Capturing the History and Change Structure of Evolving Data

George Papastefanatos, Yannis Stavrakas, Theodora Galani
IMIS, RC ATHENA
Athens, Greece
{gpapas,yannis,theodora}@imis.athena-innovation.gr

Abstract—**Evo-graph is a model for data evolution that encompasses multiple versions of data and treats changes as first-class citizens. A change in evo-graph can be compound, comprising disparate changes, and is associated with the data items it affects. In previous papers, we have shown that recording data evolution with evo-graph is very useful in cases where the provenance of the data needs to be traced, and past states of data need to be re-assessed. We have specified how an evo-graph can be reduced to the snapshot holding under a specified time instance, we have given an XML representation of evo-graph called evoXML, and we have presented how interesting queries can be answered. In this paper, we explain how evo-graph is used to record the history of data and the structure of changes step by step, as the current snapshot evolves. We present C2D, a novel framework that implements the concepts in the paper using XML technologies. Finally, we experimentally evaluate C2D for space and time efficiency and discuss the results.**

Keywords-data evolution; change modeling

I. Introduction and Preliminaries

Data published on the Web undergo frequent changes due to advancements in knowledge and due to the cooperative manner of their curation. Users of scientific data, in particular, would like to go beyond revisiting past data snapshots, and review how and why the recorded data have evolved, in order to re-evaluate and compare previous and current conclusions. Such an activity may require a search that moves backwards and forwards in time, spread across disparate parts of a database, and perform complex queries on the semantics of the changes that modified the data. The need for accounting for past changes and tracing data lineage is evident not only in scientific data, but also in a wide range of web information management domains.

Motivating Example. We will use an example taken from Biology: the revision in the classification of diabetes, which was caused by a better understanding of insulin [12]. Initially, diabetes was classified according to the age of the patient, as *juvenile* or *adult onset*. As the role of insulin became clearer two more subcategories were added: *insulin dependent* and *non-insulin dependent*. All *juvenile* cases of diabetes are *insulin dependent*, while *adult onset* may be either *insulin dependent* or *non-insulin dependent*. In Fig. 1, the leftmost image depicts a tree representation of the initial diabetes classification, while the rightmost the revised classification. These two representations, however, do not provide any information about which parts of the data evolved and how, which changes led from one version to another, or what changes were applied on which parts of the data. Recording change operations in a log or discovering deltas out of successive versions, like many systems do, do not solve the problem; in most cases isolated operations are impossible to interpret a posteriori. This is because they usually form more complex, semantically coherent changes, each comprising many small changes on disparate parts of the data.

We argue that in systems where evolution issues are paramount, changes should not be treated solely as transformation operations on the data, but rather as *first class citizens* retaining structural, semantic, and temporal characteristics. In previous work, we proposed a graph model, *evo-graph* [16], and its XML representation, *evoXML* [17], capturing the relationship between evolving data and changes applied on them. A key characteristic is that it explicitly models changes as first class citizens and thus, enables querying data and changes in a uniform way. In what follows, we discuss some preliminary concepts on evo-graph and then present the contribution and structure of this paper.

Snap-graph. We assume that data is represented by a rooted, node-labeled, leaf-valued graph called *snap-graph*. A snap-graph S (V, E) consists of a set of nodes V, divided into complex and atomic, with atomic nodes being the leaves of the graph, and a set of directed edges E. At any time instance, the snap-graph undergoes arbitrary changes.

Evo-graph. An *evo-graph* G is a graph-based model that captures all the instances of an evolving snap-graph across time, together with the actual change operations responsible for the transitions. It consists of the following components:

- *Data nodes*, divided into *complex* and *atomic*: $V_D = V_D^c \cup V_D^a$.
- *Data edges* depart from every complex data node, $E_D \subseteq (V_D^c \times V_D)$.
- *Change nodes* are nodes that represent change events. Change nodes are depicted as triangles, to distinguish from circular data nodes. They are divided into *complex* and *atomic* (denoting basic change operations): $V_C = V_C^c \cup V_C^a$.
- *Change edges* connect every complex change node to the (complex or atomic) change nodes it encompasses: $E_C \subseteq (V_C^c \times V_C)$.
- *Evolution edges* are edges that connect each change node with two data nodes, specifically the version before and after the change: $E_E \subseteq (V_D \times V_C \times V_D)$.

