University of Konstanz
Department of Computer and Information Science

Master Thesis

A Visual Analytics Approach for Comparing Tree-Structures.

in fulfillment of the requirements to achieve the degree of

Master of Science (M.Sc.)

Johannes Lichtenberger
Matriculation Number :: 01/584875
E-Mail :: ⟨firstname⟩.⟨lastname⟩@uni-konstanz.de

Field of Study :: Information Engineering
Focus ::
Topic ::

First Assessor :: *Prof. Dr. M. Waldvogel*
Second Assessor :: *Jun.-Prof. Dr. Tobias Schreck*
Advisor :: *M.Sc. Sebastian Graf*

dedicated to...

Abstract. Todays storage capabilities facilitate the accessibility and long term archival of increasingly large data sets usually refered to as "Big Data". Tree-structured hierarchical data is very common, for instance phylogenetic trees, filesystem data, syntax trees and often times organizational structures. Analysts often face the problem of gathering information through comparison of multiple trees. Visual analytic tools aid analysts by combining visual clues and analytical reasoning. Visual representations are ideal as they tend to stress human strength which are great at interpreting visualizations.

We therefore propose a prototype for comparing tree-structures which either evolve through time or usually share large node-sets. Our backend Treetank is a tree-storage system designed to persist several revisions of a tree-structure efficiently. Different types of similarity measures are implemented adhering to the well known tree-to-tree edit problem.

The aggregated tree-structure is input to several interactive visualizations. A novel Sunburst-layout facilitates the comparison between two revisions. It provides several interaction options such as zooming as well as drilling down into the tree by selecting a new root node. Using hierarchical edge bundles to visualize moves reduces clutter from edge crossings.

Several filtering-techniques are available to compare even very large tree-structures up to many hundred thousand or even millions of nodes. Small multiple displays of the Sunburst-layout aid the comparison between multiple trees.

A short evaluation and a study of three application scenarios as well as performance evaluations proves the applicability of our approach. It surpasses most other approaches in terms of generability and scalability due to our database driven approach which allows for a fast ID-based difference algorithm optionally using hashes for filtering changed subtrees.

Table of Contents

List of Figures

List of Tables

Todo list

1 Introduction

1.1 Motivation

Ever growing amounts of data require effective and efficient storage solutions as well as highly scalable, interactive methods to gain new insights through exploratory analysis or to prove assumptions. Almost all data is subject to change. Nowadays storage is cheap and adheres to Moore's law[2] of doubling about every 18 months supporting the storage of several snapshots of time varying data. Furthermore existing storage techniques minimize the impact of storing such potentially very large data-sets.

Hierarichal information in the form of tree-structures is inherent to many datasets. It is almost always mapped through primary-/foreign-key relations in relational databases. Whereas this might be sufficient in many situations it introduces an additional artificial mapping. Using either a graph-DBMS for directed acyclic graphs (DAG)s or a native XML-DBMS for tree-structures facilitates a straight forward approach of storing data as well as efficient traversal methods and other domain specific advantages (for instance Dijkstra's algorithm for shortest path search in graph databases and most often extensive XQuery support in XML-DBMS).

Comparison of tree-structures In order to be human readable every tree-structure has to be serialized in some form. Character based line by line comparsion difference-tools as for instance used within Subversion (SVN) or the GNU diff tool to compare serialized textual tree-structure representations most often does not add up. Even though most of them colorize the differences based on character differences or provide other limited graphical representations of the computed differences they are not able to recognize the tree-structure and certain domain specific characteristics. For instance XML (Extensible Markup Language), which is a human readable meta markup language, exemplary for tree-structures in general and used in our prototype, has some inherent features which can not be recognized by such tools. Among those are the *lack of semantic differences* in case two XML documents only differ by an arbitrary amount of whitespace between attributes, namespaces[1] and elements or the permutation of attributes. Changes from empty elements to start tag, end tag sequences (`<root/>` to `<root></root>`) or inversely must not be considered as well. Furthermore moves of nodes or subtrees and differences in the order of child nodes can not be determined. The major disadvantage however attributes to the tree-structure itself. Node-boundaries ca not be recognized as these tools incorporate no knowledge about the structure itself. A comparison between two very simple XML documents (or two versions thereof) done with GVim, which utilizes a line by line character based comparison algorithm is illustrated in Fig. 1. Several of the aforementioned deficiencies are depicted in this simple example.

[1] special kind of attributes

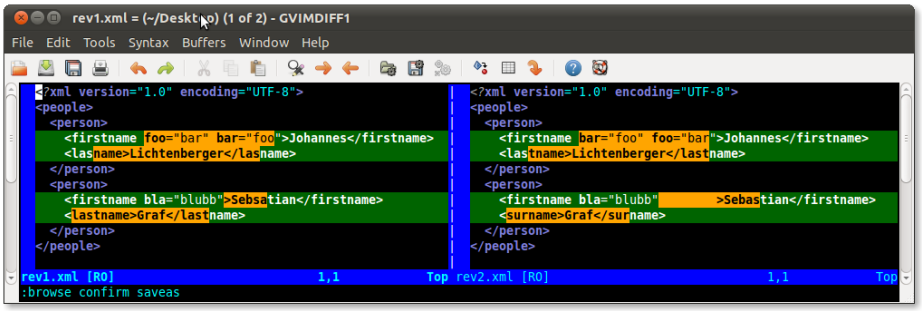

Fig. 1. Diff example illustrating the deficiencies of line by line character based diff tools

1.2 Problem Statement

Analysts often face the problem of having to compare large tree-structures. While coping with rapidly increasing amounts of data is effectively solved by means of Treetank, comparison requires sophisticated methods on top of it.

Generally two cases of tree-structures have to be distinguished which our system must be capable of.

- Tree-structures evolving naturally through applying changes.
- Similar tree-structures.

The research task adressed in this thesis is defined as:

Definition 1 *Provide methods to help analysts to quickly gain knowledge from comparing tree-structures.*

1.3 Approach

A promising solution to the task at hand is to use methods from "Visual Analytics", a term coined by James J. Thomas in [3]. Thomas states that Visual Analytics is "the science of analytical reasoning facilitated by visual interactive interfaces.". Thus we provide (at least semi-)automatic analytical methods facilitated by an interactive visual interface. Analytical methods are inevitable to compare trees in the first place. Furthermore interesting patterns can be revealed by custom `XPath` queries.

Whereas hierarichal visualizations have been studied for some time and sophisticated representations have been found, Visual Analytics of comparing tree-structures just recently gained momentum. Visualizations stress human strengths.

Value of visualizations Francis J. Anscombe reveals the value of graphs (which is generalizable to every (useful) kind of data-visualization) by illustrating in a simple example with four data sets, why graphs are essential to good (statistical) analysis. Using statistical calculations from a typical regression program shows that each computation yields the same result even though fundamental differences are visible on first glance once plotted. Furthermore human brains are trained to interpret visual instead of textual content which is another facet illustrated by this example. It is almost impossible to gain further insights running through the printed out form of these data sets [4].

Generalization and refinement of our research task While several data mining tools are available which specify on specific tasks, tree-structures are flexible and come in many flavors. XML is a meta markup language which is capable of describing all kinds of rooted, labeled trees. Thus it is used by our prototype. In fact it is a semi structured, flexible, meta markup language. XML in stark contrast to relational data does not have to adhere to a schema or structure, which has to be planned and implemented beforehand. Due to that it is mandatory that the visual interface offers great flexibility and thus is not restricted to a special use case.

The high level goal defined in the last section (1.2) can be divided into:

- Preprocessing and import of differences.
- Structural comparison based on `insert-/delete-`operations.
- Comparison of non-structural data (for instance `TextNode` values).
- Extend with `replace`, `update` and `move`-operations (optional).
- Provide visualizations to quickly gain insights into which subtrees/nodes have been changed.

1.4 Contributions

The main aim of this thesis is the research and development of an interactive visual interface supporting analysts to determine changes in hierarichal tree-structures along with analytical methods to compute the differences.

In a nutshell this thesis provides the following computer science contributions:

- Preprocessing of realworld XML data, for instance the import of *Wikipedia* and monitoring changes in a specific Filesystem directory.
- Several contributions including compacting the storage and new edit-operations to support the implementation of an ID-based differencing algorithm and expressive visualizations. A new subtree-insertion operation based on a existing component speeds up hashing of subtrees considerably from $O(n^2)$ to $O(n)$ due to a simple postprocessing postorder traversal whereas n is the size of the nodes in the inserted subtree.
- Analytical methods (algorithms) to compute structural and non structural differences between similar or evolving tree-structures.
- Several views:

- A `TextView` which serializes an aggregated tree-structure to a syntax highlighted XML output. Furthermore only the visible area plus additionally space to add a slider is filled.
- A `SunburstView` facilitating comparison of tree-structures by a novel layout algorithm and several pruning techniques. Further interaction mechanisms like zooming/panning, a fisheye view, support of XPath-queries and several other techniques are provided as well.
- A `SmallmultipleView` supporting different modes (incremental, differential, a hybrid mode and sorted by similarity).

1.5 Conventions

Pseudocode which is used to illustrate algorithms in this thesis is based on a Java-like syntax as our prototype is based on Java. The following conventions in particular apply:

- The logical operator || from Java and other programming languages is denoted by *OR*.
- Similar the logical operator && is denoted by *AND*.
- Variable or reference assignments = are denoted by $\leftarrow$.

1.6 Outline

The work is structured as follows:

Chapter 2 provides an overview of algorithms to compute differences in tree-structures. Next, research efforts in visualizing differences of trees are examined. The chapter concludes with a summary of the visualizations which are examined in respect to various attributes.

Chapter 3 starts off with a short description of Treetank and its encoding. Next, as most tree-structures do not use unique node-IDs we compute differences based on the FMSE-algorithm. FMSE matches nodes based on similarity-functions for leaf- and inner-nodes in the first place and in subsequent steps modifies a tree with as few edit-operations as possible to transform the first tree into the second tree or the first version of a tree into the second. Thus implementation of FMSE is described as well as numerous extensions to Treetank to support the implementation. Once the data is imported we use an internal diff-algorithm based on unique node identifiers to compare several trees, which is described thereafter. The chapter concludes with performance measures of our ID-based algorithm and concludes with a short summary.

Chapter 4 is introduced with an overview of the GUI structure. Detailed descriptions of our visualizations follow. Furthermore several interaction mechanisms are examined.

Chapter 5 demonstrates the feasability of our approaches based on real world data.

Chapter 6 discusses our approach in relation to the State-of-the-Art.

Chapter 7 summarizes the results and provides suggestions for future work.

