Managing Semantic Evolutions in Semi-Structured Data

Pedro Ivo Siqueira Nepomuceno(✉) and Kelly Rosa Braghetto

Department of Computer Science, University of Sao Paulo, Sao Paulo, Brazil
{pedro.siqueira,kellyrb}@ime.usp.br

Abstract. This paper introduces a model to store semi-structured data while documenting its semantic changes over time. The paper also presents algorithms for querying semantic evolved data, which conciliate the multiple versions the data may have. An implementation of the model and algorithms, MellowDB, was developed, and its performance was analyzed, showing the proposed algorithms and model are feasible.

Keywords: Databases · Semantic heterogeneity · Query Rewriting

1 Introduction

Several works have addressed database evolution in structured [3] and semi-structured databases [6]. Most, however, focus on schema evolution. Our work, on the other hand, focuses on operations over the attributes' values (semantic evolution), which change the data semantics over time. The Brazilian county of "Moji Mirim" for example, was renamed to "Mogi Mirim" in 2016 [5]. Official statistical data before 2016 refers to "Moji Mirim", while from 2016 and beyond, "Mogi Mirim" is referred to. In another example, "Laguna" was ungrouped in 2013 into "Laguna" and "Pescaria Brava". After ungrouping, numbers inform the population estimates for each new county. However, it is possible to group new estimates to make a grouped analysis using all previous registers.

Even when subtle, semantic heterogeneity can make old and new data incompatible so that they cannot be judiciously grouped or compared [9]. This paper presents a model to represent the semantic evolution of semi-structured data collections and algorithms for easily querying them. Both model and algorithms were implemented as a middle layer over MongoDB, and its performance was evaluated through extensive experiments.

This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, FAPESP proc. 14/50937-1, and FAPESP proc. 15/24485-9.

2 Related Work

Temporal Data Models (TDMs) preserve the complete history of data changes. This way, it is possible to retrieve current values and query states in specific moments of past time [8]. Most TDMs have been proposed or implemented using relational database management systems (RDBMS), although there are some implementations in semi-structured data, such as in JSON files [1].

TDMs do not directly tackle semantic evolution. Mainly because in semantic evolution, changes are generated following declared rules (the SEOs). But they do present a deep framework to deal with time, including timestamping, modeling, and querying techniques useful for dealing with semantic evolution.

Database Evolution demands special care to enable easy querying. The main strategy to support good querying interfaces for databases that suffered evolution is *query rewriting*. Moon et al. [7] and Möller et al. [6] developed systems capable of dealing with different schema versions using query rewriting as long as the evolution history is known. Another related technique (*delta code generation*) automatically generates views to mimic tables before and after the evolution. Herrmann et al. [4] presented a tool for generating delta code between schema versions.

It is important to notice that all the above-cited works deal with schema evolution. **Semantic evolution, which is the main target of this paper, is not dealt with**. In fact, semantic evolution is a less explored area in scientific literature. Ventrone [9] defined some types of semantic heterogeneity and evolution forms which result in operations similar to the ones considered in this work. However, no algorithms or models to deal with them were presented.

3 Framework to Handle Semantic Evolution Operations

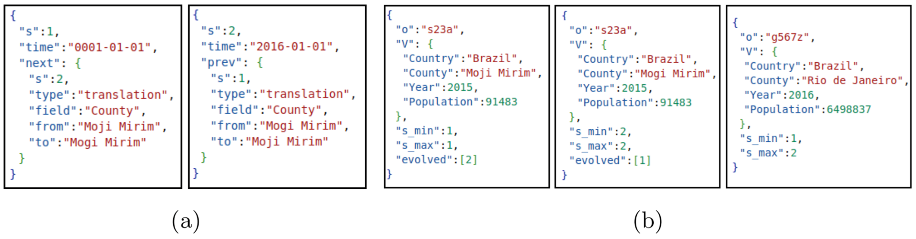
This section formalizes a semantic evolution operation, the *translation*, to illustrate how to deal with semantic evolution in semi-structured collections. Other operations such as grouping and ungrouping can be defined similarly.