Intuitively, the evo-graph consists of two interconnected graphs: a data graph comprising the different versions of

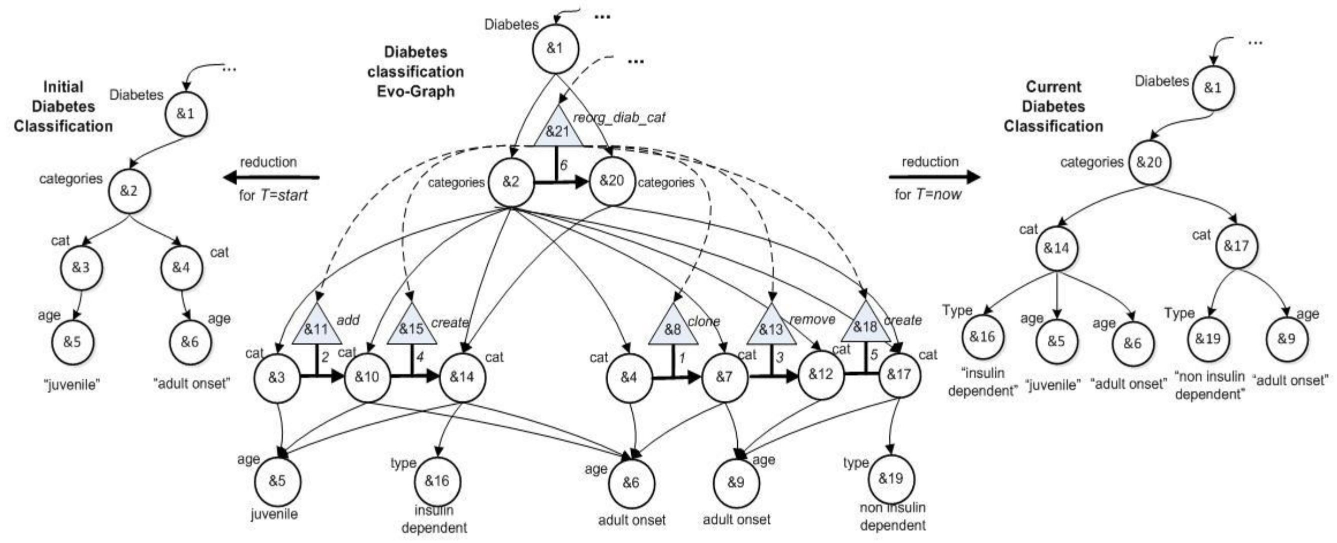

Figure 1. Snap Graphs of diabetes classification before (left) and after (right) revision and the corresponding evo-graph (middle).

data, and a tree of changes. The data graph defines the structure of data, while the change graph defines the structure of changes. These two graphs interconnect via evolution edges. Consequently, there are two roots: the data root, r_D, and the change root, r_C. Moreover, we annotate change nodes with a timestamp denoting the time instance that the specific change occurred. These timestamps are used for determining the validity timespan of all data nodes and data edges in the evo-graph. Evo-graph can be reduced to a snap-graph holding under a specified time instance through the *reduction* process [16]. A snap-graph is actually a trivial case of an evo-graph, consisting of a set of data nodes $V \subseteq V_D$ and a set of data edges $E \subseteq E_D$.

As an evo-graph example consider the middle image in Fig. 1, which represents the revision in the diabetes classification from the graph of Fig. 1 left to the graph of Fig. 1 right. The revision process is denoted by the complex change *reorg_diab_cat*, (node &21) composed by 5 basic snap changes (in the order they occurred): *clone* (node &8), *add* (node &11), *remove* (node &13), *create* (node &15), and *create* (node &18). Note the use of evolution edges; in the case of *add* the evolution edge is annotated with the timestamp 2 and connects node &3 (initial version) with node &10 (version after adding the child node &6). Node &10 is still a child of node &2, but for simplicity the relevant edge is omitted. The reduction of the evo-graph for T=*start* results in the snap-graph of the leftmost image of Fig. 1, while for T=*now* in the snap-graph of the rightmost image of Fig. 1. For the complete definitions of basic snap changes see section 2.1.

EvoXML. In [17] we have shown how evo-graph can be represented in an XML format, called *evoXML*. TABLE I. presents an evoXML example. Due to space limitations, the evoXML example covers up to time instance 1 of the evo-graph in Fig. 1; specifically it includes only the *clone* operation (node &8) in lines 12-15, 20. Notice that the edge from node &7 to node &6 (which actually denotes that &6 remains a child of the next version of node &4) is represented through the evoXML reference *evo:ref* in line 13, which points to the element in line 10. Also notice how the change node &8 is represented in line 20.

Querying Evolution. Finally, in [16],[17] we have outlined *evo-path*, an XPath extension that help us posing regular queries over data snapshots as well as time- and change-aware queries on evo-graph. We have also shown how evo-path expressions can be evaluated on evoXML via equivalent XQuery expressions. Evo-path takes advantage of the complex change information in order to retrieve and relate data that are otherwise distant and irrelevant to each other. Queries expressed on evo-graph include:

- Temporal queries on the history of data nodes, like "which is the structure of categories before the time instance 6"?
  `Evo-path: //Diabetes/categories [ts() not covers {now}]`
- Evolution queries on changes applied to data nodes, like "which changes are associated with the change responsible for the reorganization of diabetes categories" (node &21)?
  `Evo-path: <//reorg_diab_cat/*>`
- Causality queries on relationships between change nodes and data nodes, like "what are the previous versions of all data nodes that changed due to the reorganization of diabetes categories"?
  `Evo-path: //* [evo-before() <//reorg_diab_cat>]`