2 Preliminaries and State-of-the-Art

2.1 Introduction

Today's storage capabilities facilitate the growing amount of data which is most often collected and stored without filtering or preprocessing. One of the consequences is the information overload problem defined as:

- Irrelevant to the current task at hand.
- Processed or presented in an inappropriate way.

To turn these issues into advantages the science called "Visual Analytics" recently became popular.

James J. Thomas and Kristin A. Cook coined the term "Visual Analytics"[3] and defined it as: "Visual analytics is the science of analytical reasoning facilitated by interactive visual interfaces." It combines (semi-)automatic analytical analysis with interatice visualization techniques, thus emphasizes both cognitive human and electronical data processing strenghts.

Whereas the information seeking mantra is described as "overview first, zoom/filter, details on demand" Keim et al defined the Visual Analytics mantra as:

"Analyse First - Show the Important - Zoom, Filter and Analyse Further - Details on Demand"[1]

This implies and confirms the important role of humans in the analysis process. As meantioned in the introduction humans are trained to interpret visual impressions but often fail in the same way to construe inappropriate representations.

Our Visual Analytics pipline is largely influenced by the proposal of Keim et al. which is depicted in Fig. 2.

Comparing tree-structures by a Visual Analytics approach requires analytical reasoning through the computation of differences in the first place. In order to support large tree-structures we decided to use a secure treebased storage system.

2.2 Storage backend

Treetank is an effective and efficient secure storage system tailored to revisioned tree-structures. Currently it supports the import of XML documents which is commonly referred to as *shredding*. To process stored data the W3C recommendations XPath 2.0, XQuery 1.0 and XSLT 2.0[2], as well as a cursor like Java-API using transactions is supported. The architecture is based on four exchangeable vertical aligned layers.

This architecture supports *Snapshot-Isolation* through Multiversion Concurrency Control (MVCC) based on the page- and transaction-layer. Furthermore

[2] parts of the XPath 2.0 recommendation have been implemented, alternatively the Saxon XPath 2.0 binding can be used which also offers the XQuery 1.0 and XSLT 2.0 support

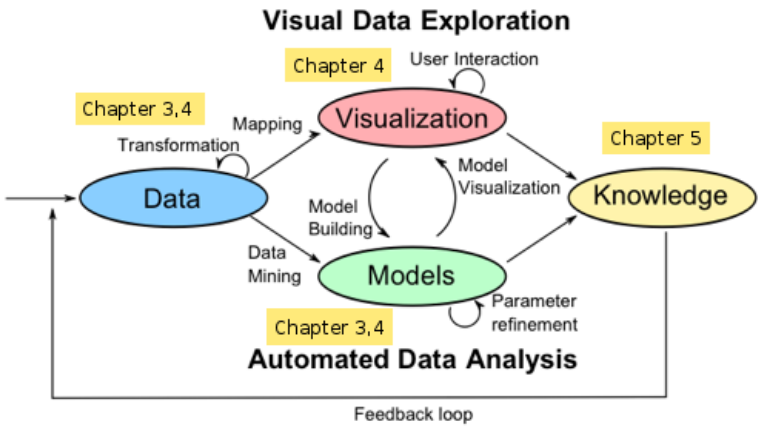

Fig. 2. Visual Analytics Process proposed by Keim et al. Presented in [1].

the well known ACID properties are supported. While importing large XMark-instances [5] which are commonly used for benchmarking we encountered a space-overhead due to our pointer based approach. Appendix A.1 details a number of enhancements to the persistent storage made which came up during writing this theses. Furthermore the the ACID properties are briefly described together with a few new consistency constraints tailored to the XML storage.

Treetank does not provide deltas. The revisioning algorithms merge *NodePages*, which contain nodes, with the same unique ID together and override existing nodes with the latest version. Deleted nodes are introduced to guarantee the correctness after the merge-phase. The combination of NodePages relies on the specific revisioning algorithm. Thus, the merge-phase of NodePages usually refers to the latest full dump of all NodePages or the previous revision. As a direct consequence we are not ably to simply use the page-layer or a delta between consecutive revisions. However the introduction of hook-mechanisms in a future version of Treetank will facilitate the generation of deltas. As a result the differences between consecutive revisions in the future will always reflect the update-operations. Yet, such a delta can not be used comparing other revisions.

One of the most interesting properties of Treetank for our purpose is the *versioning* and unique node-IDs as well as query-capabilities.

The next section describes existing ID-less algoritms focused on XML-comparison. A short study and summary of existing visualization techniques follows.

2.3 Analysis of differences

Line by line textual diffs are based on algorithms which solve the Longest Common Subsequence (LCS) problem. Whereas they are sufficient to track changes in flat text-files, tree-structures need more sophisticated methods as pointed out in the introduction.

The *Extendable Markup Language* (XML) is a textual data format for encoding and structuring documents in machine- and human-readable form. Its inherent data structure is a rooted, ordered, labeled tree.

Definition 2 *A rooted* ***tree-structure*** *is an acyclic connected graph, which starts with a root node whereas every node has zero or more children with the exception of the root-node having exactly one parent-node. We define a tree T as $T = (N, E, root(T))$ whereas N denotes all nodes, E denotes edges, the relation between child- and parent-nodes whereas each child except the root-node has exactly one parent node, root(T) defines a root-node which is the only node having no parent.*

Definition 3 *A rooted, ordered, labeled tree is a tree-structure which extends the rooted tree-definition by defining a specific order for child nodes (that is extending the parent/child edge relation E) and a label for each node. Furthermore each node has a label. Thus T is an ordered, labeled, tree if $T = (N, E, root(T), \Lambda(n) \in \Sigma)$. Σ is a finite alphabet and n is a node in the tree.*

Thus a tree is more restricted than a hierarchy based on a directed acyclic graph (DAG) in which every node except the root[3] might have one or more parent nodes.

Many algorithms have been developed to determine differences in tree-structures for instance to provide deltas, which represent a compact version of the changes to the original document.

Next, some essential terms are defined to set the stage for the upcoming sections.

The tree-to-tree correction problem tries to transform a source tree into a destination tree by edit-operations.

Definition 4 *An edit-operation is an atomar operation which changes a tree.*

A delta/edit-script is defined as:

Definition 5 *A delta/edit-script is a sequence or set of (elementary) edit-operations which when applied to one version v1, yields another version v2.*

Definition 6 *A symmetric delta is a directed delta which is invertible.*

In the following we use the term *delta* and *edit-script* interchangeably in the generic form meaning directed delta. Each edit operation is usually defined with a fixed cost (usually unit cost).

Definition 7 *A minimal edit script is a minimum cost edit script.*

[3] which has no parent

Besides providing a minimal or close to minimal edit script further metrics of a diff-algorithm are the CPU runtime and the compactness of the delta in terms of storage-space (e.g. it is in most cases sufficient to define edit operations on subtrees, such as a delete- or move-operation which usually removes the whole subtree).

Some of the most popular approaches to detect differences in XML-documents and to generate a delta are described next.

DocTreeDiff[6] is designed for difference detection in document-centric XML documents. The algorithm computes the Longest Common Subsequence (LCS) on leaf nodes which are on the same level using hash values. Subsequently ancestor nodes which differ are updated. Then inserted and deleted nodes are determined based on unmatched leaf nodes in the respective tree. As only leaf nodes in the first step are matching candidates which are on the same level, that is the number of nodes on the path up to the root node, the algorithm does not detect same subtrees if a inner node is added (or a leaf node is inserted and a subtree is moved beyond the new node which is sufficient for a tree model which only allows insertions on leaf nodes). Furthermore moves are detected in a postprocessing step instead of applied on the fly. The runtime complexity is $O(leaves(T)D + n)$ whereas T is the sum of nodes in both trees and D is the number of edit operations. The space complexity is $O(T + D)$.

DeltaXML[7] Elements are matched according to element types, the level in the tree and the Longest Common Sequence (LCS). Furthermore matching PCDATA nodes may optionally be prefered. Similar, attribute-IDs in the deltaxml-namespace may be used to mark nodes with unique IDs. Marking all nodes with an ID generates a minimum edit script. However, a complete description of the algorithm is not published.

XyDiff[8] Cobena et al. present in [8] a fast heuristic algorithm in the context of Xyleme, an XML database warehouse. Signatures which are hash-values computed based on the value of the current node and all child signatures are used. Inserts and deletes are restricted to leaf nodes in spirit of Selkow's tree-model [9]. Based on heuristics large subtrees are matched and based on weights possibly propagated to parent nodes. In [6] several deltas are examined and the greedy subtree-algorithm yields large deltas. However, tuning parameters as the weights and how far to go up to match parent nodes is not considered which for sure affects the generated delta.

The CPU runtime of the algorithm is $O(n \log n)$ and the space complexity is $O(n)$ whereas n is the size of both documents.

LaDiff / Fast Match Simple Editscript (FMSE)[10] operates on different versions of LaTeX documents. It is developed in the context of LaTeXto

demonstrate and measure the feasability of an approach to detect changes in hierarichally structured information.

Chawathe et al. divides this task into two main problems:

the Good Matching problem is the problem of finding matches between the two trees, which are either equal for some predefined function or approximately equal.

finding a Minimum Conforming Edit Script (MCES) is the second obstacle. An *edit script* is a sequence of edit operations wh the source file which transform it into the target document once applied. Costs are therefore applied to every edit operation.

The algorithms used to solve these two problems operate on rooted, ordered, labeled trees. Four edit operations (`insert`, `delete`, `update` and `move` are defined with unit costs.

The algorithm proves to yield minimum edit scripts in case the assumption holds true that no more than one leaf node is considered equal to a predefined function which compares the values of leaf nodes and the labels match. XML does provide labels in the form of `QName`s for `element`- and `attribute`-nodes and a slightly restricted alphabet for `text`-nodes. Thus either text-node values have to be compared or `QName`s.

Thus the first criterium for leaf nodes is

$$compare(v(x), v(y)) \leq f \; such \; that \; 0 \leq f \leq 1 \tag{1}$$

Inner nodes are match candidates according to the forumla

$$\frac{|common(x,y)|}{max(|x|,|y|)} > t \; and \; label(x) = label(y) \tag{2}$$

$common(x,y) = \{(w,z) \in M | x \; contains \; w \; and \; y \; contains \; z\}$ whereas a

node x contains a node y if y is a leaf node descendant of x and $|x|$ denotes the number of leaf nodes x contains.

The threshold t is defined as $0.5 \leq t \leq 1.0$.