Definition 1. *A document $d = (t, V)$ contains a timestamp t and a set V of attribute-value pairs. The notation $V[a]$ will represent "the value of attribute a". In other words, $V = \{(a, v)|V[a] = v\}$.*

According to these definitions, documents may contain only simple values. The extension to consider complex values in nested structures is future work.

Definition 2. *The translation operation($T_{t_h,a,q,r}(d)$) transforms the value of attribute a of a document d from q to r starting at time t_h. This is defined as:*

$$T_{t_h,a,q,r}(d) = \begin{cases} (t, V \setminus \{(a,q)\} \cup \{(a,r)\}), & \textit{if } t \leq t_h \textit{ and } V[a] = q \\ d, & \textit{otherwise} \end{cases} \quad (1)$$



```
{
  "s":1,
  "time":"0001-01-01",
  "next": {
    "s":2,
    "type":"translation",
    "field":"County",
    "from":"Moji Mirim",
    "to":"Mogi Mirim"
  }
}
```

```
{
  "s":2,
  "time":"2016-01-01",
  "prev": {
    "s":1,
    "type":"translation",
    "field":"County",
    "from":"Mogi Mirim",
    "to":"Moji Mirim"
  }
}
```

(a)

```
{
  "o":"s23a",
  "V": {
    "Country":"Brazil",
    "County":"Moji Mirim",
    "Year":2015,
    "Population":91483
  },
  "s_min":1,
  "s_max":1,
  "evolved":[2]
}
```

```
{
  "o":"s23a",
  "V": {
    "Country":"Brazil",
    "County":"Mogi Mirim",
    "Year":2015,
    "Population":91483
  },
  "s_min":2,
  "s_max":2,
  "evolved":[1]
}
```

```
{
  "o":"g567z",
  "V": {
    "Country":"Brazil",
    "County":"Rio de Janeiro",
    "Year":2016,
    "Population":6498837
  },
  "s_min":1,
  "s_max":2
}
```

(b)

Fig. 1. (a) Versions collection C_s and (b) documents in the processed collection C_p. The first two of (b) are different semantic versions of the same document.

The example cited in Sect. 1 is a translation with q = *Moji-Mirim*, r = *Mogi-Mirim* and $t_h = 2016$. The translation operation is reversible; it can be formalized similarly.

Definition 3. *A semantic evolution compatible collection C is composed of tuples (d, s), where d is a document and s is the semantic version of the document.*

When a semantic evolution operation (SEO) takes place over a collection, first, a new semantic version is created. Then a new version of every document in the collection is created and associated with the new semantic version.

Every version of a document is a copy of the original document after all changes from previous semantic operations are applied. Each tuple (d_1, s_i) is a *version* of the original document d_1. A *semantic version* s_i is as a subset of the semantic compatible collection, where all its associated documents have the same semantic interpretation for their attribute-value pairs.

4 Storage Model and Algorithms

The proposed model contains three collections. The *raw collection* stores the original documents. The *semantic versions collection* keeps metadata of the semantic versions. The *processed collection* stores documents in all semantic versions.

The **Raw Collection** (C_r) contains the original document attribute-value pairs $(d_r.V)$ as well as its valid time $(d_r.time)$ and the *original version number* $(d_r.s)$ which is the version in effect using $d_r.time$ as reference.

Each document (d_s) of the **Semantic Versions Collection** (C_s) contains: the *version number* $(d_s.s)$; the *valid time* $(d_s.time)$ of the version and the *next and previous version operation and arguments* $(d_s.next/d_s.prev)$ with the arguments of the SEO that needs to be applied to map the version into the next and the previous one (depending if it is a reversible operation or not). These two fields resemble a doubly linked list. Any arguments needed, such as the translation t_h, q, and r, are also included in these attributes. Figure 1a shows an example of two documents of C_s.