Contribution and Structure. In this paper, we first define a set of *basic changes* on the snap-graph, and how these can be combined to construct *complex changes* (section 2). We then define a *set of basic operations on the evo-graph, and a translation from snap-graph changes to evo-graph operations*, such that as changes occur on the snap-graph, the evo-graph grows to represent those changes together with all the successive snap-graph versions (section 2). Furthermore, we introduce the *C2D framework* (section 3), a prototype system that implements the concepts introduced in this paper, and progressively builds the evo-graph as changes take place on the current snap-graph. We present

TABLE I. EVOXML FOR TIME INSTANCE 1.

```
<evo:evoXML xmlns=””
 xmlns:evo=”http://web.imis.athena-innovation.gr/projects/c2d”>
  <evo:DataRoot evo:id=”dataroot”>
    <Diabetes evo:id=”1”>
      <categories evo:id=”2”>
        <cat evo:id=”3”>
          <age evo:id=”5”>juvenile</age>
        </cat>
        <cat evo:id=”4”>
          <age evo:id=”6”>adult onset</age>
        </cat>
        <cat evo:id=”7” evo:ts=”1” evo:previous=”4”>
          <age evo:ref=”6”/>
          <age evo:id=”9”>adult onset</age>
        </cat>
      </categories>
    </Diabetes>
  </evo:DataRoot>
  <evo:ChangeRoot evo:id=”changeroot”>
    <clone evo:id=”8” evo:tt=”1” evo:before=”4” evo:after=”7”/>
  </evo:ChangeRoot>
</evo:evoXML >
```

and discuss a detailed *experimental evaluation* of C2D (section 3). Finally, we review the related work (section 4) and we conclude the paper (section 5).

II. ACCOMMODATING BASIC AND COMPLEX CHANGES IN EVO-GRAPH

A. Snap Basic and Complex Change Operations

In this section, we define the basic change operations applied on a snap-graph S(V,E) (*snap changes* for short) and present how they can be employed to define complex changes. We consider the following snap changes:

- *create(v^P, v, label, value)*. Creates a new atomic node *v* with a given *label* and *value* and connects it with its parent node v^P. If v^P is an atomic node, it becomes complex.
- *add(v^P, v)*. Adds the edge *(v^P, v)* to E, effectively adding *v* as a child node of v^P. The nodes v^P, *v* must already exist in V. If v^P is an atomic node, it becomes complex.
- *remove(v^P, v)*. Removes the edge *(v^P, v)* from E. If *v* has no other incoming edges, it is removed from V. If v^P has no other children, it becomes an atomic node with the default value (empty string).
- *update(v, newValue)*. Updates the value of an atomic node *v* to *newValue*.
- *clone(v^P, v^{source}, v^{clone})*. Creates a new data node v^{clone} with the same label/value as v^{source}, and a deep copy of the subtree under v^{source} as a subtree under the node v^{clone}. The node v^P must be a parent of v^{source}. The edge (v^P, v^{clone}) is added to E, making v^{clone} a sibling of v^{source}.

The above definitions describe the effect of each snap change to the current snap-graph. These changes leave the snap-graph in any possible consistent state. Note that the effect of the *clone* snap-change is to create a deep copy of a subtree under the same parent node. Although *clone* can be expressed as a sequence of other snap changes, we chose to consider it as a basic operation. The reason is that deep copy is difficult to express using successive *create* operations, while at the same time it is an essential operation for expressing complex changes like *move-to*, and *copy-to*.

A *complex change* applied on a node of the snap-graph is a sequence of basic and other complex change operations that are applied on the node itself or/and the node’s descendants, and allows us to group operations in semantically coherent sequences. Applying a complex change on a snap-graph involves the application of each constituent change in the order they appear. Consider the complex change *reorg_diab_cat* applied on *categories* node of the leftmost image of Fig. 1. This change is expressed as a sequence of five basic snap changes, as follows:

```
reorg-diab-cat (&2) {
   clone (&4, &6, &9)
   add (&3, &6)
   remove (&4, &6)
   create (&3, &16, “type”, “insulin dependent”)
   create (&4, &19, “type”, “non insulin dependent”) }
```

B. Capturing Versions and Changes with Evo-graph

In our approach, snap changes are not actually applied on the snap-graph, but on the evo-graph. This is shown in Fig. 2, which illustrates the effects of snap changes to the evo-graph. Fig. 2 depicts three images for each snap change; the leftmost image shows the initial snap-graph before the change, the rightmost image shows the current snap-graph after the snap change, and the middle image shows the evo-graph fragment encompassing both snapshots, together with the change. Notice that these snap-graph fragments are actually *reductions* [16] of the respective evo-graph under different time instances. Thus, the *create* operation in Fig. 2 actually causes node &4 to be added under the parent node &5, and not under &2, as would be the case if *create* was applied directly on the snap graph. This is a technical issue tackled with at the implementation level, and does not introduce any ambiguities.