In a first step the good matching problem is solved by means of concatenating nodes/labels starting from bottom up and finding a LCS at each level of the tree. Furthermore if nodes are left which are equal according to the predefined function they are subsequently matched on each level. If the assumption does not hold which might be the case for several XML-documents, especially in data centric XML files the algorithm yields large output-deltas according to Lindholm et al.[11] and Rnnau et al.[6]. This is a direct result of the ambigiouty of the LCS as well as of the subsequent matching of nodes on every level. However this is a problem common to almost all differencing algorithms and can be minimized by proper definitions of the similarity functions for leaf- and innder-nodes.

After that in a breadth first traversal nodes are inserted, updated, moved and deleted. The children of each node are aligned based on the LCS once again.

Nodes which are matched but not in the LCS are moved. The order in which operations are applied to the source tree and the edit script is crucial to the correctness of the algorithm.

It is apparent that a large number of moves are appended to edit scripts in case the assumption that every leaf node in the old revision is similar to at most one leaf node in the old revision. If this assumption does not hold true the algorithm yields suboptimal deltas due to mismatches. A postprocessing step reduces other mismatches and thus move-operations such that children of matched nodes, which have not the same parent are tried to match with children of the same parent in the other tree, thus correcting some misaligned nodes. Note that this step can not reduce errors, which are propagated from mismatched leaf nodes up in the tree.

The runtime complexity is $O(n * e + e^2)$ and the space complexity is $O(n)$ whereas n is the number of unchanged nodes .

X-Diff[12] operates on unordered, labeled trees. Thus the order of child nodes does not matter. Despite using an unordered tree-model which is not suitable in many cases as for instance document centric XML furthermore updates on element nodes to the best of our knowledge are not defined. However updating an internal node is crucial due to otherwise potentially large subtree delete- and insert-operations whereas only a single node is updated.

The runtime is defined for the three steps defined in [12] separately:

1. **Parsing and Hashing**: $O(|T_1| + |T_2|)$
2. **Mapping**: $O(|T_1| \times |T_2|) \times max\{deg(T_1, deg(T_2)\} \times \log_2(max\{deg(T_1), deg(T_2)\}))$
3. **Generating Minimum-Cost Edit Script**: $O(|T_1| + |T_2|)$

Faxma[11] uses fast sequence aligning transforming the parsed documents into sequences of tokens. Subsequently the diff is computed using hashes with different window-sizes. Moves are handled through the combination of `delete/insert` pairs which is similar to the approach used by *DocTreeDiff* thus potentially leading to large

Summary The problem in common to all approaches is to efficiently compute a minimum or near minimum edit script to transform the first into the second tree. Unfortunately a guaranteed minimum edit script for the the tree-to-tree correction problem is known to be bound in runtime by
$O(nm\ min(depth(T1),\ leaves(T1))\ min(depth(T2),\ leaves(T2)))$, with n, m denoting the number of nodes of the trees $T1$, $T2$. Using heuristics speeds up the process but it does in most cases produce non optimal (minimal) edit scripts which might be counterintuitive to humans, because of mismatched nodes which have not been changed. Every diff-algorithm has its strength and pitfalls. Depending on the input and expected modification patterns some algorithms provide better results than others. Even though algorithms are compared in [] we are critical as the size of the delta and the amount of edit-operations might

	runtime comp.	**space comp.**	**tree model**	**move support**
DeltaXML	not published	not published	not published	not published
XyDiff	$O(n \log n)$	$O(n)$	ordered tree	yes
FMSE	$O(ne + e^2)$	$O(n)$	ordered tree	yes
X-Diff	$O(n^2)$	not published	unordered tree	no
DocTreeDiff	$O(leaves(T)D + n)$	$O(T + D)$	ordered tree	yes
Faxma	$O(n)$ (average) $O(n^2)$ (worst)	not published	ordered tree	no

Table 1. Comparsion of tree-to-tree difference algorithms.

not be the best All algorithms work best if leaf nodes can be descriminated very well. Comparing document oriented XML thus usually procudes better results in comparison to data centric XML in terms of minimum or near minimum edit-scripts/deltas.

Memory consumption is very important considering larger XML instances ranging from 1Gb and far above. Reducing the cost of computing the LCS which has a large memory footprint might be mandatory but also results in heuristics. A survey of the wide range of algorithms is summarized in [13]. Several algorithms are described and compared according to the attributes memory consumption, time complexity supported operations and ordered/unordered. Ordered/unordered denotes if anchestor/child relationships and the child order is considered (ordered) or not (unordered).

In summary a trade-off between the minimality of edit scripts/operations and the memory consumption as well as the time complexity of the algorithm exists. Furthermore no algorithm exists which outperforms and in respect to the edit-script cost produces always better results than the others while comparing trees of different domains and characteristics. It heavily depends on the change pattern of the input document.

2.4 Visualization of differences

Several visualization techniques have been proposed for hierarichal data ranging from simple node link diagrams, force directed layouts to space filling approachs. Recently database systems which are capable of storing hierarichal temporal data efficiently and therefore store snapshots of time varying data put forth the need to determine and visualize changes between several revisions such that analysts are able to answer time dependent questions like the ones raised in the motivating application examples.

While in the past it has been possible to map temporal hierarichal data to relational databases it required the storage of a full snapshot through foreign/primary key relations instead of just storing incremental or differential updates as well as the hierarichal mapping overhead.

Treevolution[14] visualizes the evolution of hierarchical data in a radial node-link diagram whereas each node can have arbitrary many parent nodes. Each Ring represents one snapshot. Inserted nodes are placed on the appropriate ring depending on the time of insertion. However edge crossings due to nodes having possibly more than one parent node result in visual clutter which complicate the analysis of the hierarichal relationship between inserted nodes and their parents. Label overplotting is a result from drawing the labels in one direction, however a simple interaction method improves on this by providing rotation.

Interactive Visual Comparison of Multiple Trees[15] The authors propose a prototype to compare multiple phylogenetic trees. Several views are available to analyse the trees on different levels of detail. A matrix view for instance displays pairwise tree-similarities based on a similarity score which takes overlapping subtrees into account. The similarity score depends on all nodes in a subtree including inner nodes instead of just determining overlapping leaf nodes. A histogram shows the score distribution among all nodes in all trees. The consensus tree is "a compact form of representing an 1:n comparison" in one tree. The score is "the average of the scores comparing a reference tree node against its best matching unit in all other trees". The last view is a Tree Comparison View which highlights all nodes in the subtree a user marks through a linking and brushing technique in all other trees. It is the only system which is capable of comparing multiple trees on different levels at the same time. However we assume that the quadratic runtime of comparing all nodes with all other nodes will be restricted to (many) small trees. Furthermore it is not meantioned how nodes are compared, but we assume unique labels or node identifiers are required as the prototype is proposed for phylogenetic trees.

Spiral-Treemap/Contrast-Treemap[16] A Treemap is a space filling approach which maximizes available screen space for the visualization. Most treemap layouts suffer from abrupt significant layout changes even if the underlying data changes were rather small. Tu et al. propose a new layout algorithm called *Spiral Treemap* to improve the layout stability arranging child nodes in spirals changing the orientation by 90 in the edges. Child-nodes are aligned along a spiral in each level beginning at the upper left corner. Therefore edit-operations as for instance inserts and `deletes` only affect local regions. However, we argue that it is not trivial to analyse strutural differences as they are not explicitly visualized in the Contrast Treemap and the texture distortion depends on the layout algorithm, whereas small changes are hardly visible. Furthermore the Spiral Treemap . Thus, labels the readability of labels is very good if the rectangles are not too thin which occurs frequently in large trees ranging from about 50000 nodes to a few hundred or even millions of nodes. Improving the aspect ratio of the rectangles results in Squarified Treemaps[17], which lack the property of ordered siblings. However child-nodes in trees are often ordered which is why Squarified Treemaps in general are only feasable in certain specific cases which do lack a semantic difference in node ordering.

TreeJuxtaposer[18] TreeJuxtaposer is a system designed to support biologists to compare the structures of phylogenetic trees. A new comparsion algorithm to determine matching nodes in near-linear average time has been developed. Perfect matching nodes have the same labels for each of their leaf nodes. Based on a simple similarity measure ($S(A, B)$ between two sets whereas A, B is defined as $\frac{A \cup B}{A \cap B}$) they propose a method to colorize edges of non perfectly matching nodes and a rectangular magnifier to emphasize changed nodes. The visualization itself contains several revisions side by side plotted in a node link diagram. Selections and rectangular magnifications are synchronized. TreeJuxtaposer uses a node link algorithm, therefore it shares the drawbacks of other node link visualizations such as *Treevolution* and the *Ripple presentation*. In comparison to space filling approaches further attributes as for instance value comparisons, subtreesizes and labels (besides some leaf labels) are not visualized. Furthermore the fast differencing algorithm to the best of our knowledge relies on unique node labels to support the region query on a two-dimensional label

Code Flows: Visualizing Structural Evolution of Source Code[19] Code Flows is proposed for determining and tracking changes in source code between several revisions. It is a space filling approach which uses horizontally mirrored icicles and therefore certain attributes of nodes can be visualized besides highlighting actual tree changes. Labels are readable in smaller trees or when zoomed in because of the rectangular layout which underlies an icicle plot. Due to the spline tubes matching nodes can be tracked very well through different revisions. Even code splits and merges are easily trackable. On the downside small code changes resulting in the addition or deletion of a few nodes might not be visible at first glance. Furthermore the spline connections between matched nodes leads to visual clutter due to overplotting when nodes are moved.

Ripple presentation for tree structures with historical information[20] The Ripple presentation has been developed to visualize both evolving hierarchies and categories. Concentric circles are used to indicate the evolving hierarchy through time. Each circle represents one point in time. Nodes are plotted in a special node link layout. The root node of each subtree is in the focus of the view. Leaf nodes are arranged in ascending order meaning older nodes are drawn on circles further away from the current root of the subtree. The angles of edges are application dependent and facilitate the clustering of categories through time. In the news articles example categories can be extracted from the content. For each child being in the same category the angle of the edge has to be in between the parent angle. Since the application examples require no diff-calculation and updates as well as deletions of nodes are not considered it is not useful to compare every aspect of changing tree structures. It suffers from a lot of clutter consequent to label overplotting as well. In common with Treevolution deletions and updates have not been considered since the example use cases to the best of our knowledge just add nodes and categories. Due to the fact that it is also a node link representation and not a space filling approach attributes of nodes can

	hierarchy	**space filling**	**readability**	**changes**
Spiral-/Contrast-Treemap	+	++[4]	++	++
Treevolution	+	-	+	+
Code Flows	+++	++	++	++
Juxtaposer	++	-	++	+++
Ripple Presentation	+	-	+	+
IVCoMT	+++	-	++	++

Table 2. Comparsion of tree-to-tree differences visualizations, whereas "-" is used to indicate the absence of an attribute and "+" to "+++" implies how good or bad the attribute is supported.

not be visualized. Thus it is best comparable to Treevolution, but because of the more complex layout algorithm it can group nodes according to categories.

