. In: Kementsietsidis, A., Salles, M.A.V. (eds.) ICDE Workshops. pp. 171–177. IEEE Computer Society (2012)

[4] Badia, A.: Extending Entity-Relationship Models with Higher-Order Operators. In: Ras, Z.W., Ohsuga, S. (eds.) ISMIS. Lecture Notes in Computer Science, vol. 1932, pp. 321–330. Springer (2000)

[5] Barcia, R., Hambrick, G., Brown, K., Peterson, R., Bhogal, K.S.: Object Relational Impedance Mismatch. IBM Press, Westford (Massachusetts) (2008)

[6] Böhm, R., Fuchs, E.: System-Entwicklung in der Wirtschaftsinformatik. vdf, Hochsch.-Verl. an der ETH, Zürich, 5 edn. (2002)

[7] Chen, P.P.: The entity-relationship model - toward a unified view of data. ACM Trans. Database Syst. 1(1), 9–36 (1976)

[2] see Dell PowerEdge R910

[3] an open data compression tool

[8] Codd, E.F.: Data Models in Database Management. In: Brodie, M.L., Zilles, S.N. (eds.) Workshop on Data Abstraction, Databases and Conceptual Modelling. vol. 11, pp. 112–114. ACM Press (1980)

[9] Fowler, M.: Patterns of Enterprise Application Architecture. Pearson Education, Inc., Boston (2003)

[10] Graves, M., Bergeman, E., Lawrence, C.: Graph database systems. Engineering in Medicine and Biology Magazine, IEEE 14(6), 737–745 (1995)

[11] Gumm, D., Janneck, M., Langer, R., Simon, E.J.: Mensch - Technik - Ärger? Zur Beherrschbarkeit soziotechnischer Dynamik aus transdisziplinärer Sicht. Lit Verlag Dr. W. Hopf Berlin, Münster (2008)

[12] Have, C.T., Jensen, L.J.: Are graph databases ready for bioinformatics? Bioinformatics 29(24), 3107–3108 (2013)

[13] Islam, S., Fariha, A., Ahmed, C.F., Jeong, B.S.: EGDIM: evolving graph database indexing method. In: Lee, S.H., Hanzo, L., Ismail, R., Kim, D.S., Chung, M.Y., Lee, S.W. (eds.) ICUIMC. p. 56. ACM (2012)

[14] Jin, R., Ruan, N., Xiang, Y., Wang, H.: Path-tree: An efficient reachability indexing scheme for large directed graphs. ACM Trans. Database Syst. 36(1), 7 (2011)

[15] Jouili, S., Reynaga, A.: imGraph: A Distributed In-Memory Graph Database. In: SocialCom. pp. 732–737 (2013)

[16] Jouili, S., Vansteenberghe, V.: An Empirical Comparison of Graph Databases. In: SocialCom. pp. 708–715 (2013)

[17] Koch, M., Mancini, L.V., Parisi-Presicce, F.: A graph-based formalism for rbac. ACM Trans. Inf. Syst. Secur. 5(3), 332–365 (2002)

[18] Lehner, F., Wildner, S., Scholz, M.: Wirtschaftsinformatik: Eine Einführung. Carl Hanser Verlag, München · Wien, 2 edn. (2008)

[19] Liu, Y., Vitolo, T.M.: Graph data warehouse: Steps to integrating graph databases into the traditional conceptual structure of a data warehouse. In: BigData Congress. pp. 433–434. IEEE (2013)

[20] Macko, P., Margo, D.W., Seltzer, M.I.: Performance introspection of graph databases. In: Kat, R.I., Baker, M., Toledo, S. (eds.) SYSTOR. p. 18. ACM (2013)

[21] Mamadolimov, A.: Search Algorithms for Conceptual Graph Databases. CoRR abs/1207.2837 (2012)

[22] Møller, A., Schwartzbach, M.I.: An introduction to XML and web technologies. Addison-Wesley (2006)

[23] Neubauer, W., Rudow, B.: Trends in der Automobilindustrie. Oldenburg Wissenschaftsverlag GmbH, München (2012)

[24] Pluciennik, T., Pluciennik-Psota, E.: Using Graph Database in Spatial Data Generation. In: Gruca, A., Czachórski, T., Kozielski, S. (eds.) ICMMI. Advances in Intelligent Systems and Computing, vol. 242, pp. 643–650. Springer (2013)

[25] Poulovassilis, A.: Database research challenges and opportunities of big graph data. In: Gottlob, G., Grasso, G., Olteanu, D., Schallhart, C. (eds.) BNCOD. Lecture Notes in Computer Science, vol. 7968, pp. 29–32. Springer (2013)

[26] Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw-Hill, New York, 7 edn. (2010)

[27] Rupp, C.: Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis. Carl Hanser Verlag, München, 5 edn. (2009)

[28] Samiullah, M., Ahmed, C.F., Nishi, M.A., Fariha, A., Abdullah, S.M., Islam, M.R.: Correlation Mining in Graph Databases with a New Measure. In: Ishikawa, Y., Li, J., Wang, W., Zhang, R., Zhang, W. (eds.) APWeb. Lecture Notes in Computer Science, vol. 7808, pp. 88–95. Springer (2013)

[29] Sundaram, G., Skiena, S.S.: Recognizing small subgraphs. Networks 25, 183–191 (1995)

[30] Thalheim, B.: Extended Entity-Relationship Model. In: Liu, L., Özsu, M.T. (eds.) Encyclopedia of Database Systems, pp. 1083–1091. Springer US (2009)

[31] Thomas, R.: Introduction to the Unified Modeling Language. In: TOOLS (25). p. 354. IEEE Computer Society (1997)

[32] Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM 23(1), 31–42 (Jan 1976), `http://doi.acm.org/10.1145/321921.321925`

[33] Wallmüller, E.: Software-Qualitätsmanagement. Carl Hanser Verlag München, München · Wien, 2 edn. (2001)

[34] Yuan, Y., Wang, G., Chen, L., Wang, H.: Efficient subgraph similarity search on large probabilistic graph databases. CoRR abs/1205.6692 (2012)

[35] Zheng, W., Zou, L., Lian, X., Wang, D., Zhao, D.: Graph similarity search with edit distance constraint in large graph databases. In: He, Q., Iyengar, A., Nejdl, W., Pei, J., Rastogi, R. (eds.) CIKM. pp. 1595–1600. ACM (2013)