In order to implement snap changes on an evo-graph G we introduce the following evo-graph operations:

- *addDataNode (v_D^P, v_D, label, value)*. Creates a new atomic data node v_D as a child of v_D^P with a given *label* and a *value*. If v_D^P is an atomic node, it turns into complex.
- *addDataEdge (v_D^P, v_D)*. Creates a new data edge from node v_D^P (parent) towards node v_D (child). The two nodes must already exist in V_D. If v_D^P is an atomic node, it turns into complex.
- *applyAtomicChange(v_D^1, v_D^2, value, v_C, v_C^P, label, timestamp)*. This operation “evolves” node v_D^1 to node v_D^2, as the result of applying a snap change. First, a new atomic data node v_D^2 with the same label as v_D^1 and a given value is created, and is connected as a child of all the current parents of v_D^1. Then, a new atomic change node v_C with the *label* and *timestamp* is created, and is connected as a child of node $v_C^P \epsilon V_C^c$. The *label* denotes one of the snap changes defined previously. Finally, a new evolution edge *e* = *(v_D^1, v_C, v_D^2)* is created between the data nodes v_D^1, v_D^2 and the change node v_C.

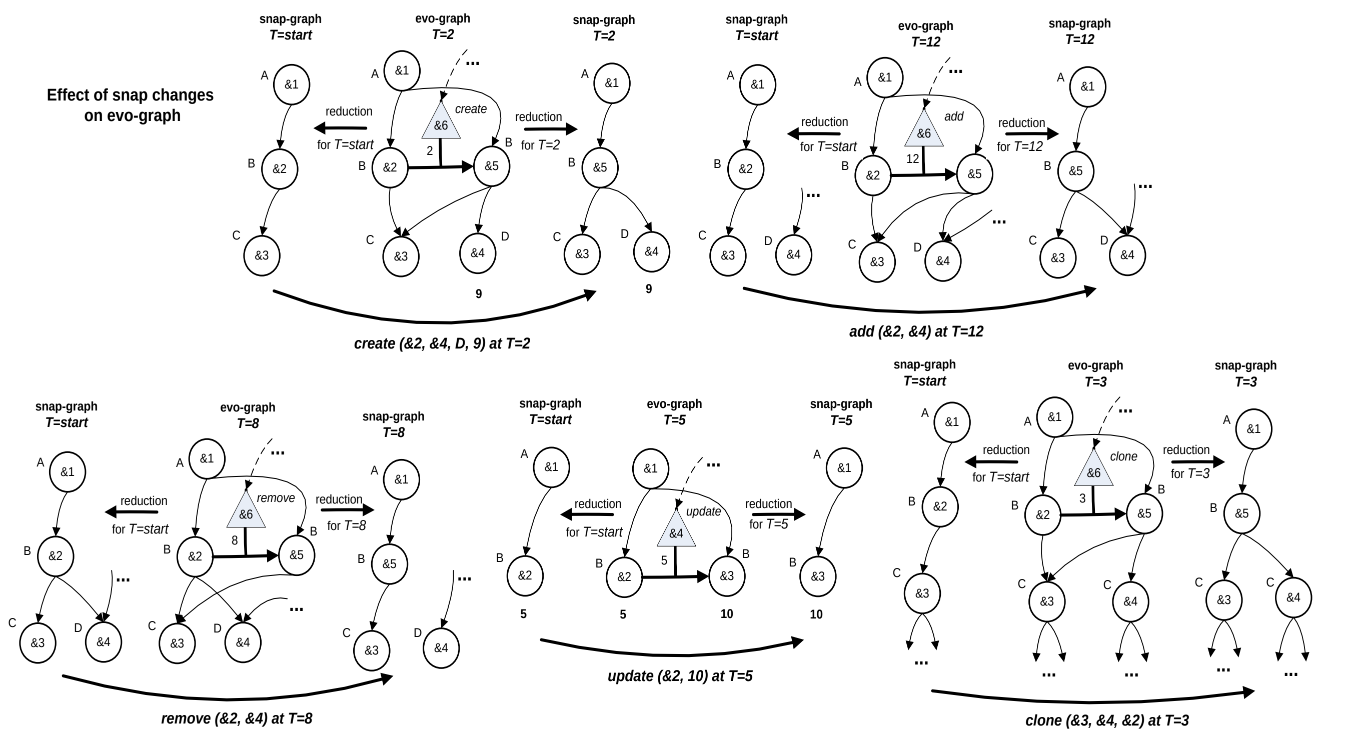

Figure 2. Effect of snap change operations on the evo-graph.