Summary Recently, few interactive visualizations of the evolution of tree-structures or different similar trees have been proposed. Table 2 summarizes the visualizations according to several attributes. The first column denotes how well the hierarchy is represented. The second column indicates if a space filling approach is used and to which extend the whole display space is utilized. The third column characterizes how well labels as well as the whole visualization is readable. The last column is the most significant. It determines how well and to which extent changes are visualized. Even deletions are not considered in some cases which might be due to the use cases of the respective visualization. Note that the ratings range from "-", not present to "+++".

Most of the visualizations are tailored to specific tasks and are only partly useful for other applications. In fact besides adapting the diff-algorithm in **??** none of the proposed systems to the best of our knowledge is able to compare every kind of tree structure due to diff-algorithms which rely on domain specifics as unique node identifiers/node labels or on change detections which hook into a system. Furthermore we suppose that except TreeJuxtaposer and CodeFlows no other system is able to compare large trees. However, in CodeFlows the filtering of nodes depends on the level of detail (per class-level, function-level...). Thus a global view which filters relevant nodes is not available.

3 Analysis of structural differences

3.1 Introduction

This chapter describes the implementation of an ID-less algorithm (FMSE, described in the last chapter (2.3)) as well as a new ID-based algorithm which utilizes a preorder traversal on both trees to compare tuples of two nodes each time. The FMSE algorithm facilitates the import of differences between two tree-structures which do not incorporate unique node-IDs in the first place either to compare different, similar tree-structures or the evolution of a tree. Thus, our similarity measures are based on the tree-to-tree correction problem. The visualizations proposed in the next chapter rely on diff-algorithms which detect edit-operations/diff-types to transform a source tree into a target tree. A reference tree is initially imported in a database system. Subsequently the changes between this tree and either other trees or the evolution of the reference-tree in terms of edit-operations are stored as subsequent revisions. As described in the last chapter, the Fast Matching Simple Edit-Script algorithm depends on similarity-measures and does not require nor use unique node-IDs in our case. Thus, a minimum edit-sequence usually is not guaranteed. While importing the differences through FMSE, the database system, Treetank assignes unique stable node-IDs which are subsequently utilized by visualizations to support a fast linear-runtime difference-computation. Fig. 3 describes the import using FMSE and a subsequent invokation of a fast ID-based diff-algorithm. When collections are going to be imported, that is multiple revisions of one tree the differences are encountered through comparing the latest stored revision in the database backend with the next revision to import. In case of multiple similar trees, the algorithm compares the first revision stored with the next revision to import. Once imported unique node-IDs and optionally hashes facilitate a new fast diff-algorithm. The computed diff-tuples which include the diff-type, the compared nodes and their depth in the tree by comparing two nodes of both revisions each time in turn facilitate an aggregated tree-structure made up of both changed- and unchanged-nodes through collecting the diff-tuples in the model of a visualization as described in the next chapter (Chapter 4).

Both algorithms, the FMSE algorithm, which does not require unique node-IDs, used for importing differences and a fast ID-based diff-algorithm are implemented using Treetanks' transaction-based Java-API, the native secure tree-storage system, which is used as an integral part to demonstrate our approach. After a short description of Treetank the implementation of several new edit-operations is described to support the FMSE-algorithm and a compact, meaningful aggregated tree-structure. Note that a rich set of edit-operations also facilitates an expressive visualization. It is much more intuitive and meaningful to provide atomar `replace`- and `move`-operations to reflect changes between tree-structures than to simply use combinations of `delete`- and `insert`-operations without any association.

The next section describes the implementation of the ID-less diff algorithm called FMSE, described in general in the last Chapter (2).

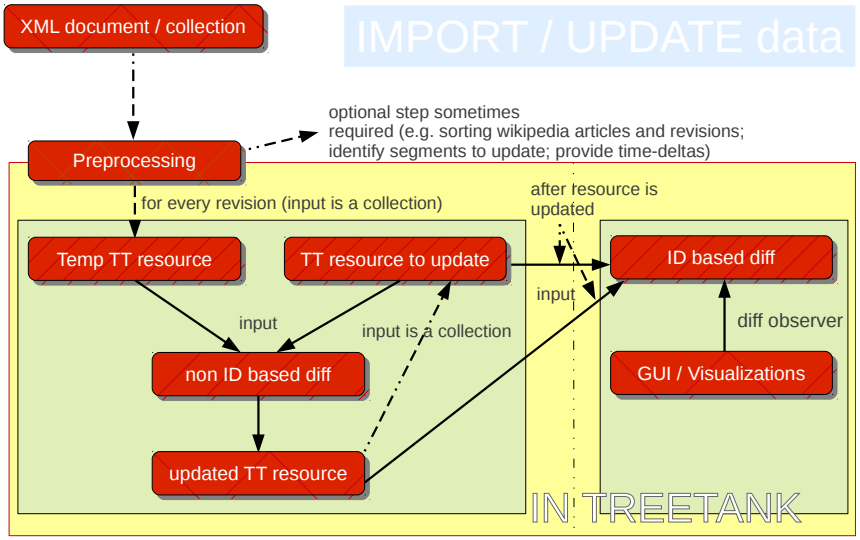

Fig. 3. Importing differences encountered through the FMSE ID-based diffing-algorithm.

3.2 ID-less diff-algorithm (FMSE) / Preprocessing

Preprocessing of raw data is a major task in every data processing pipeline. Besides data specific preprocessing, databases/resources which do not evolve through the Java-API of Treetank have to be imported. Note that it is very common to simply dump full revisions of temporal data, thus most often no direct deltas are provided which just have to be applied to a base revision. Furthermore similar distinct trees have to be compared. Both types often times do not include unique node-IDs and therefore must be compared using tree-to-tree comparison heuristics which try to determine and match the most similar nodes/subtrees.

As described in the introduction the FMSE algorithm described in the Chapter 2 is implemented. The reasons for choosing FMSE are based on three properties: (1) it has been proven to be successfully implemented a few times (specifically for XML-documents) [21], [22], (2) it utilizes a move-operation and (3) supports applying edit-operations/changes on the fly. The move-operation is one of the new edit-operation Treetank supports and very lightweight. It is defined for subtrees. Only local nodes are affected as well as the ancestor nodes of the node which moves (before and after the move).

The Nodes are matched based on a bottom-up traversal searching for the Longest Common Subsequence (LCS) of matching nodes on each level. Predefined functions determine the similarity of nodes/subtrees as described in Chapter 2, used by the LCS-algorithm to determine matches. Unmatched nodes after

determining the LCS on each level are examined for crossmatches (moves). The algorithm not only facilitates the analysis of temporal evolving tree-structures but also the comparison of similar distinct trees. To support the FMSE implementation and expressive visualizations Treetank is enhanced in several ways. The following new operators/methods and components are available:

- `LevelOrderAxis` which incorporates attribute- and namespace-nodes if desired.
- `copy-operation` to copy nodes/subtrees of other *database/resource*-tuples.
- `move-operation` to move nodes/subtrees in the currently opened *resource*.
- `replace-operation` to replace a node and its subtree with another node/-subtree.
- Visitor pattern support for nodes/transactions.
- Merging or avoidance of adjacent text nodes.

The `LevelOrderAxis` and the other operations are described in detail in Appendix A.3 and A.4. Having described the preliminaries the next section describes the FMSE implementation itself.

FMSE The FMSE implementation first saves node-types and the according node-IDs in two maps during a postorder traversal. Next, the algorithm determines a longest common subsequence of matching nodes. Leaf nodes are compared first, then inner nodes. Thus the inorder-traversal described in [10] must be replaced by a postorder-traversal. Otherwise some leaf nodes are not processed beforehand which are required to determine the similarity of inner nodes. The matching of nodes involves two different similarity-metrics as described in the last chapter. However our implementation requires some explanation, as the matching is crucial and the changes applied by FMSE are propagated to a subsequent ID-based diff-algorithm:

- `TextNode`s are matched based on their String-value. The Levenshtein algorithm is used to compute a similarity measure of the values, which counts update costs of individual characters normalized between 0 and 1. `QName`s of `Namespace`- and `attribute`-nodes are matched first based on equality. In case of attributes afterwards their value is compared yet again using the Levenshtein algorithm in addition to their ancestor-elements.
- `ElementNode`s are compared based on the number of matched nodes in their subtree. Recapitulate that all node-types are chained for the fastMatching-algorithm bottom up during a postorder traversal. Empty elements however are compared based on their `QName` similarity, whereas all ancestor nodes are also compared once more using Levenshtein. This ensures the possibility of matching empty-elements after a deletion or insertion of a subtree. Treating empty nodes as leaf nodes otherwise will prohibit matching empty `element`-nodes with other `element`-nodes which include a subtree because leaf nodes and internal nodes are compared in different successive steps and thus not cross-compared. Matching nodes are stored in a `BiMap` containing forward and backward matchings of nodeKeys.

After storing matching node-IDs, FMSE step one is implemented straight forward. However whenever an `attribute`- or `namespace`-node is determined to be moved it is deleted from the old parent and inserted at the new parent node as moves of these node-types are not permitted by Treetank. Another noteworthy subject regarding moves is, that deleted text nodes in case adjacent nodes are collapsed and must be removed from the mapping as well. Due to adding the consistency constraint that *never*, before and after a commit, duplicate attributes with the same `QName` are permitted, a new attribute value is set in the `WriteTransaction.insertAttribute(QName, String)` method instead of adding a new one if the `QName` of the node to insert is identical to another attribute-node with the same parent. This also saves from time overhead due to node-creation. This case occurs whenever the attributes with the same `QName` and parent node are not matched because of very different attribute-values or parent `QName`s. All updated or inserted nodes are added to the matches as described by Chawathe et al. in [10] to prevent them from deletion in the next step.

The second FMSE step, which deletes non matching nodes with their whole subtrees, involves a preorder traversal of the tree. Thus a new `VisitorDescendantAxis` which optionally expects a visitor instance is implemented [5] and detailed in Appendix A.5.

