# Support of Part-whole Relations in Query Answering

Piotr Kozikowski[1], Ekaterini Ioannou[2], Yannis Velegrakis[1], and Francesco Guerra[3]

[1] University of Trento (piotr.kozikowski@gmail.com, velgias@disi.unitn.eu)
[2] Technical University of Crete (ioannou@softnet.tuc.gr)
[3] University of Modena and Reggio Emilia (francesco.guerra@unimore.it)

**Abstract.** Part-whole relations are ubiquitous in our world, yet they do not get ”first-class” treatment in the data managements systems most commonly used today. One aspect of part-whole relations that is particularly important is that of attribute transitivity. Some attributes of a whole are also attributes of its parts, and vice versa. We propose an extension to a generic entity-centric data model to support part-whole relations and attribute transitivity and provide more meaningful results to certain types of queries as a result. We describe how this model can be implemented using an RDF repository and three approaches to infer the implicit information necessary for query answering that adheres to the semantics of the model. The first approach is a naive implementation and the other two use indexing to improve performance. We evaluate several aspects of our implementations in a series of experimental results that show that the two approaches that use indexing are far superior to the naive approach and exhibit some advantages and disadvantages when compared to each other.

## 1 Introduction

Part-whole relations exist virtually everywhere and their modelling plays an important role in many application domains [6]. Despite this, they do not get ”first-class” treatment in the data managements systems most commonly used today.

Part-whole relations have a number of properties and can be subdivided into different, more specific, kinds of relations. The conceptual modeling of part-whole relations and their different types and properties is a big challenge in itself and has been studied previously. The work in [1] provides a good summary of the difficulties and nuances involved. We believe that from all properties of part-whole relations the one that is most universal and deserves built-in support in a general-purpose data management system is that of attribute transitivity. It is very common to find that if some entity $x$ is part of other entity $y$, some attributes of $x$ are also attributes of $y$ and vice versa. For example, if Bob is an employee that works for the R&D department of company X, he also works for company X. Similarly, the person who owns a car usually also owns the car's engine. Query answering should take this transitivity into account: given that the contents of a database include “Bob works for R&D” and “R&D is part of company X”, the answer to the question “who works for company X?” should include Bob, even though this fact is not explicitly stated in the data.

This transitivity does not always hold, though. The hand of a violinist is definitely a part of his or her, but it would make little sense to say that this hand plays for an orchestra given that the violinist does. Therefore, it is desired to have the ability to specify which attributes are transitive with respect to part-whole relations and which are not.

In schema-centric systems, such as relational databases, this kind of functionality can be achieved by using the schema to establish specific types of entities and attributes and define which types of entities are part of other types of entities and which type of attributes should be transitive. For example, a table *department* may be declared to be *part-of* the table *company* by having a column *part-of* containing a foreign key to *company*. If a third table *employee* is associated to *department* with the relationship *works-for*, the question "who works for company X?" can be answered with a predefined query that takes the semantics of the schema into account. The main limitation of this approach is that all the semantics have to be known a priori and defined in the schema, which imposes a rigid structure to which all data must conform. In many cases, the data is not structured enough for the design of such a schema to be practical, yet considering attribute transitivity when answering queries is still useful.

Entity-centric systems, which deal with unstructured or semi-structured data and do not rely on a schema, offer even less support for part-whole relations. Entities can have any number of arbitrary attributes, the values of which are other entities or atomic values such as strings and numbers. In such scenario the *part-of* relation is just like any other attribute.

In this paper we propose a new type of database that allows attributes to be transitive with respect to part-whole relations and takes this transitivity into account in query answering, all while preserving the flexibility and suitability for schema-less semi-structured data characteristic of entity-centric systems. We introduce a new data model to account for part-of relations and attribute transitivity, implement this model using existing database technology, suggest indexing techniques to speed-up query answering, and explore many aspects of the performance of our implementation in a series of experiments.

The paper is organized as follows. Section 2 provides a motivating example that illustrates why considering attribute transitivity with respect to part-whole relations is useful for answering queries. Section 3 describes the generic data model and our extension for supporting part-whole relations and attribute transitivity. We describe our implementation of this extended data model in Section 4, discussing three different approaches. We provide an experimental evaluation of our implementation in Section 5, concluding in Section 7.

## 2 Motivating Example

Consider a system incorporating a repository that stores data related to non-governmental organizations (NGOs). NGOs typically represent and organize information in a very diverse way. This constitutes a challenge that the system needs to deal and especially when executing queries over the repository.

Figure 1 shows a small fragment of a repository that includes data related to the *Earth Charter*, one of the many NGOs affiliated with the United Nations. In the figure, each box represents an entity and each arrow represents an entity attribute. For example, the arrow labeled as *"affiliated-with"* denotes the *affiliation* of the entity *Earth Charter* with the *United Nations*. The arrows labeled as *"part-of"* denote that the *Earth Charter* is divided into the areas of *business*, *education*, and *religion*.