- *applyComplexChange(v_D^1, v_D^2, v_C, v_C^p, label, timestamp, \{v_C^1, v_C^2, ..., v_C^n\})*. This operation "evolves" node v_D^1 to node v_D^2, as the result of applying a complex change operation on the snap-graph. First, a new atomic data node v_D^2 with the same label as $v_D^1$and the default value (empty string) is created, and is connected as a child of all the current parents of v_D^1. A new *complex* change node v_C with the *label* and *timestamp* is created, and is connected as a child of the complex change node $v_C^p \epsilon V_C^c$. The *label* is the name of the complex change and can be any string. After that, v_C is connected as a parent of the change nodes $\{v_C^1, v_C^2, ..., v_C^n\}$. Finally, a new evolution edge $e=(v_D^1, v_C, v_D^2)$ is created between the data nodes v_D^1, v_D^2 and the change node v_C.

Note that we employ two separate evo-graph operations for applying snap-graph basic and complex changes. For complex changes, the *applyComplexChange* is used, which creates a new *complex* change node, a new version for the affected data node, a new evolution edge connecting the change node and the two data node versions and finally connects the complex change node as the parent of its constituent change nodes. For basic changes, the *applyAtomicChange* is used, which creates a new *atomic* change node, a new version of the data node that is affected by the change, and a new evolution edge. The exact implementation of each snap change in terms of evo-graph operations is given in TABLE II. .

For each snap change in TABLE II. , a *timestamp* is given (appears as *t*) and, if this change is part of a complex change, the parent complex change (v_C^P) is also specified. If no parent complex change is specified, we assume the parent is the change root r_C. Note, that all snap change implementations in TABLE II. start with *applyAtomicChange*, which creates the corresponding change node and the associated data node in evo-graph.

TABLE II. ACCOMMODATING SNAP CHANGES IN EVO-GRAPH.

	create **(v_D^P, v_D, *label, value*), *t*, v_C^P**
1	{ applyAtomicChange(v_D^P, $v'_D{}^P$, ‘’,v_C, v_C^P, ‘create’, t);
2	for $v_i \in$getCurrentChildren(v_D^P)
3	addDataEdge ($v'_D{}^P$,v_i);
4	*// create the new data node* and *connect it to the new parent node*
5	addDataNode *($v'_D{}^P$, v_D, label, value);* }
	add **(v_D^P, v_D), *t*, v_C^P**
1	{ applyAtomicChange(v_D^P, $v'_D{}^P$, ‘’,v_C, v_C^P, ‘add’, t);
2	*//connect the new parent node to all current children plus v_D*
3	for $v_i \in$(getCurrentChildren(v_D^P)$\cup v_D$)
4	addDataEdge ($v'_D{}^P$,v_i) ; }
	remove **(v_D^P, v_D), *t*, v_C^P**
1	{ applyAtomicChange(v_D^P, $v'_D{}^P$, ‘’,v_C, v_C^P, ‘remove’, t);
2	*//connect the new parent node to all current children except for v_D*
3	for $v_i \in$(getCurrentChildren (v_D^P)-v_D)
4	addDataEdge ($v'_D{}^P$,v_i); }
	update **(v_D, *newValue*), *t*, v_C^P**
1	{ applyAtomicChange(v_D, v'_D, newValue,v_C, v_C^P, ‘update’, t) }
	clone **(v_D^P, v_D^{source}, v_D^{clone}), *t*, v_C^P**
1	{ applyAtomicChange(v_D^P, $v'_D{}^P$, ‘’,v_C, v_C^P, ‘clone’, t);
2	for $v_i \in$(getCurrentChildren *(v_D^P)*
3	addDataEdge ($v'_D{}^P$,v_i);
4	*//clone the source data node*
5	addDataNode *($v'_D{}^P$, v_D^{clone}, v_D^{source}.label, v_D^{source}.value);*
6	*//create a deep copy of the cloned node*
7	for $v_i \in$getCurrentChildren (v_D^{source})
8	addDataNode(v_D^{clone}, v'_i, , v_i.*label*, v_i.*value*);
9	*repeat step* 7 *for v_D^{source} = v_i and v_D^{clone}*=v'_i }

III. Implementation and Evaluation

A. The C2D Framework

We have implemented all above concepts into the C2D (standing for Complex Changes in Data evolution) framework. C2D has been developed in Java, on top of Berkeley DB XML [3], an embedded XML database used to manage the evoXML representation of evo-graphs. In C2D, changes applied on the snap-graph are fed into a process that populates the evo-graph. A snap change is always applied on the current snap-graph (represented in XML in C2D), which is actually produced as a reduction [16] of the evo-graph for the time instance T=*now*. This flow is depicted in Fig. 3. The top layer in Fig. 3 is the *view layer*, where changes are launched. The purpose of the *logical model* layer is to guide the translation processes between the view layer and the *storage representation layer*, where changes actually take place.