The following cases have to be distinguished. The node to move

- has no right- and no left-sibling
- has no right- and no left-sibling but the parent has a right-sibling (the parent must be removed from a stack which is used to save right-siblings for nodes which have a first child.)
- has a right- and a left-sibling
- has no right- but a left-sibling
- has a right- but no left-sibling

In case of a node has a right- and a left-sibling (Fig. 4) it has to be determined if the sibling nodes are `TextNode`s as these will be concatenated during the remove-operation. The transaction moves to the left `TextNode`. Thus the preorder traversal in the VisitorDescendantAxis continues without skipping any nodes. Otherwise if no adjacency text nodes are merged during the remove()-operation the transaction is moved to the right-sibling before the operation is finished. Thus, the transaction first has to be moved to the left sibling.

3.3 ID-based diffing

Once revisioned data is stored in Treetank the main task is to reveal and present structural differences of the tree-structures. Treetank supports collections in form of databases which include one or more resources. Due to stable unique node IDs

[5] Visitors are always preferable to other methods if algorithms depend on the specific node-types, due to runtime errors during downcasts or possibly long chains of `instanceof` checks

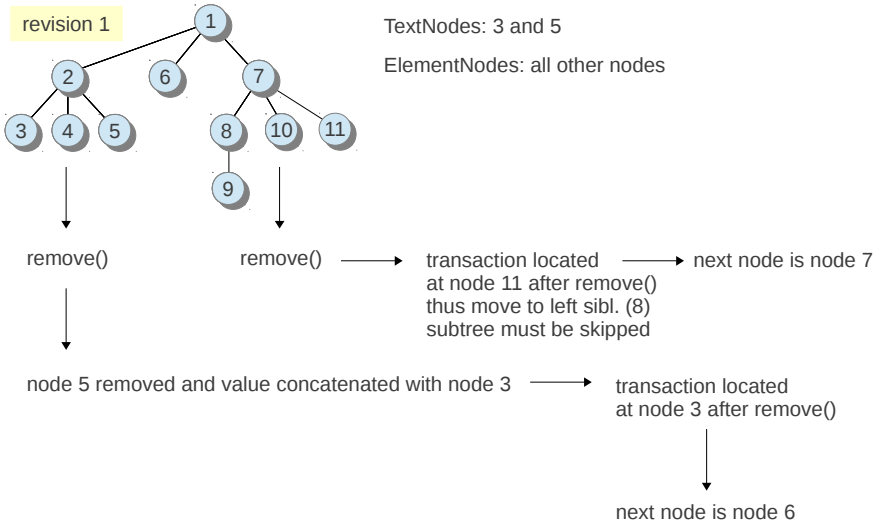

Fig. 4. Deletion visitor; two cases are depicted if the node to removed has a left- and a right-sibling.

in each resource every kind of tree-structure is imported updating the resource with the computed changes using FMSE. Even similar distinct tree-structures are imported updating the same resource. Otherwise, using separate resources it is not possible to utilize the unique node-IDs. Different resources do not share unique node-IDs.

A fast diff-algorithm utilizes these unqiue node-IDs and optionally hash-values which represent the content of the entire subtree rooted at a specific node. Note that the algorithm is designed to be able to compare any two revisions and thus not just consecutive revisions. It compares two nodes each time and determines the type of diff. Fig. 5 illustrates a simple example with edit-operations using Treetank.

Hashes One of our goals is the efficiency of our approach as it has to be usable within interactive visualizations. Meantioned briefly in the storage section hashes of the nodes are optionally used to skip subtree-traversals if the hashes of both nodes in each revision are identical. They are build incrementally based on the nodes in the subtree bottom up either during a postorder traversal during bulk inserts or on the fly depending on the concrete hash-algorithm used. The postorder-traversal to build hashes is a new bulk-insertion method to minimize minimize affected nodes and reduce the asymptotic bound from $O(n^2)$ to $O(n)$. Two kinds of hashes are available, rolling hashes and postorder hashes. *Rolling* hashes only affect the inserted or updated nodes on the ancestor axis whereas

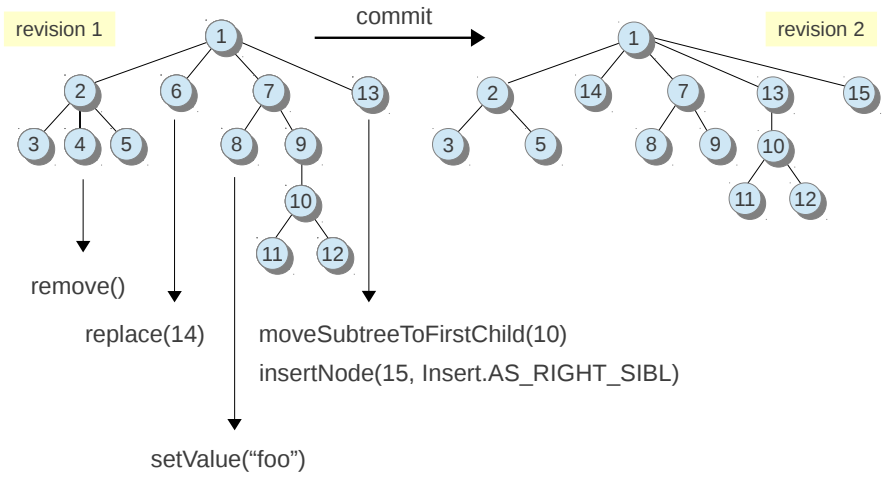

Fig. 5. Two revisions of a tree-structure and the edit-operations involved. By comparing consecutive revisions the edit-operations usually equal the diff-types considering `REPLACEDOLD/REPLACEDNEW` as one operation just like `MOVEDFROM/MOVEDTO`.

postorder hashes also affect nodes in a postorder traversal starting at the current node. Whenever identical hashes are determined the nodes are matched and the two transactions which compute the diff are moved to the next node in document order, which is not a descendant of the current node. Thus the transactions move to the first node in the XPath `following::`-axis. Hence, whole subtrees are skipped from traversal. The hashes include the unique node-IDs as well as node specific content. The hash-method is designed to be fast and to reduce collisions to a minimum. Even if hash-collisions which are extremely unlikely appear it is not possible to match subtrees with same hash-values as the node-IDs are also compared which are stable and unique during all revisions. Rolling-hashes are enabled by default during the database/resource creation and optionally used by our diff-algorithm. It is for instance used by an optional pruning of the tree in a Sunburst-layout to speed up the computation as well as the construction of the visualization. An in depth explanation of this application is provided in the Chapter 4. The next subsection briefly described two modes of the algorithm.

Kind of differences Interested observers are notified of the diff between two nodes through registration and the implementation of a special interface method. Currently two kinds of diffs can be computed.

- *Structural Diff* calculates changes without comparing attribute and namespace nodes. This implies that whenever the overall structure is crucial this algorithm should be chosen.

- *Full Diff* takes structural nodes as well as attribute and namespace nodes into account. However currently we do not emit non-structural changes. Changes in `namespace`- or `attribute`-nodes results in an `UPDATED` parent-element. This restriction applies as the *SunburstView* which is described in Chapter 4 currently does not include special `namespace`- or `attribute`-items. Instead these are part of the element item and shown on mouseover.

They are implemented by a simple template method `checkNode(IReadTransaction, IReadTransaction)` which is called by the basic diff-algorithm.

The following diff-types are supported by the algorithm:

- `INSERTED` denotes that a node is inserted.
- `DELETED` denotes that a node is removed.
- `UPDATED` denotes that a node is updated, that is either the QName of an `element`-node is updated or the value of a `text`-node.
- `SAME` denotes that a node is not changed.
- `REPLACEDOLD` denotes that a node or subtree is replaced (the old node/subtree).
- `REPLACEDNEW` denotes that a node or subtree is replaced (the new node/subtree).
- `SAMEHASH` denotes that a node is not changed and the hashes of the subtrees are identical.

Note that the differentiation between `REPLACEDOLD/REPLACEDNEW` is to support an expressive aggregated tree-structure as an underlying model of the visualizations. Two other diff-types are supported by an optional post-processing step.

- `MOVEDFROM` denotes that a node or whole subtree has been moved from this location to another one.
- `MOVEDTO` denotes that a node or whole subtree has been moved to this location.

The types are splitted, too to indicate the movemenet of the node, the old place as well as the new place in the aggregated tree-structure.

3.4 Traversal of both revisions

The algorithm to traverse the trees and to compute the differences between two nodes in each revision is depicted in algorithm 1. First, the method `treeDeletedOrInserted(IReadTransaction, IReadTransaction)` checks if both transactions opened on each revision can be moved to the start node. If not, either the node is inserted or deleted depending on the transaction which can not be moved.

Let's examine both cases:

- The transaction opened on the older revision can not be moved to the start node. This implies that the tree in the new revision has been inserted.
- The transaction opened on the newer revision can not be moved to the start node. This implies that all nodes in the old transaction have been deleted.

The distinction is used to support the selection of modified nodes in the visualizations which are described in Chapter 4 and only affects subtrees. Otherwise simply put all nodes in the old revision must be deleted, whereas all nodes in the new revision are inserted.

If the root-nodes of both revisions are selected by the transactions they move forward in *document order* (depicted in Fig. 6) depending on the last encountered kind of diff between two nodes. Document order is identical to a preorder traversal of a tree. In case of an insert, the transaction opened on the new revision is moved forward, in case of a delete the transaction opened on the old revision is moved (the
`moveCursor(IReadTransaction, ERevision)`-method). If a node is updated or has not been changed at all both transactions move to the next node in document order. Once the traversal in one of the two revisions is done, the transaction is located at the document root. The diff-calculation ends if either the transaction on the older revision is located at the document root and the last encountered diff kind was `DELETED` or both transactions are located at the document-root of both revisions. Note that if the transaction on the newer revision is located at the document root, but the transaction on the old revision is not the following nodes are `DELETED` at the end of the tree and have to be emitted as such (lines 22-28).

3.5 Diff-Computation

Besides moving both transactions forward in document-order depending on the type of diff the computation which determines the diff-type itself is crucial. The computation is the main task of the
`diff(INodeReadTransaction, INodeReadTransaction, Depth)`-method outlined in algorithm 2. It is invoked whenever the node-IDs or the `QNames`/`Text`-values of the nodes to compare differ.

First, the depths of the nodes have to be compared. The depth is the sum of nodes in the path up to the root-node. When the depth of the node in the old revision is greater than the depth of the node in the new revision it must have been deleted. Note that the depths are not persisted, thus counters have to keep track of the current depths. All diff-tuples which are of type `DiffType.DELETED` or `DiffType.INSERTED` are saved in a Java `List`. A second datastructure is used to gather the diff types itself, that is a whole subtree which is either inserted or deleted is cumulated by the according diff-type. This datastructure is used to find `INSERTED`, `DELETED` combinations which are instead emitted as of type `DiffType.REPLACEDOLD` (the deleted tuples) and `DiffType.REPLACENEW` (the inserted tuples).