Consider now a user that wants to retrieve the *publications* of *Earth Charter*. Based on the information explicitly included in the repository, the answer to would only include *EC in action*. The publications *How a Consensus on Global Values Can Add Value* and *Principles for Engaging Business in the ECI* would not be included in the result set, since these are not directly associated with the entity *Earth Charter*. Of course, we could modify the model and

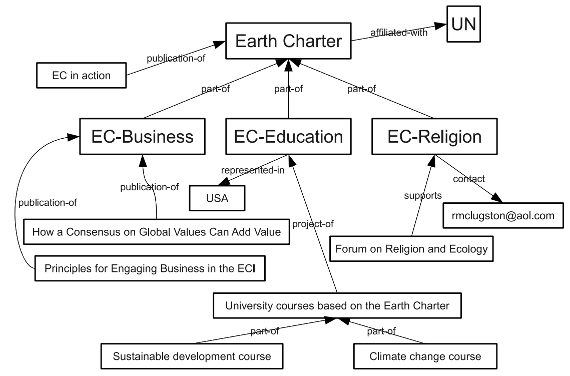

**Fig. 1.** A small fragment of the repository data.

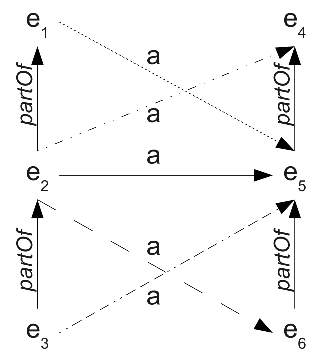

**Fig. 2.** An example of attribute transitivity.

associate the publications directly with the entity *Earth Charter* instead of *EC-Business*, but then a query asking for the publications of *EC-Business* would return no results. In addition, associating these publications with both entities would be cumbersome for the user of the repository, as she would have to figure out what is the hierarchy of areas and subareas of the Earth Charter before adding information about a publication.

Receiving inadequate query answers might frequently occur if we only use the information explicitly declared in the repository. Some additional examples with this issue appear when we try to retrieve the organizations that support the Earth Charter or the current projects of EC-Education.

To solve this problem, the repository should be able to reason about the implicit information implied by the *part-of* relations and be able to fully answer questions such as the ones discussed in the previous paragraphs. To do this in a sensible manner, we suggest incorporating additional knowledge for each attribute, and more specifically the transitivity of attributes with the *part-of* relations. We assume that the user of the repository does know this when inserting a new attribute. Note that this does not require the users to know what are the *part-of* relations themselves. For example, the user adding *Principles for Engaging Business in the ECI* as *publication-of* the entity *EC-Business* would know that this attribute has transitivity from part to whole. Thus, this publication should also be considered to be also a *publication-of* the entities that *EC-Business* is part of, i.e., the *Earth Charter* entity.

## 3 Data Model

### 3.1 Basic Database

We consider a generic data model that is centered around *entities* and *attributes*. An *entity* represents any object in the world, such as a person, a car or a school. No restrictions are imposed on the structure or characteristics of entities. An *attribute* describes some aspect of an entity and is composed of a name and a value. We assume the existence of an infinite set of entities $\mathcal{E}$, names $\mathcal{N}$, and atomic values $\mathcal{V}$. The latter contains values such as integers, strings, etc.

**Definition 1.** *Let pair* $\langle n,\ v\rangle$ *denote an attribute with* $n{\in}\mathcal{N}$ *being the attribute name and* $v{\in}\mathcal{E}{\cup}\mathcal{V}$ *the attribute value. A database is a tuple* $\langle E,\ G\rangle$ *where* $E{\subseteq}\mathcal{E}$ *is a finite set providing* *the entities (i.e., $\{e\}$). $G$ is a finite set that provides the attributes of each entity as well as the relationships between entities, i.e., $G \subseteq \mathcal{N} \times \{E \cup \mathcal{V}\}$.* ∎

Our definition can be used to represent structured data while also having the benefits of dataspaces [3] where data can be only partially structured.

It is easy to see that when all entities have at least one attribute, they will also be referred to in the $G$ set. We can actually include a triple of the form $\langle e, -, - \rangle$ in $G$ for each entity $e$ that does not have an attribute, and then consider this $G$ as the compact representation of the database. Given this compact representation, we can also retrieve the attributes of an entity $e_\alpha$ using the following function: $Attr(e_\alpha) = \{\langle n,\ v \rangle \mid \langle e_\alpha,\ n,\ v \rangle \in G\}$.

A query is described by a rule consisting of a head and a body. Both head and body are composed of a conjunction of atoms. The head can have only entity atoms of the form $e$ ($n_1$:$v_1$, $n_2$:$v_2$, ..., $n_k$:$v_k$), where $e$, $n_i$, and $v_j$ are variables or constants. Variables appearing at the beginning of an atom (outside the parenthesis), i.e., $e$, correspond to entity variables and can only be bound to elements of $\mathcal{E}$. Variables on the left side of each colon, e.g., $n_1$ and $n_2$, stand for attribute names and can only be bound to elements of $\mathcal{N}$. Variables on the right side of each colon, e.g., $v_1$ and $v_2$, stand for attribute values and can be bound to either elements of $\mathcal{E}$ (entities) or elements of $\mathcal{V}$ (atomic values). In addition to entity atoms, the body of a query can also have atomic atoms, which are boolean conditions involving variables and constant values, e.g, $x$<10 or $x$=$y$.