Change operations on the evo-graph are implemented as XML update operations on the corresponding evoXML. Expressing evo-graph operations with the XQuery Update language is straightforward. For example the *addDataNode (&17, &19, "type", "non insulin dependent")* operation is expressed with the following XQuery Update *insert* expression on the evoXML.

```
insert node <type evo:id="19">non insulin  dependent </type>
into
/evo:evoXML/evo:DataRoot/Diabetes/categories/cat[evo:id="17"]
```

B. Experimental setting

Our goal was to examine how our approach depends on a number of factors that characterize the data. We first examined the space efficiency of evoXML for various configurations, regarding: the structure of the initial XML tree, the type of snap changes, and the selectivity of the elements. We also examined the performance of the reduction process with respect to the size of the evoXML file. Note that the comparison with other versioning approaches [4], [6], [7] was not pursued, as these works do not consider the role of changes as first class citizens in storing and querying evolving data.

Experiments were performed over synthetic XML data, on a PC with Intel Core 2 CPU 2.26 GHz, and 4.00 GB of RAM. The initial XML file was generated with [19] and contained about 10^5 elements, over which 10^4 snap changes were sequentially applied as XQuery Update statements. A new version was assumed after every 1000 changes; therefore 10 successive versions have been created for each setting. We recorded the size (in terms of the number of XML elements) of each "snap" version, and the size of the evoXML file at the same instance. Furthermore, we examined the performance of the reduction process for the current snapshot (T=*now*), and the initial snapshot (T=*start*).

Regarding the structure of the initial data, we used two XML files with the same number of elements: (a) one corresponding to a snap-graph with a "deep" tree structure (denoted s_1) with five levels and elements having a fan-out of 10, and (b) a file with a "broad" tree structure (denoted s_2) with only two levels and elements with a fan-out of

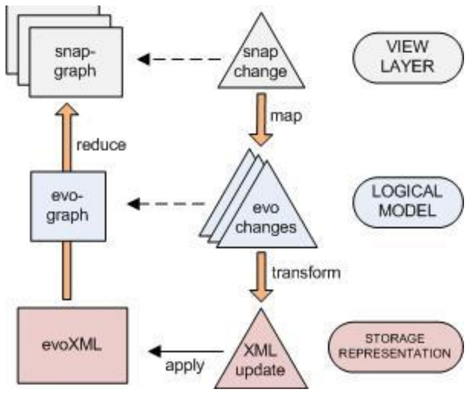

Figure 3. C2D framework overview.

about 330 elements. We have applied three sets of snap changes: (a) equal percentage for all changes except *clone* (denoted t_1), (b) 80% *update* and 20% *create* and *remove* (denoted t_2), and (c) equal percentage for all changes including *clone* (denoted t_3). Finally, concerning elements selectivity, changes have been applied either on all elements (denoted n_1) or on a fixed set of pre-selected elements so that each element is affected by 5 changes on average per version (denoted n_2).

We have examined the following combinations of the above parameters: (t_1n_1), (t_3n_1), (t_2n_1), and (t_2n_2) for each of s_1, s_2. t_1n_1 captures the typical case when random changes are uniformly applied on all elements. t_3n_1 is similar to t_1n_1, but it also includes *clone*. We have separately examined the *clone* operation, as it may arbitrarily result in the addition of a large amount of data. t_2n_1 captures the case where most (80%) change operations are *update* on random leaf elements, and only 20% are create or *remove*. Finally, t_2n_2 is like the previous case except that changes are concentrated on a pre-selected fixed set of elements.

Intuitively, we expect that the size of the evoXML depends on the number of snap changes performed. We also expect that it depends on the average fan-out of the snap-graph, while it remains insensitive to its average height. This is due to the way that each snap change operation is implemented on the evo-graph. Next, we present and discuss the results.

C. Results and Discussion

In Fig. 4 (a) and (b) we present the evoXML sizes per version. Subsequently, we discuss how this size is affected by the aforementioned configurations parameters.

File structure. For all configurations, better space efficiency is achieved for s_1. For smaller fan-outs (s_1), the evoXML has a smoother increase in size than for large fan-outs (s_2). A snap change occurring on an element adds *evo:ref* elements for all of its children (i.e. fan-out) that are still valid in the new version. Hence, the increase in the evoXML size is relative to the average fan-out.

Type of changes. t_2 outperforms t_1 and t_3. The majority of changes in t_2 are *update*, which have a smaller impact on the evoXML size. Again, the key point is the number of new elements that each change adds. Observe from TABLE II. that all changes add at least two new elements;