Algorithm 1: ID-based diff: traversal

input : HashKind mHashKind, long pOldRevKey, long pNewRevKey, long mOldStartKey, long mNewStartKey, DiffType pDiffType, DiffTypeOptimized mDiffKind, ISession mSession
output: for each node comparsion: DiffType diffType, IStructNode oldNode, IStructNode newNode, Depth depth

```
1  INodeReadTransaction rtxOld ←
   mSession.beginNodeReadTransaction(pOldRevKey);
2  INodeReadTransaction rtxNew ←
   mSession.beginNodeReadTransaction(pNewRevKey);
3  // moveTo(long) returns true in case the transaction could be moved
      to the node or false otherwise.
4  newRtxMoved ← rtxNew.moveTo(mNewStartKey);
5  oldRtxMoved ← rtxOld.moveTo(mOldStartKey);
6  treeDeletedOrInserted(newRtxMoved, oldRtxMoved);
7  DiffType ← null;
8  // Check first node.
9  if mHashKind == HashKind.None OR mDiffKind == DiffTypeOptimized.NO
   then
10 |  diff ← diff(rtxNew, rtxOld, depth);
11 else
12 |  diff ← optimizedDiff(rtxNew, rtxOld, depth);
13 // Iterate over new revision (order of operators significant --
      regarding the OR).
14 if diff != DiffType.SAMEHASH then
15 |  while (rtxOld.getNode().getKind() != ENode.ROOT_KIND AND diff ==
   |  DiffType.DELETED) OR moveCursor(rtxNew, ERevision.NEW) do
16 |  |  if diff != DiffType.INSERTED then
17 |  |  |  moveCursor(rtxOld, ERevision.OLD);
18 |  |  if rtxNew.getNode().getKind() != ENode.ROOT_KIND or
   |  |  rtxOld.getNode().getKind() != ENode.ROOT_KIND then
19 |  |  |  if mHashKind == HashKind.None OR mDiffKind ==
   |  |  |  DiffTypeOptimized.NO then
20 |  |  |  |  diff ← diff(rtxNew, rtxOld, depth);
21 |  |  |  else
22 |  |  |  |  diff ← optimizedDiff(rtxNew, rtxOld, depth);
23 |  // Nodes deleted in old rev at the end of the tree.
24 |  if rtxOld.getNode().getKind() != ENode.ROOT_KIND then
25 |  |  emitOldNodes(rtxNew, rtxOld, depth);
26 done();
```

Algorithm 2: ID-based diff: diff-computation

input : Depth pDepth, INodeReadTrx pOldRtx, INodeReadTrx pNewRtx
output: kind of diff (DiffType enum value)

```
1  DiffType diff ← DiffType.SAME;
2  // Check if node has been deleted.
3  if pDepth.getOldDepth() > pDepth.getNewDepth()) then
4      diff ← DiffType.DELETED;
5      cumulatDiffTypes(diff);
6      if checkReplace(pNewRtx, pOldRtx) then
7          diff ← DiffType.REPLACED;
8  // Check if node has been updated.
9  else if checkUpdate(pNewRtx, pOldRtx) then
10     diff ← DiffType.UPDATED;
11 // Check if node has been replaced.
12 else if checkReplace(pNewRtx, pOldRtx) then
13     diff ← DiffType.REPLACED;
14 else
15     long oldKey ← pOldRtx.getNode().getNodeKey();
16     boolean movedOld ← pOldRtx.moveTo(pNewRtx.getNode().getNodeKey());
17     pOldRtx.moveTo(oldKey);
18     long newKey ← pNewRtx.getNode().getNodeKey();
19     boolean movedNew ←
       pNewRtx.moveTo(pOldRtx.getNode().getNodeKey());
20     pNewRtx.moveTo(newKey);
21     if !movedOld then
22         diff ← DiffType.INSERTED;
23     else if !movedNew then
24         diff ← DiffType.DELETED;
25     else
26         // Determine if one of the right sibling matches.
27         EFoundEqualNode found ← EFoundEqualNode.FALSE;
28         long key ← pOldRtx.getNode().getNodeKey();
29         while pOldRtx.getStructuralNode().hasRightSibling() AND
           pOldRtx.moveToRightSibling() AND found ==
           EFoundEqualNode.FALSE do
30             if checkNodes(pNewRtx, pOldRtx) then
31                 found ← EFoundEqualNode.TRUE;
32                 break;
33         pOldRtx.moveTo(key);
34         diff ← found.kindOfDiff();
35     cumulatDiffTypes(diff);
36 return diff;
```

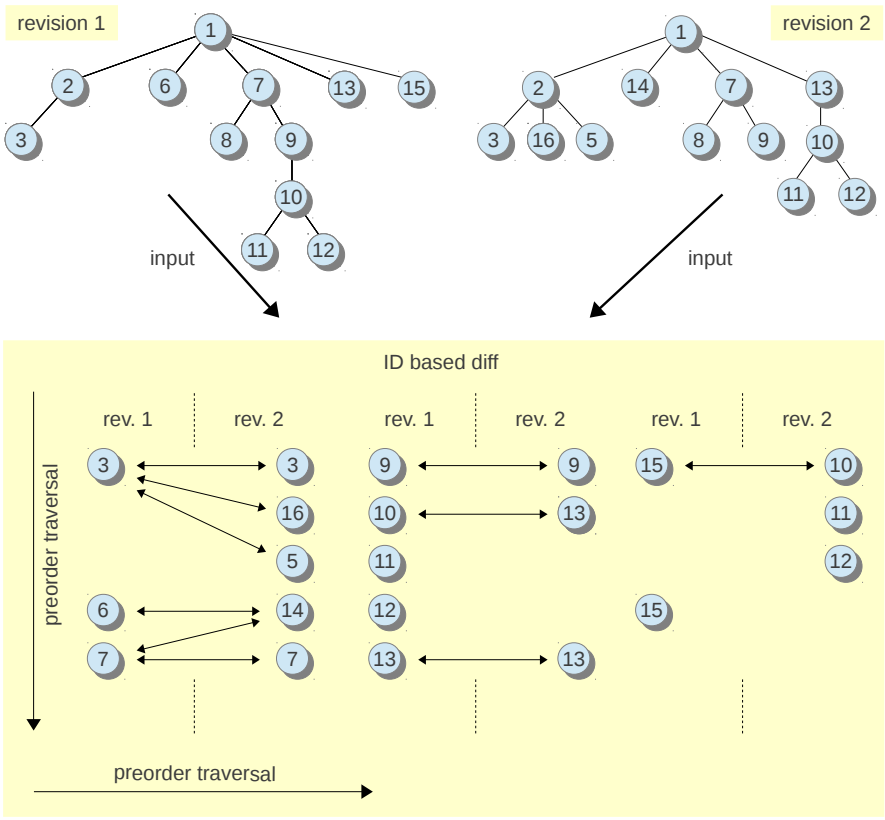

Fig. 6. ID-based diffing.

When the depth of the node in the old- and the node in the new-revision instead either is identical or the depth of the node in the more recent revision is greater at first the node is checked for an update through comparison of the nodeKeys and the depths of the nodes. Unless the check yields true the nodes are examined for replacement. Therefore the datastructure which keeps track of deleted- and inserted-subtrees is reviewed. Consecutive insert/delete- or insert/insert/delete/delete-tuples are emitted as replaced subtrees. Note that this replace-detection is just a simple heuristic and currently does only detect the aforemeantioned pairs of inserted and deleted nodes.

Assuming a node or subtree has not been replaced, it must be decided if the current node in the new revision is inserted or the current node in the old revision is deleted. First, it is determined if the transaction opened on the old revision can be moved to the current node in the new revision. If not it is immediately obvious that the node has been inserted. Otherwise, if the transaction opended on the new revision can *not* be moved to the current node in the old revision it must

be a deleted node. For the simple reason that move-operations are supported both checks might succeed. In this case the right siblings of the node in the old revision have to be examined in order to determine the type of diff until one of them matches the node in the new revision, that is the node-IDs are identical or no more right siblings are available. In the first case the new node must be inserted. Otherwise it must be deleted.

3.6 Detecting moves

An optional postprocessing step is required to detect moves. The two basic diff-types `MOVEDFROM` and `MOVEDTO` are detected after all operations have been emitted. The detection requires three datastructures to store all diff-types, the inserted nodes and the deleted nodes.

Impossible to detect moves on the fly Note that it is not possible to include the detection of moves in the preorder-traversal of both revisions itself, as it is not known which of the two nodes is the one which has been moved and which one is the node which is unchanged. However this is required to determine which of the two transactions must be moved to the next node in document order. All we can argue is, that it would be possible to detect a move itself if the transaction on the new revision can be moved to the current node from the transaction in the old revision and vice versa. That is all `DELETED` or `INSERTED` nodes have already been emitted, meaning that one of the two nodes which are currently compared must have been moved and the other must have been unchanged constant. It is not possible to decide which one of the two nodes stayed the same and which one has been moved. The position in the tree is no implication wheter a node has been moved from or moved to another place, but this is crucial to decide which of the two nodes has not changed and which one actually has been moved. As thus the types have to be matched. Whenever the unique nodeKey of a `DELETED`-node matches the key of an `INSERTED`-node, the corresponding diff-types can be changed into `MOVEDFROM` and `MOVEDTO`. The next subsection details a postprocessing algorithm which is based on this idea.

Detection of moves in a postprocessing step All encountered diff-types are saved in an associative array, a map (index of their encounter $\Leftrightarrow$ diff-tuple) [6] in preorder which is the order in which they have been observed. Additionally `DELETED` and `INSERTED` nodes are recorded mapping their unique nodeKey to the index in the original map with all entries. The algorithm described in 3 expects the three maps. It tries to match `INSERTED` $\Leftrightarrow$ `DELETED` pairs and vice versa and checks whether the diffType in the map needs to be adjusted to `MOVEDTO` or `MOVEDFROM` (or not at all). A map which does not contain a value for the

[6] which is used just like a List, to switch between a map implementation based on a persistent BerkeleyDB database depending on a specified threshold value or a simple LinkedHashMap instance

specified key returns the special value `null`. First, the old nodeKey (might have been deleted) is searched for in the Map containing all inserted tuples. If the key is found (value ! = null) the type is checked. If the current diff-tuple is of type `DELETED` or `MOVEDFROM` the diff type is set to `MOVEDTO`. Note that the check for the `MOVEDFROM` type is necessary as the corresponding `INSERT` tuple might have been encountered before and thus the type has been changed to `MOVEDFROM` already. The following is the inverse case to set the `MOVEDFROM` type if necessary. Furthermore a link in the form of the index of the matching node with the same nodeKey and the corresponding `MOVEDTO` diff-type is additionaly saved. Note that the algorithm does not detect `text`-nodes which are moved to a right sibling of another `text`-node. In this case our implementation of the `moveSubtreeToRightSibling(long)` prepends the value of the current `text`-node to the moved `text`-node and subsequently deletes the current node. This ensures, that no two text-nodes are ever adjacent which otherwise contradicts the XQuery/XPath data model (XDM). In this particular edge case the algorithm determines a `REPLACED` node (a direct consequent from the deletion of the node the transaction resides at and the insertion of the moved node with the prepended text-value) and a `DELETED` node (the node which has been moved).