In the **Processed Collection** (C_p), storing one version of each document for each semantic version is impractical. A better approach is to associate documents with an interval of versions. Then, when there are no changes in a document, the version can only be extended. Each document d_p, besides the original attribute-value pairs set ($d_p.V$) edited to fit its semantic version, includes the following metadata attribute-value pairs: the *original document* ($d_p.o$), a reference to the original document in the raw collection; the *minimum* ($d_p.s_{min}$) and *maximum* ($d_p.s_{max}$) *version number* that define the limits of version range in which the copy of the document is valid; and the *evolution list* ($d_p.evolved$), indicating every SEO that affected that document. Figure 1b shows an example of the processed collection for a document affected by a semantic evolution and one that has not. For documents that have not been affected, the full interval of semantic versions can be synthesized in only one processed document.

4.1 Semantic Operation Processing

The first semantic version document d_{s_1} is also created when the collection is created. Valid time of this version ($d_{s_1}.time$) is set to zero ($d_{s_1}.t = 0$).

When a SEO is executed, a new semantic version is created with a new version number. If the operation happened before another previously informed one, this number might be fractional to "fit" between two other pre-existing versions. The *prev* and *next* of neighboring versions must be reconnected correspondingly.

The next step is to process documents into the *processed collection* accordingly. For unaffected documents, limits of pre-existing processed documents are just extended. For each affected document, it is necessary to create another copy to represent it from that point in time. Figure 1b show an example of how documents stay when affected by a semantic operation (the Moji/Mogi Mirim case) and when not affected (the Rio de Janeiro case).

After all affected processed documents are copied or have their value extended, the SEO may occur. This step depends on the operation and will happen as stated in Definition 2 over the documents associated with the new semantic version s_j. Then, all posterior operations must be reapplied over these documents because their results might be different than before.

When new documents are inserted into the database, they must also be processed consistently, checking if it has been affected by any SEO.

4.2 Query Transformation Algorithm

When querying, it is necessary to consider semantic changes affecting queried attributes. This way, users may query an attribute by its old or new value seamlessly. To make an *attribute:value* filter query the procedure is:

1. Query the semantic versions collection (C_s) for any semantic "next" operation where the *attribute:value* combination has been transformed into another value (*attribute:new_value*). If there are any, add the *attribute:new_value :semantic_version_ number* of these semantic versions to a queue P.

While P is not empty, pop the first *attribute:value:semantic_version* tuple and make the same query again in C_s, to check if this attribute has been transformed into still another value. If it has, push the new *attribute:value :semantic_version* to P. If not, add to another list, L_2. This is to detect "new names" that could represent the queried value in the most recent version.

2. All of L_2 values will be used to compose the final query, using an OR operator in the selection criteria while filtering the semantic version in the *evolved* attribute. The original *attribute:value* is also added.
   As an example, consider the collections shown in Figs. 1a to 1b and the query *"County":"Moji Mirim"*. The final version of the query (Q) will be:

$$\begin{aligned} Q = &\{\text{“County”:“Moji Mirim”}\} \\ &\text{or } (\{\text{“county”:“Mogi Mirim”}\} \text{ and } \{\text{“evolved\_\_contains”:“1”}\}) \end{aligned} \tag{2}$$

The final query should be executed in C_p, but only in the last semantic version subset. It considers both counties that were called "Moji Mirim" and were renamed to "Mogi Mirim" and counties that are still named "Moji Mirim" in order to consider homonyms also if they exist.

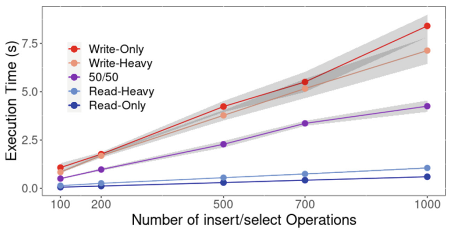
5 Implementation and Performance Analysis

To validate and evaluate the model and algorithms introduced in Sect. 4, we implemented MellowDB, a middle-layer library developed in Python to deal with semantic evolution in MongoDB. For now, it implements operations for insertion and querying. The developed code and all experiments scripts are publicly available on https://github.com/pisn/semantic_heterogeneous_database.