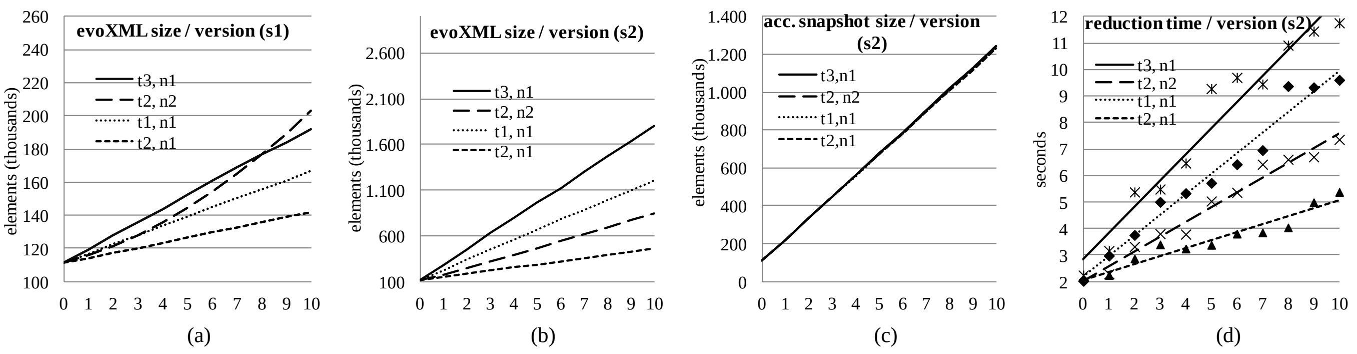

Figure 4. evoXML size (a), (b), accumulative snapshot size (c) and current snapshot reduction time (d) per version for various configurations.

one evolved data element and one change element. *update* adds only these two elements, whereas *create* and *add* insert one additional element for the new child, plus *evo:ref* elements for its siblings. *remove* results in inserting *evo:ref* elements in the evoXML for all the siblings of the removed element. Finally, *clone* adds a variable number of elements according to the height and average fan-out of the subtree that is cloned. On the other hand, the percentage of *create* and *remove* in t_1 is higher. In t_3, the use of *clone* further increases the file size by creating a deep copy of the subtree of the elements on which it is applied.

*Selectivity of elements.*Applying changes randomly on all elements (n_1) seems to have a smoother impact on the increase of the file size (e.g., compare t_2n_1 and t_2n_2 for each of s_1, s_2). This is due to the fact that changes are uniformly distributed over all the elements. On the other hand, the increase is higher when changes are targeting a fixed set of elements (n_2). Changes in t_2n_2 are sequentially applied on the same elements, i.e., *create* is applied on the same elements, increasing the number of their children and thus the number of *evo:ref* elements to be inserted when a subsequent *create* occurs on the same element.

Overall, the evoXML size depends almost linearly on the number of the snap changes applied, given that the average fan-out is constant. Moreover, the increase rate of the evoXML size is proportional to the average fan-out of its elements. This is more evident in t_2n_2 for s_1, where the average fan-out of the elements sustaining changes increases significantly per version, resulting in a boost in the evoXML size, whereas in s_2 the fan out increase rate is much smoother.

In Fig. 4 (c) we present the accumulative size of the snapshots produced per version. This approach can be considered as an alternative to evoXML. For space reasons, we only depict the series for s_2, as s_1 shows a similar trend. The accumulative size of all snapshots per version is significantly bigger than the evoXML size, for all runs over s_1. The same holds for all configurations of s_2, except for t_3n_1 where many *evo:ref* elements are added in the evoXML file. Note that the overlap of the series is due to the small variance in the accumulative snapshot size between configurations.

Regarding the performance of our reduction algorithm, we have measured the time the reduction process takes for producing the current and the initial snapshots. The results for the current snapshot for s_2 are shown in Fig. 4 (d), where the mark signs are the recorded time values, and the series are the trends for each configuration. A safe conclusion is that the reduction time depends mostly on the evoXML size. For small file sizes, the reduction performs the same for all versions. In addition, the increase rates in time are similar for both the current and the initial snapshot, for both s_1 and s_2. Therefore, the time instance parameter of the reduction process does not affect the reduction performance.

Concluding, both space and time efficiency are mostly affected by the average fan-out, which deteriorates as more changes are applied. That is mainly because of the evo:ref elements that are added for all children of an element that "evolves". Still, our approach is much more efficient than retaining separately every different version. Future optimizations will take into consideration the above and will aim to encode evo:ref elements and to the overall compression of the file.

IV. RELATED WORK

Numerous approaches have been proposed for the management of evolving semistructured data. One of the early works [6] proposes DOEM, an extension of OEM capable of representing changes, such as *Create Node*, *Add Arc*, *Remove Arc* and *Update Node*, as annotations on the nodes and the edges of the OEM graph. In [10], the authors employ a *diff* algorithm for detecting changes between two versions of an XML document and storing them either as edit scripts or deltas. For each new version, they calculate the deltas with the previous and retain only the last version and the sequence of deltas. A similar approach is introduced in [7], where instead of deltas calculation, a referenced-based identification of each object is used across different versions. New versions hold only the elements that are different from the previous version whereas a reference is used for pointing to the unchanged elements of past versions. In [9] the authors propose MXML, an extension of XML that uses context information to express time and models multifaceted documents. Recently, there are works that deal with change modeling [15] and detection [13] in semantic data, in which the aforementioned problems are applied to ontologies and RDF.