Note that the variables appearing anywhere in the query are shared across all atoms from both the head and the body. Nesting is possible by using atoms as attribute values.

Given a binding of the variables $e$, $n_i$, and $v_i$ to $e^b$, $n_i^b$, and $v_i^b$ respectively, for every $i$ from 1 to $k$, the entity atom $e$ ($n_1$:$v_1$, $n_2$:$v_2$, ..., $n_k$:$v_k$) is said to be *true* if there is an entity $e^b$ in the database that has attributes $\langle n_i^b,\ v_i^b \rangle$, $\forall\ i \in$ [1,$k$]. If all the atoms in the body of a query are true, the atoms in the head of the query are also considered to be true, in which case a set of entities and attributes as described in the head of the query is returned.

*Example 1.* Consider a user that want to detect an entity with the attribute `publication-of` having as value some entity, which in turn has the attribute `affiliated-with` with yet some other entity as value. She thus poses the following query:

***Q 1*** | *\$pub(related-to:\$org)* **:-** *\$pub(publication-of: \$ngo),\$ngo(affiliated-with: \$org)*

Executing **Q 1** over the data of Figure 1 return the entity `EC in action` having the attribute `related-to` with the entity `UN` as value. ∎

## 3.2 Extension to Part-of Databases

We now extend the data model for the basic database (introduced in the previous paragraphs) to a model that accounts for the transitivity of attributes with respect to *part-of* relations.

The set $\mathcal{T}$ = { *up*, *down*, *both*, *none* } defines the four possible types of attribute transitivity. The elements *up* and *down* represent transitivity from part to whole, and whole to part, respectively. The element *both* denotes that both of the previous types of transitivity apply, and element *none* indicates that there is no transitivity. Additionally, the special name *partOf* is removed from the set of valid attribute names $\mathcal{N}$ and is reserved for part-of relations.

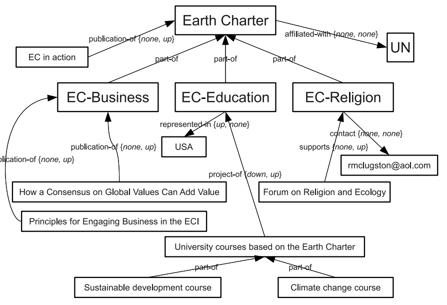
Earth Charter
publication-of {*none, up*}
affiliated-with {*none, none*}
UN
EC in action
part-of
part-of
part-of
EC-Business
EC-Education
EC-Religion
represented-in {*up, none*}
USA
contact {*none, none*}
publication-of {*none, up*}
supports {*none, up*}
rmclugston@aol.com
publication-of {*none, up*}
project-of {*down, up*}
How a Consensus on Global Values Can Add Value
Forum on Religion and Ecology
Principles for Engaging Business in the ECI
University courses based on the Earth Charter
part-of
part-of
Sustainable development course
Climate change course

**Fig. 3.** Part-of database for Figure 1 data.

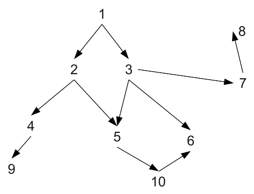

**Fig. 4.** DAG example.

| $et$ | $eu$ | $ed$ |
|---|---|---|
| *none* | *false* | *false* |
| *up* | *true* | *false* |
| *down* | *false* | *true* |
| *both* | *true* | *true* |

**Fig. 5.** Correspondence for transitivity.

**Definition 2.** *A part-of database is a tuple $\langle E,\ G,\ P\rangle$. $E$ is the entity set, i.e., $E\subseteq\mathcal{E}$. $P$ is a set of entity pairs, i.e., $P\subseteq E\times E$, with each pair $\langle e_i,\ e_j\rangle$ denoting $e_i$ is partOf $e_j$. $G$ provides relationships of an entity with its attributes or other entities, including also the transitivity type, i.e., $G\subseteq E\times\mathcal{N}\times\{E\cup\mathcal{V}\}\times\mathcal{T}\times\mathcal{T}$. Each element in $G$ is $\langle e,\ n,\ v,\ et,\ vt,\ \rangle$ with $e\in\mathcal{E}$ being the entity, $n\in\mathcal{N}$ the attribute name, $v\in E\cup\mathcal{V}$ the attribute value, and $et$,$vt\in\mathcal{T}$ the transitivity for the entity that has the attribute and the attribute value $v$, respectively.* ∎

Each element of $G$ states that an entity $e$ has an attribute with name $n$ and value $v$ and that this attribute is transitive with respect to the part-of relations defined in $P$ as specified by $et$ and $vt$. Only one combination of $n$, $v$, $et$, and $ev$ is allowed per attribute (i.e. the same attribute cannot be declared twice with conflicting types of transitivity). In addition, using the information of $P$, i.e., $partOf$ relationships between entities, we form a graph and we assume that there are no cycles in this graph.