Algorithm 3: ID-based diff: postprocessing to detect moves

input : Map allDiffs, Map inserted, Map deleted
output: none (void)

```
1 // For every diff tuple in the map which saves all encountered diffs
     in document-order
2 for Diff diffTuple : allDiffs.values() do
3     Integer newIndex ← inserted.get(diffTuple.getOldNodeKey());
4     if newIndex != null AND (diffTuple.getDiff() == DiffType.DELETED OR
      diffTuple.getDiff() == DiffType.MOVEDFROM) then
5         allDiffs.get(newIndex).setDiff(DiffType.MOVEDTO);
6     Integer oldIndex ← deleted.get(diffCont.getNewNodeKey());
7     if oldIndex != null AND (diffTuple.getDiff() == DiffType.INSERTED OR
      diffTuple.getDiff() == DiffType.MOVEDPASTE) then
8         allDiffs.get(oldIndex).setDiff(DiffType.MOVEDFROM).setIndex(
          inserted.get(diffTuple.getNewNodeKey()));
```

3.7 Runtime/Space analysis and scalability of the ID-based diffing algorithm

The runtime of the ID-based diff algorithm is $O(n+m)$ whereas n is the number of nodes in the first tree and m the number of nodes in the second tree. It is thus a fast linear diff-computation in comparison to the FMSE-algorithm which is used in the first place to determine the differences which are imported. The

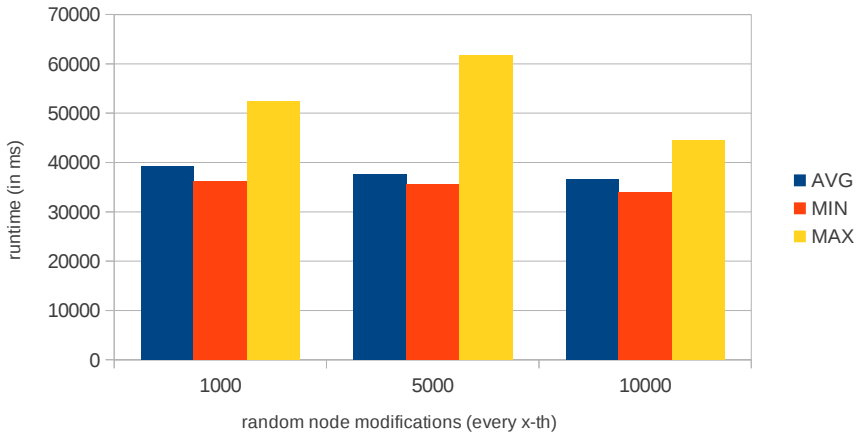

Fig. 7. Scaling during different modification-schemas (update/insert/delete/replace/move every 1000st, 5000st and 10000st node) in a 111 MiB XMark instance.

space-consumption is $O(1)$ and in case the replace-operation is enabled at worst $O(k)$ whereas k is the sum of the subtrees which are cached (at most 4 subtrees currently). However the space consumption of an aggregated tree-structure which is going to be described in the next chapter (4) is $O(u*k+v*k)$. u is the number of unchanged nodes, v is the number of changed nodes and k is tuple relevant stuff (type of diff, the nodes and the depths in both trees). Thus the asymptotic space consumption is linear depending only on the number of unchanged and changed nodes.

Our performance evaluation involves measuring the scaling during different modification-loads (Fig. 7) and different document sizes with scaled modification-loads (Fig. 8). The modification-loads are increased in the same scale as the document size such that the documents are modified with approximately the same number of modifications. The hardware used is a common notebook with 4Gb RAM and a Core 2 Duo 2,66 Ghz CPU. Fig. 7 shows that the number of modifications minimally affects the runtime. The runtime decreases linear as less modifications between two revisions have been made. However the linear decrease is minimal due to a few more checks.

Fig. 8 depicts the scaling during different document sizes and modification numbers. The scale is logarithmic, thus we are able to identify the linear runtime due to increased document sizes. It therefore emphasizes the asymptotic bound.

	1000mods	**5000mods**	**10000mods**
min	36265.14	35717.34	33948.96
max	52459.38	61860.29	44451.06
average	39167.75	37650.67	36579.37

Table 3. Comparsion of different modification-schemas of a 111 MiB XMark instance (update/insert/delete/replace/move every 1000st, 5000st and 10000st node). Runtime in ms.

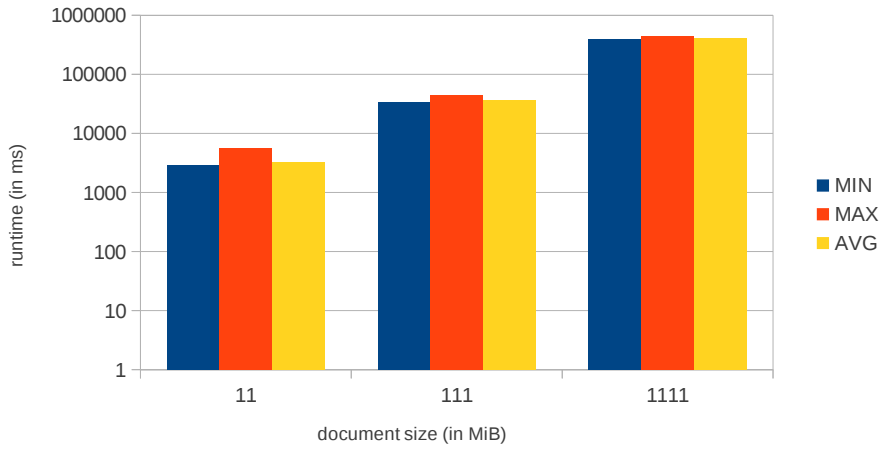

Fig. 8. Different document sizes with modification-count scaled accordingly (11 MiB ⇔ modifiy every 1000th node, 111 MiB ⇔ modify every 11000 th node, 1111 MiB ⇔ modify every 122221th node / Y-axis logarithmic scaled)

plot the pruned diffs and mention the tables

3.8 Conclusion and Summary

This chapter first introduced Treetank, the DBMS tailored to revisioned tree-structures. We motivated the import of differences between full dumps of temporal tree-structures to subsequently take full advantage of Treetanks' (1) revisioning strategies, (2) unique node-IDs which identify a node through all revisions and (3) hashes of each node, which almost guarantee a unique hash guarding the whole subtree through MD5 rolling hashes. The FMSE-algorithm has been implemented to support the initial import of differences in these full dumps, that is revisioned data (currently) in the form of XML-documents whereas each document represents one revision, a snapshot at a specific time. Considering no such data is available other even more sophisticated preprocessing steps have to be implemented and executed before a diff-algorithm on the revisioned data

	11 MiB	**111 MiB**	**1111 MiB**
min	2957.79	33948.96	401878.64
max	5694.65	44451.06	439337.71
average	3323.96	36579.37	413158.13

Table 4. Comparsion of different XMark instances (11 MiB, 111 MiB, 1111 MiB modifying every 1000st, 11000st and 122221st node). Runtime in ms.

	11 MiB	**111 MiB**	**1111 MiB**
min	1239.01	14208.30	197223.12
max	3698.94	19135.01	271794.82
average	1408.74	14596.86	210296.70

Table 5. Comparsion of different XMark instances skipping subtrees of nodes with identical hash-values (11 MiB, 111 MiB, 1111 MiB modifying every 1000st, 11000st and 111000st node). Runtime in ms.

in Treetank are usable. Two of the use cases which are going to be discussed in Chapter 5 require further preprocessing. Note that the FMSE-algorithm is used to import data which does not assume unique node-IDs and matches nodes based on a longest common subsequence (LCSS)-calculation (which is ambiguous) for leaf nodes and inner nodes in a bottom up traversal. Thus it might mismatch similar nodes eventuating in too many edit-operations. The algorithm is particularly useful to compare similar different tree-structures which do not temporally evolve.

Furthermore we have implemented many edit-operations which were not available in Treetank to support the implementation of the FMSE-algorithm and a very expressive agglomerated tree-structure. We have shown that a subsequent diff-calculation based on IDs and hashes which guard the whole subtree is faster than the same algorithm without utilizing hashes. However it depends on the properties of the tree-structures. If large subtrees can be skipped due to same hashes meaning they are unchanged the running time is reduced drastically.

Besides using hashes optionally to skip whole unchanged subtrees our ID-based diff-algorithm combines `INSERT/DELETE` and `DELETE/INSERT` sequences to a single replace-operation. To support an agglomerated tree-structure which incorporates nodes of both compared revisions described in the next chapter either emits diff-tuples of type `REPLACEDOLD` or `REPLACEDNEW` depending on which of the two subtrees are larger, thus increasing the expressiveness. Note that this detection potentially involves a lot of state if large subtrees are inserted/deleted or vice versa successively. Therefore we additionally provide a different replace-detection which uses heuristics checking right-sibling nodeKeys for equivalence or the parent node keys if in both revisions no right sibling exists).

Moves are optionally detected in a postprocessing-step by searching for `INSERT/DELETE` and `DELETE/INSERT` combinations with the same node-IDs. Move

detection is especially useful to support analysts with an expressive visualization. Otherwise in document-centric XML, for instance DocBook[23] documents, it might be impossible to draw conclusions from simple inserts/deletes, whereas an author simply moved a sentence along with other inserts/deletions of text and/or markup.

4 Visualizations

4.1 Introduction

The last chapter introduced the first part of the pipeline, the diff-algorithms in detail. Usually however sophisticated preprocessing-methods are needed, which are explained for a few applications in chapter 5.