Most approaches employ temporal extensions for the lifespan of different versions of documents. In [1], [6], the authors enrich data elements with temporal attributes and extend query syntax with conditions on the time validity of the data. In [14], the authors model an XML document as a directed graph, and attach transaction time information at the edges of the graph. Techniques for evaluating temporal queries on semistructured data are presented in [8], [18]. In [8] the authors propose a temporal query language for adding valid time support in XQuery. In [18] the notion of a temporally grouped data model is employed for uniformly representing and querying successive versions of a document. In [11], the authors extend this technique for publishing the history of a relational database in XML and employ a set of schema modification operators (SMOs) for representing the mappings between successive schema versions. In [1] the problem of archiving curated databases is addressed. The authors develop an archiving technique for scientific data that uses timestamps for each version, whereas all versions are merged into one hierarchy. This is in contrast with approaches that store a sequence of deltas and apply a large number of deltas for retrieving backwards the history of an element. Lastly, [5] deals with provenance in curated databases. All user actions for constructing a target database are recorded as sequences of insert, delete, copy and paste operations stored as provenance links from current data towards previous versions of the target database or external source databases.

Compared to the above approaches, our model introduces a change-based perspective for evolving data, in which changes are not derived by data versions but are modeled as first class citizens together with data. In our view, changes are not described through diffs or transformations with edit scripts between document versions, but are complex objects operating on data, and exhibit structural, semantic, and temporal properties. Change-centric modeling of evolving semistructured data can provide additional information about *what*, *why*, and *how* data has evolved over time.

V. Conclusions

In this paper, we showed how a data model called evo-graph can be used to progressively capture the structure of changes and the history of data. We believe that capturing structured changes within a data model enables a range of very useful queries on the provenance of data, and on the semantics of data evolution. We defined basic and complex changes over snap-graph, and described the process of building evo-graph step by step, as changes occur on the current snap-graph. We outlined C2D, a framework based on XML technologies that implements the ideas presented in this paper. We evaluated C2D using synthetic XML data for its space and time efficiency, and discussed the results.

Acknowledgment

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

[1] T. Amagasa, M. Yoshikawa, S. Uemura, “A Data Model for Temporal XML Documents”, In DEXA 2000.

[2] P. Amornsinlaphachai , N. Rossiter and M. A. Ali, “Translating XML Update Language into SQL”, Journal of Computing and Information Technology, 2006, 2, 91–110.

[3] Berkeley DB XML. http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html. 19 June 2012.

[4] P. Buneman, S. Khanna, K. Tajima, W.C. Tan, ”Archiving Scientific Data”, ACM Transactions on Database Systems, Vol. 20, pp 1-39, 2004.

[5] P. Buneman, A. P. Chapman, J. Cheney, “Provenance Management in Curated Databases”, In SIGMOD’06.

[6] S. Chawathe, S. Abiteboul, J. Widom, “Managing Historical Semistructured Data”, Journal of Theory and Practice of Object Systems, Vol. 24(4), pp.1-20, 1999.

[7] S-Y. Chien, V. J. Tsotras, C. Zaniolo, “Efficient Management of Multiversion Documents by Object Referencing”, In VLDB 2001.

[8] D. Gao, R. T. Snodgrass, “Temporal Slicing in the Evaluation of XML Queries”, In VLDB 2003.

[9] M. Gergatsoulis, Y. Stavrakas, “Representing Changes in XML Documents using Dimensions”, In 1st International XML Database Symposium, (XSym 2003).

[10] A. Marian, S. Abiteboul, G. Cobena, L. Mignet, “Change-Centric Management of Versions in an XML Warehouse”, In VLDB 2001.

[11] H.J. Moon, C. Curino, A. Deutsch, C.Y. Hou, C. Zaniolo, “Managing and querying transaction-time databases under schema evolution”, In VLDB 2008.

[12] National research council - Committee on Frontiers at the Interface of Computing and Biology. Catalyzing Inquiry at the Interface of Computing and Biology. Edited by J. C. Wooley, H. S. Lin., National Academies Press, 2005.

[13] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, V. Christophides, “On Detecting High-Level Changes in RDF/S KBs”, In ISWC 2009.

[14] F. Rizzolo, A. A. Vaisman, “Temporal XML: modeling, indexing, and query processing”, In VLDB J. 17(5): 1179-1212 (2008).

[15] F. Rizzolo, Y. Velegrakis, J. Mylopoulos, S. Bykau, “Modeling Concept Evolution: a Historical Perspective”, In ER 2009.

[16] Y. Stavrakas, G. Papastefanatos, “Supporting Complex Changes in Evolving Interrelated Web Databanks”, In In CoopIS 2010.

[17] Y. Stavrakas, G. Papastefanatos, “Using Structured Changes for Elucidating Data Evolution”, In DaLi’11 (with ICDE 2011).

[18] F. Wang, C. Zaniolo, “Temporal Queries in XML Document Archives and Web Warehouses”, In TIME 2003.

[19] Xmlgener: Synthetic XML data generator. http://code.google.com/p/xmlgener/.

[20] XQuery Update Facility 1.0. http://www.w3.org/TR/xquery-update-10/, W3C Recommendation, 17 March 2011.