As with the original model (Section 3.1), under the assumption that every entity in a part-of database has at least one attribute, the set $E$ is no longer needed to represent the database and a more compact representation consisting of the quintuples in $G$ and the pairs in $P$ is possible. We denote this representation with tuple $\langle G,\ P\rangle$.

Figure 2 provides a graphical representation of the semantics of attribute transitivity. The entity $e_3$ is part of $e_2$, which in turn is part of $e_1$. Entities $e_4$, $e_5$, and $e_6$ have a similar configuration. The entity $e_2$ has an attribute $a$ with $e_5$ as value. If *et=up* or *et=both* for attribute $a$, it is inferred that $e_1$ also has an attribute with name $a$ and value $e_5$ by virtue of transitivity, even if such attribute is not explicitly declared in $G$. Similarly, if *et=down* or *et=both* for attribute $a$ then $e_3$ has an attribute with name $a$ and value $e_5$. The cases for the different values of $vt$ are analogous.

Consider again our example with the NGOs repository. The data about Earth Charter can be represented using the part-of database shown in Figure 3. All attributes have now values for $et$ and $vt$. Part-of relations are no longer normal attributes but a special kind of relationship between entities. For instance, the attribute *publication-of* of the entity *Principles for Engaging Business in the ECI* has *et=none* and *vt=up*. This means that this attribute has no transitivity on the entity's side, and transitivity of type *up* on the value's side. Since the value of this attribute is *EC-Business*, which is part of *Earth Charter*, an attribute *publication-of* is implicitly established for the entity *Principles for Engaging Business in the ECI* having *Earth Charter* as value.

The query language for part-of databases has the same elements and structure as the one defined in the previous section. The difference lies in the semantics of queries. Each atom in the body of a query can now be satisfied with the entities and attributes in the inferred closure of $\langle G, P\rangle$, defined as $\langle G \cup G', P\rangle$, where $G'$ is the set of all triples $\langle e, n, v\rangle$ that can be inferred from the transitive attributes in $G$ and the part-of relations in $P$ by applying the inference rules described previously.

*Example 2.* Consider now searching for all entities that have an attribute `publication-of` with `Earth Charter` as value in the inferred closure of $\langle G, P\rangle$.

***Q 2*** | *\$pub(publication-of: Earth Charter)* **:-** *\$pub(publication-of: Earth Charter)*

If applied to the data of Figure 3, this query returns entities: (i) `EC in action`, (ii) `How a Consensus on Global Values Can Add Value`, and (iii) `Principles for Engaging Business in the ECI`. Each of these three entities has an attribute `publication-of` with `Earth Charter` as value. ∎

## 4 Our Solutions

We implemented the part-of database using Sesame[4], which is an open source framework for storing and querying RDF data. We chose RDF because it is a well established specification of a data model that is similar to the one we use. Sesame offers one of the most robust and mature implementations of an RDF repository available as open source.

### 4.1 Naive Approach

Our first implementation follows a lazy inference approach. All information is stored in a standard RDF repository and nothing is inferred at the time of insertion. When the database is queried, it analyzes the body of the query for patterns, inserts RDF statements corresponding to the inferred closure of those patterns, translates the query into SPARQL, executes it, and removes the inferred statements.

Update operations are uninteresting since they are direct mappings to the corresponding operations of the RDF repository. The only exception is that whenever a part-of relation is added, a check is made to ensure the addition does not introduce a cycle in the part-of graph $G_p$. Supposing that the function *addPartOfRelation*($e_1$, $e_2$) is invoked, the part-of graph is traversed to ensure that there is no path from $e_2$ to $e_1$, in which case the operation fails.

Query answering involves several steps. The query is parsed and every entity atom is broken into one or more minimal atoms of the form *e(n:v)*. The atoms in the body of the query are classified according to which elements are constants and which are variables. If we denote variables with "?", the eight possible patterns are: *e(n:v)*, *e(n:?)*, *e(?:v)*, *e(?:?)*, *?(n:v)*, *?(n:?)*, *?(?:v)* and *?(?:?)*. If the pattern *?(?:?)* is found, the inferred closure of the entire database is computed, otherwise, only the implicit statements involving the constants in the query patterns are inferred.

The inferred closure of a pattern can be a subset of that of other pattern. In this case the former is considered redundant and only the latter is computed. For example, a query may contain the pattern $e_1$*(?:*$v_1$*)*, which requires the inference of all the implicit statements that have $e_1$ as entity and $v_1$ as attribute value. If the same query also contains the pattern $e_1$*(?:?)*,

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

all the implicit statements that have $e_1$ as entity have to be inferred, which are a superset of those with $e_1$ as entity and $v_1$ as attribute value.