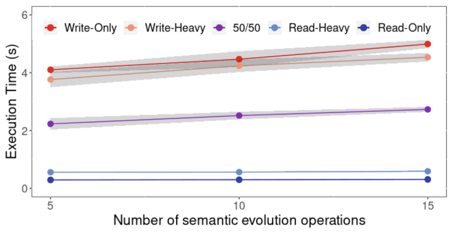
For the experiments, databases with 500K documents containing 20 fields, each with a domain of 20 possible values, were randomly generated. Five different scenarios were simulated in this phase: *Read-Only* (only queries), *Heavy Read* (95% of queries and 5% insertions), *Write Only* (only insertions), *Heavy Write* (95% insertions and 5% queries) and *50/50* (50% insertions and 50% queries). These scenarios were inspired by YCSB Framework workload scenarios [2]. Every experiment was repeated 5 times. Repetitions were executed over a newly created database in an environment with Debian 5.4.19-1 OS, Intel(R) Core(TM) i7-6700K CPU @ 4.00 GHz, and 30 GB RAM.

Figure 2a shows that queries suffer less overhead than inserts because documents are already pre-processed in C_p to be queried, while documents being inserted must pass through the evolution process. Figure 2b shows that the heterogeneity level of the database affects the insert operations, but not the queries, also because documents are already pre-processed to be queried.

MellowDB obviously added some overhead over the operations. However, querying without its aid would demand from users not only much more effort and time but also a deep knowledge of the database domain. Nevertheless, for the insertion of 500 documents, the worst average time was roughly 5 s.



(a)



(b)

Fig. 2. Execution times (95% confidence interval) for all scenarios with different (a) quantities of select/insert operations and (b) levels of semantic heterogeneity.

6 Concluding Remarks

This work advances the state-of-art techniques in managing semi-structured data heterogeneity caused by database evolution. The formalization of the evolution operations and the storage model and algorithms to deal with them presented here are original contributions, there is no similar approach in the related work.

The theoretical framework, models, and algorithms provide tools to deal with semantic heterogeneity in semi-structured data. As long as the operations history is registered, users may query the database without being aware of details on the values' changes. Results show that the use of the proposed models is feasible, achieving desired results much faster and more conveniently than if the operations were manually treated.

References

1. Brahmia, S., Brahmia, Z., Grandi, F., Bouaziz, R.: τJSchema: a framework for managing temporal JSON-based NoSQL databases. In: Hartmann, S., Ma, H. (eds.) DEXA 2016. LNCS, vol. 9828, pp. 167–181. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44406-2_13
2. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing, pp. 143–154 (2010)
3. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Automating the database schema evolution process. VLDB J. **22**(1), 73–98 (2013)
4. Herrmann, K., Voigt, H., Behrend, A., Rausch, J., Lehner, W.: Living in parallel realities: co-existing schema versions with a bidirectional database evolution language. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1101–1116. SIGMOD/PODS 2017 (2017)
5. Instituto Brasileiro de Geografia e Estatística - IBGE: Alterações topomínicas (2022). https://www.ibge.gov.br/geociencias/organizacao-do-territorio/estrutura-territorial/27336-alteracoes-toponimicas-municipais.html
6. Möller, M.L., Klettke, M., Hillenbrand, A., Störl, U.: Query rewriting for continuously evolving NoSQL databases. In: International Conference on Conceptual Modeling, pp. 213–221. ER 2019 (2019)

7. Moon, H.J., Curino, C.A., Deutsch, A., Hou, C.Y., Zaniolo, C.: Managing and querying transaction-time databases under schema evolution. Proc. VLDB Endowment **1**(1), 882–895 (2008)
8. Tansel, A.U., Clifford, J., Gadia, S., Jajodia, S., Segev, A., Snodgrass, R.: Temporal databases: theory, design, and implementation. Benjamin-Cummings Publishing Co., Inc. (1993)
9. Ventrone, V.: Semantic heterogeneity as a result of domain evolution. ACM SIGMOD Rec. **20**(4), 16–20 (1991)