Figure 2 is helpful in visualizing how the inferred closure of a pattern can be obtained. To see the case for pattern *e(n:?)*, suppose a part-of database contains entities $e_1, e_2, e_3, e_4, e_5$ and $e_6$, with the part of relations shown in the figure, and that $e_2$'s attribute with name $a$ and value $e_5$ has full transitivity (*et=both* and *vt=both*). The dashed lines represent the inferred closure of the database. First, every entity $e'$ that is directly or indirectly a part of $e$ (i.e. there is a path from $e'$ to $e$ in $G_p$) has to be checked for ownership of an attribute $\alpha$ with name $n$ that has *et=up* or *et=both*. If such attribute exists, the statement $\langle e, n, valueOf(\alpha)\rangle$ is added to the inferred closure. In the example, doing this for the pattern $e_1$*(a:?)* would result in $\langle e_1, a, e_5\rangle$ being added to the inferred closure. Next, every entity $e''$ that $e$ is directly or indirectly part of (i.e. there is a path from $e$ to $e''$ in $G_p$) has to be checked for ownership of an attribute $\beta$ with name $n$ that has *et=down* or *et=both*. If such attribute exists, the statement $\langle e, n, valueOf(\beta)\rangle$ is added to the inferred closure. In the example, the statement $\langle e_3, a, e_5\rangle$ would be added to the closure if this were done for pattern $e_3$*(a:?)*. Finally, every entity $v$ that is the value of an attribute $\gamma$ of $e$ that has name $n$ and *vt$\neq$ none* has to be checked for being directly or indirectly part of some entity $v'$ (if *vt=up* or *vt=both*), and/or an entity $v''$ being directly or indirectly part of $v$ (if *vt=down* or *vt=both*). If such entities exist, the statements $\langle e, n, v'\rangle$ and/or $\langle e, n, v''\rangle$ are added to the closure. In the example, the statements $\langle e_2, a, e_4\rangle$ and $\langle e_2, a, e_6\rangle$ would be added to the closure after doing this for pattern $e_2$*(a:?)*.

The case for pattern *?(n:v)* can be seen as a mirror of that for *e(n:?)*. The same steps are performed but the roles of $e$ and $v$ are interchanged. Other patterns require different steps but the underlying logic is the same and is not fully detailed for the sake of brevity.

Once the inferred closure of all the atoms in the body of a query is added to the repository, the equivalent SPARQL query is executed using Sesame's query engine. After the results of the query are available, all the inferred statements that were added are removed leaving the database in the same state as before executing the query.

## 4.2 Total Materialization

We now present another approach that follows an eager inference approach. Each time a statement or part-of relation is added, its inferred closure is computed. Similarly, when a statement or part-of relation is removed, the affected inferred statements are removed. The query execution is left entirely to Sesame's query engine, thus our main concern in this implementation is the speed of update operations.

**Part-of Relations Index.** Whenever a transitive attribute is inserted, the part-of graph has to be traversed to find all the relevant entities for the inferred closure. In the same way, deleting a transitive attribute requires a traversal of the part-of graph to find those inferred statements that should be removed. To speed-up this process, we propose an index dedicated to part-of relations.

The set of all part-of relations can be seen as a directed acyclic graph (DAG). We represent this graph with a double adjacency list that contains both the direct and indirect part-of relations. For example, the DAG in Figure 4 is represented as follows:

```
1 {2,3}   {4,5,6,7,8,9,10} {}    {}
2 {4,5}   {6,9,10}         {1}   {}
```

| | | | | |
|---|---|---|---|---|
| 3 | {5,6,7} | {8,10} | {1} | {} |
| 4 | {9} | {} | {2} | {1} |
| 5 | {10} | {6} | {2,3} | {1} |
| 6 | {} | {} | {3,10} | {1,2,5} |
| 7 | {8} | {} | {3} | {1} |
| 8 | {} | {} | {7} | {1,3} |
| 9 | {} | {} | {4} | {1,2} |
| 10 | {6} | {} | {5} | {1,2,3} |

There is an entry for every vertex in the DAG. The first column contains the list of vertices each vertex is connected to. The second column contains the list of vertices each vertex is indirectly connected to through other vertices. Columns three and four contain what the first and second columns would, if the direction of all edges were reversed. The idea is that a single look-up of this index is all that is needed for the insertion or deletion of any transitive statement. Depending on the type of transitivity, the lookup may include all columns or only some of them.

This index can be implemented with a key-value store. We used Ehcache[5], a system developed in Java meant primarily for caching the contents of some other larger and slower datastore, but it can also be used as a persistence solution to store very large datasets on disk without depending on any external database. The reason to prefer Ehcache over something such as Berkeley DB is that it performs object caching -Java objects are only serialized and deserialized when written to or read from disk-; Berkeley DB and other similar systems must serialize and deserialize Java objects every time they are inserted or retrieved, even if no access to disk is made. We found that object caching has a big influence on performance even if the amount of objects kept in memory is a relatively small part of all the stored data.

**Transitive Attributes Index.** The index for part-of relations is useful when inserting and deleting transitive statements, but the insertion and removal of part-of relations can benefit in turn from fast access to transitive attributes. This is because the inferred closure of transitive attributes already existing in the database can be affected by the addition or removal of a part-of relation. Given that transitive statements are stored as reified triples in RDF, their retrieval is not very efficient. Moreover, it is usually the case that only some of the transitive attributes are relevant for a given operation (e.g., only those with the transitivity type *up* and involving certain entity). It is therefore desired to have a fast way of retrieving transitive attributes with different characteristics.

To index transitive attributes, we propose to use an alternative representation of transitivity. Instead of having the two variables $et$ and $vt$ with four possible values each, we store four binary variables: *eu*, *ed*, *vu* and *vd*. Table 5 shows the correspondence of the two representations for *et*, *eu* and *ed*. The correspondence between *vt*, *vu*, and *vd* is analogous.

Using this representation, the index consists of four tables, one for each of the binary variables *eu*, *ed*, *vu* and *vd*. To avoid having to modify the inner workings of the Sesame repository, we also added a fifth table containing all transitive attributes; this is the table that is indexed by the other four. Figures 7-10 show how the example contents of Figure 6 are indexed. Naturally, in the real implementation the index tables contain pointers to entries in the main table rather than the entire triples as shown here.

[5] http://ehcache.org/

| | |
|---|---|
| $e_1, n_1, v_1$ | *eu=true, ed=true, vu=false, vd=false* |
| $e_1, n_2, v_2$ | *eu=true, ed=false, vu=false, vd=false* |
| $e_1, n_3, v_3$ | *eu=false, ed=true, vu=false, vd=false* |
| $e_2, n_4, v_4$ | *eu=false, ed=false, vu=true, vd=true* |
| $e_2, n_5, v_5$ | *eu=false, ed=false, vu=true, vd=false* |
| $e_2, n_6, v_6$ | *eu=false, ed=false, vu=false, vd=true* |

**Fig. 6.** Table with transitive attributes.

| | |
|---|---|
| $e_1$ | $e_1, n_1, v_1$ |
| $e_1$ | $e_1, n_2, v_2$ |

**Fig. 7.** Index for *eu*.

| | |
|---|---|
| $e_1$ | $e_1, n_1, v_1$ |
| $e_1$ | $e_1, n_3, v_3$ |

**Fig. 8.** Index for *ed*.

| | |
|---|---|
| $v_4$ | $e_2, n_4, v_4$ |
| $v_5$ | $e_2, n_5, v_5$ |

**Fig. 9.** Index for *vu*.

| | |
|---|---|
| $v_4$ | $e_2, n_4, v_4$ |
| $v_6$ | $e_2, n_6, v_6$ |

**Fig. 10.** Index for *vd*.

Similarly to the part-of relations index, we implemented the transitive attributes index using a key-value store.

## 4.3 Smart Indexing

The previous approaches represent two extremes. The naive approach has fast update operations and the best possible space utilization given our choice of RDF repository, but much processing is required for answering queries, making them slow. The total materialization approach, on the other hand, achieves the fastest query answering possible given our choice of RDF repository and SPARQL query engine, but uses considerable extra space to have all the implicit information always readily available for queries, and needs to perform maintenance operations with every update.

We now present an approach that compromises the conflicting goals of fast query answering and fast update operations with low space requirements. Ideally, we should rewrite the query engine itself to compute the transitive closure of query patterns progressively and avoid the inference of statements that can be discovered to be unnecessary given the partial results already obtained from the query. This is, however, outside of the scope of our research. Instead, we still compute the inferred closure of all non-redundant patterns in the body of a query, as in the naive approach, but we try to do this as efficiently as possible with the help of an index.

In order to avoid traversing the part-of graph when computing the inferred closure of query patterns, we use the same part-of relations index used for the total materialization approach. We also need an index for transitive attributes, but the one from the total materialization approach is not very suitable for our current task. While inserting and deleting part-of relations is only concerned with transitive attributes involving either a specific entity or a specific attribute value, query answering often requires both entity and attribute value, and also the attribute name, depending on the query pattern. We could add a table for each of the query patterns, but this would result in at least six tables, requiring much extra space and defeating part of the purpose of not opting for the total materialization approach.

Instead, we propose to use B-trees -the basic data structure employed in Berkeley DB and many other databases- to simulate tries and allow prefix search. This way we can use a single index to search for relevant transitive attributes having one, two, or all three elements $e$, $n$, $v$ of an attribute. This allows us to provide fast response time to all the access patterns required by query patterns using only three indexes: one sorted by $env$, other by $vne$, and the third by $pev$.

Using the entity, attribute name, and attribute value as key, and the attribute transitivity as value in a B-tree, just as Figure 6 does, we achieve prefix search functionality by providing a custom comparator to be employed as the basis for sorting by the B-tree. A different comparator is used for each of the three sortings we need. For example, the index sorted by $env$ employs a comparator that uses $v$ as basis for comparison of two keys only if their respective values for $e$ and $n$ are equal; otherwise only $e$ and $n$ are considered. Likewise, the values of $n$ are considered only if the keys have the same values of $e$. When searching the B-tree having only a prefix of the key, like $en$, we get the first key with the values of $e$ and $n$ that we specified, if such key exists, and can iterate through contiguous entries until we encounter a key that has a different value of some element of the prefix than what we specified. At this point we stop, having retrieved all attributes that share the prefix we searched for.

## 5 Experimental Results

We tested our implementations of the part-of database using synthetic datasets composed of entities connected to each other with part-of relations in a tree configuration. We used five parameters: $n$ is the number of trees in a dataset, $h$ the height of each tree, $b$ the number of children per node, $attrs$ the number of non-transitive attributes per node, and $t\_attrs$ the number of transitive attributes. Transitive attributes are transitive either from part to whole ($et = up$ and $vt = up$), or from whole to part ($et = down$ and $vt = down$), with an equal number of each.

Each experiment consist in the creation of a synthetic dataset from scratch, a warm-up routine to ensure the database is fully initialized, and a set of measurements of different operations, each repeated several time selecting random nodes and the results averaged. In addition to measuring the time to complete common operations, we also measured the size taken on disk by the database and its index, if any, after ensuring that all its contents were flushed to disk. In all graphs we provide the main variable being measured (i.e., time or space) on the vertical axis on the left, and the number of RDF triples used by the given configuration, not including any inferred triples, on the vertical axis on the right. This helps to observe the behavior relative to the size of the data.

**Performance.** Our main interest is the performance of queries. We evaluated all combinations of queries with the patterns *e(n:?), e(?:v)* and *?(n:v)* using as $e$ and $n$ the root of the tree and a random node at the bottom of the tree, and transitive attributes as $n$.

Our first experiment focused on testing the behavior of trees with $n$=1, $h$=1, $attrs$=0 $t\_attrs$=2, and large numbers of children per node $b$. The results are shown in Figure 11. In the following experiment we tested the opposite extreme: trees with $b$=1 (not really trees) and high values of $h$, the other parameters remaining unchanged. Figure 12 shows the results. As one may expect, the performance achieved with total materialization is much better than the other two approaches. The time taken for processing queries by the naive implementation seems to be directly proportional to the number of RDF statements. Smart indexing performs noticeable better and seems to be more resilient to wide and shallow trees than it is to long chains.

To try a more realistic scenario, we also tested trees that have more balanced ratios of $h$ to $b$. Figure 13 shows the case where $b$=2 for various values of $h$. Figure 14 shows the inverse case with $h$ fixed to 4 and various values of $b$. The results are very similar to the previous ones. Height has a slightly greater impact on performance than number of children, but performance is best correlated with the number of RDF triples, which in these experiments is directly proportional to the number of nodes in the tree.

Next, in experiment 5, we explored the effect of adding transitive attributes with other parameters fixed at $n = 1$, $h = 4$, $b = 5$ and $attrs = 0$. Figure 15 shows the results. It is clear that, at least with tree configurations, the number of transitive attributes has a much

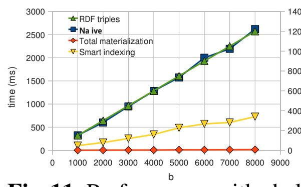

**Fig. 11.** Performance with shallow trees.

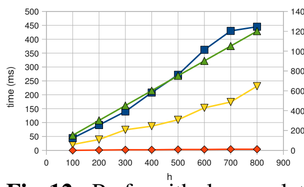

**Fig. 12.** Perf. with long relation chains.

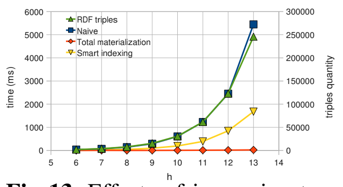

**Fig. 13.** Effects of increasing tree height.

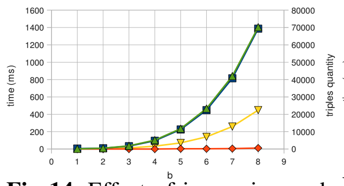

**Fig. 14.** Effect of increasing node children.

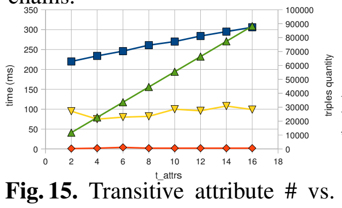

**Fig. 15.** Transitive attribute # vs. performance.

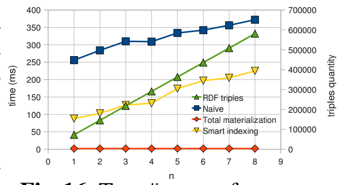

**Fig. 16.** Tree # vs. performance.

smaller effect on performance than the number of part-of relations. This time even the naive implementation has a performance degradation less than proportional to the number of RDF triples.

Finally, experiment 6 consisted in maintaining the tree structure and number of attributes fixed at $h = 4$, $b = 5$, $attrs = 50$ and $t\_attrs = 6$ and observing the effect of increasing the size of the database by adding trees. The results are shown in figure Figure 16. In this experiment smart indexing had less of an advantage over the naive implementation. In both cases the time for answering queries grows less than proportionally to the number of RDF triples.

Overall, the result of all experiments are similar. Smart indexing has a big advantage over the naive implementation, but does not come anywhere even close to the levels of performance achieved by total materialization.

**Database Size.** Figure 17 shows the results of experiment 6 for database size. Evidently, the total materialization approach gets the worst results, but considering that it uses both an index and additional space in the RDF repository to store inferred statements, it is surprising that the amount of space it uses grows only slightly more than proportionally to the number of RDF triples. Obviously the naive approach is the best in terms of database size. Smart indexing is right in the middle.

In a typical RDF repository there are many more distinct object, predicates and objects than the URI prefixes they use. Therefore, storing the identifiers separately from URI prefixes can improve space utilization. This is most likely already done in Sesame's RDF repository and it is a simple way to improve the space efficiency of our indexing.

**Update Operations.** Update operations where tested by inserting a part-of relation between the root of a tree and a newly created entity and between a leaf node of a tree and other newly created entity. New transitive attributes were also inserted for the same nodes. The average time for those insertions was measured, after which the operations were undone with the corresponding deletions, the average time of which was also measured. Figures 18 and 19 show the results of doing this in experiment 6.

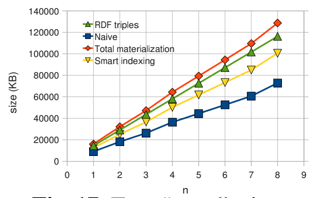

**Fig. 17.** Tree # vs. db size.

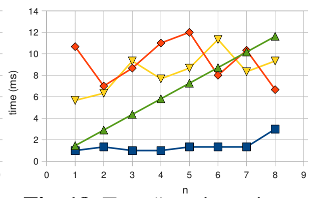

**Fig. 18.** Tree # vs. insertion.

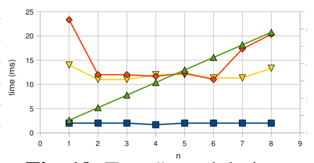

**Fig. 19.** Tree # vs. deletion.

Maintaining the indexes imposes a noticeable overhead, but in a typical scenario in which queries are much more common than updates, it is well worth it.

## 6 Related Work

Part-whole relations have been studied from a conceptual modelling perspective [5, 4, 7]. The different cases that have been considered there are covered by our model. Modelling of part-whole relations has been studied in object oriented databases [1, 6, 2]. The main difference with our work is that they focus specifically on object oriented systems and their modelling of part-whole relations takes place at the schema level, as a mechanism to among other things improve the enforcement of the semantics defined in the schema. The model we consider does not require a schema and is more suited for semi-structured data. A similar line of work studied the information that can be inhereted across the different relations proposed for their epistemological layer of knowledge.

## 7 Conclusions

We proposed a new model to express part-of relations and the transitivity of attributes from whole to part and part to whole. Queries in this model take into account part-of relations and attribute transitivity to return more meaningful results, without any extra effort from the user. We implemented this model using an RDF repository and suggested two different approaches to index part-of relations and transitive attributes to improve query performance. Our experimental evaluation of these techniques indicates total materialization is unmatched in terms of query performance, but the performance of inferring implicit information in a lazy fashion can be improved significantly with an index with modest space requirements.

## References

1. Alessandro Artale, Enrico Franconi, Nicola Guarino, and Luca Pazzi. Part-whole relations in object-centered systems: an overview. *Data Knowl. Eng.*, 20(3):347–383, 1996.
2. Elisa Bertino and Giovanna Guerrini. Extending the ODMG object model with composite objects. *SIGPLAN Not.*, 33(10):259–270, 1998.
3. Xin Dong and Alon Halevy. Indexing dataspaces. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 43–54, Beijing, China, 2007. ACM.
4. Peter Gerstl and Simone Pribbenow. Midwinters, end games, and body parts: a classification of part-whole relations. *Int. J. Hum.-Comput. Stud.*, 43(5-6):865–889, 1995.
5. Peter Gerstl and Simone Pribbenow. A conceptual theory of part-whole relations and its applications. *Data Knowl. Eng.*, 20(3):305–322, 1996.
6. Michael Halper, James Geller, and Yehoshua Perl. An OODB part-whole model: semantics, notation and implementation. *Data Knowl. Eng.*, 27(1):59–95, 1998.
7. Morton Winston, Roger Chaffin, and Douglas Herrmann. A taxonomy of Part-Whole relations. *Cognitive Science*, 11(4):444, 417, 1987.