This chapter describes several visualizations which help analysts to gain fast knowledge. First the GUI-framework which embeds various visualizations is briefly described. Next, an aggregation of the two tree-structures to compare follows. the visualizations themselves are detailed. Our visualizations rely on the diff-algorithms described in Chapter 3 and therefore depict the tree-edit distance, that is structural (insert/delete/move/replace) and non-structural (update) operations which transform one tree-structure into the other one. Different similarity measures are used to indicate the similarity of leaf-nodes and internal nodes either based on values or in the latter case based on overlapping subtree-structures. However the usage of similarity measures is highly modular and can be extended with further measures which either can be switched by user interaction or through heuristics. After briefly describing the *TreeView* and the *TextView* as well as basics of our specialized Sunburst layout, an explanation of filtering techniques follows which together with the ID-based diff-algorithm [7] facilitates the analysis of even large tree-structures ranging from about 100MB to even GBs of data. The key assumption underlying this efficient diff-algorithm/visualization is that similar trees are compared and therefore only a small fraction of a tree-structure has to be transformed to derive the other tree-sturcture. Querying capabilities, similarity measures and the the visualization of moves are described subsequently. Next, Smallmultiple variants are detailed which facilitate the comparison of several tree-structures. The chapter concludes with a summary.

4.2 GUI

First, a GUI framework has been developed which incorporates several views. The framework has been written from scratch based on some key-ideas and software patterns used by BaseX [24]. The GUI is designed to be easily extendable. It currently offers the ability to view and interact with the stored Treetank data in many ways. Incorporated are several different views. Most of them are developed to support the analysis of differences as well as similarities between tree-structures. Others will be extended in the future. Furthermore the views are synchronized meaning that several types of actions in one view are reflected in other views as well. Special care is taken to adhere to the Model View Controller (MVC) architecture with a controller managing the interactions between the views which is depicted in Fig. 9. The next section describes the visualizations in detail.

[7] usually the hash-based version comparing the hash-values of the nodes first

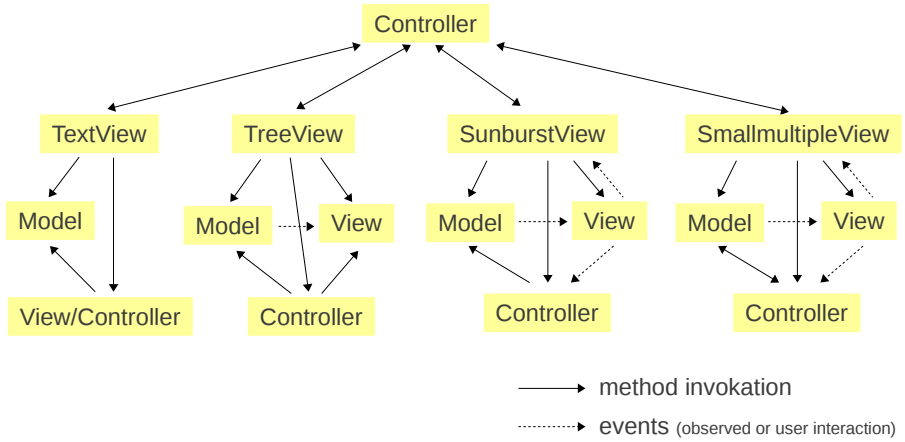

Fig. 9. MVC-paradigm use. The *TextView* and the *TreeView* use standard Swing components. A `JTree` Swing-component is used implement the tree-model in the *TreeView*. A `JTreeCellRenderer` implements the view and controller. It is responsible to translate user actions and to render the cells appropriately. The `JTextEditPane`-Swing component represents both the view and controller in the *TextView* whereas a new `StAXDiffSerializer` described later on implements the `XMLEventReader` StAX interface to support a pull based API. It is thus the model which interacts with Treetank through a open database handle.

4.3 Aggregation

An aggretion of two tree-structures is illustrated in Fig. 10. The top half depicts the two tree-structures (revision 1 and revision 2) to compare whereas the bottom displays the aggregation or fusion of the trees based on diff-tuples encountered by the internal ID-based diff-algorithm. The two trees are input to the ID-based diff-algorithm which in turn fires diff-Tuples. These tuples form the basis of the agglomerated tree-structure. A straight forward approach which we followed is to store the tuples in a simple List datastructure [8]. The colors of the nodes in the agglomeration denote if and what change is made. Deletions for instance are marked in red, whereas insertions are colored blue. All update-operations of the ID-based diff-algorithm in Chapter 3 are supported. Updates are not only supported for leaf nodes, as in the ContrastTreemap approach described in Chapter 2 but also for internal nodes. Furthermore the replace-operation as well as moves are supported. Move operations are plotted via curves using hierarchical edge bundles which are drawn on top of the Sunburst layout, whereas an item indicating the position in the old tree-structure (`DiffType.MOVEDFROM`) and another item denoting the position in the other tree-structure (`DiffType.MOVEDTO`)

[8] in our case a Map which is used like a List to exchange a Java core collection map implementation with a persistent BerkeleyDB map implementation

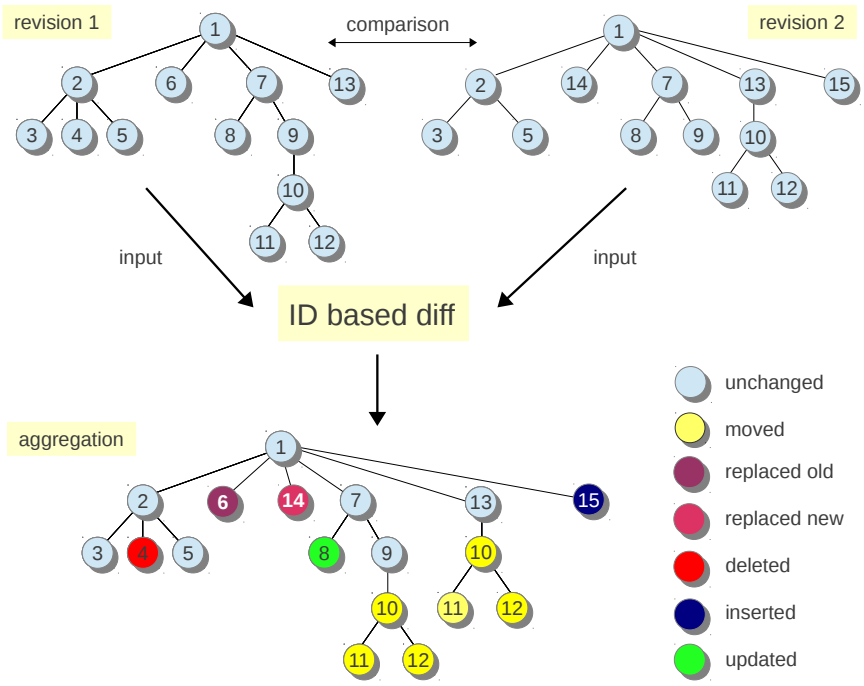

Fig. 10. Two tree-structures aggregated. The numbers denote unique node-IDs and refer to Fig. 5 in Chapter 3 just like the changes from revision 1 to 2. Both revisions are input to the ID-based diff-algorithm. The output represents diff-tuples including the node-IDs from both nodes which are compared in each step, the type of diff and the depths of both nodes. Storing the observed diff-tuples in an ordered data-structure forms a tree-aggregation.

is drawn. Items which represent updated nodes include both the value from the first tree and the value from the second tree. Items which constitute replaced nodes include the value from the replaced node as well as the new node. Note that this operation is useful if comparing temporal tree-structures which change over time. The aggregation is automatically achieved through the usage of the ID-based internal diff-algorithm. The model is registered as an observer and the changes are added to a List which is a parameter of a special diff-axis to create the Sunburst Items for the new comparsion layout.

4.4 Visualizations

Visualizations are a major contribution in this thesis. As described in the motivation humans are best in interpreting visual content. Therefore visualizations have

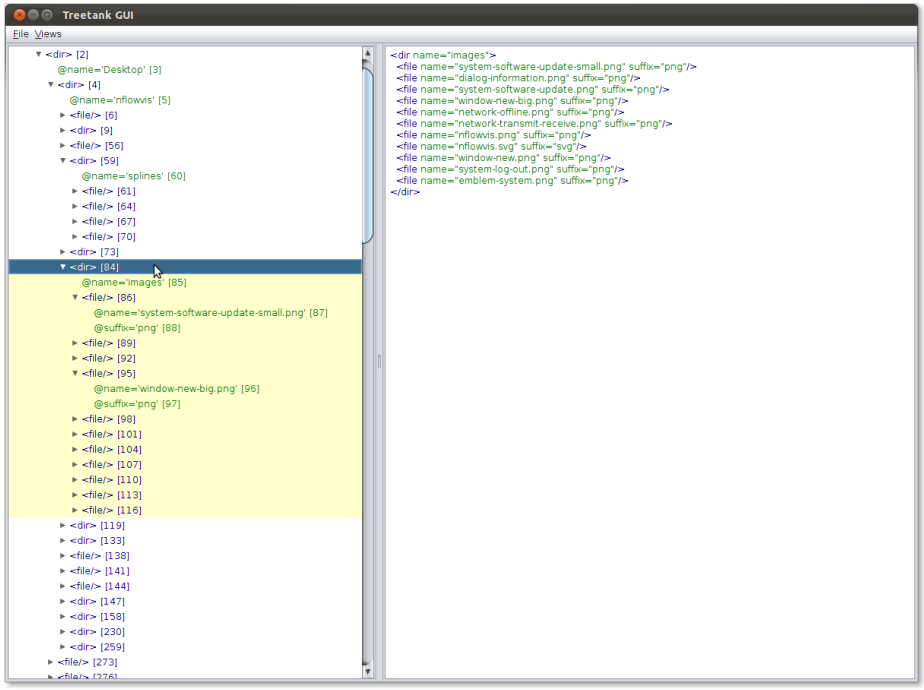

Fig. 11. TreeView and TextView side-by-side

been developed which facilitate humans in gaining new insights and quickly detecting differences in tree-structured data. Next, all available views are described in detail.

- First, a *TreeView* displays nodes in a tree structure just like visual frontends for filesystems as illustrated on the left side in Fig. 11. Nodes can be expanded to show all child nodes which are inside the current viewport or collapsed to hide children. Therefore a Java-Swing `JTree` is used which has been extended to mark the subtree of a selected node with a background color. The `TreeCellRenderer` renders nodes according to their node type and a `TreeModel` interacts with the storage. Currently the view is not able to incorporate the differences encountered via the diff-algorithm and therefore not further described.
- The *TextView* displays serialized XML-documents or fragments and supports syntax highlighting. Moreover it just serializes the part of data which is currently viewable and an additional small overhead of pixels to enable scrolling. Other data is serialized and appended to the text pane once a user scrolls down. A new *StAX*-parser implementation provides a pull-based API to support this "append-on-scroll" behaviour. It provides two features. Since it is pull based the application (the GUI) can determine when and how much data is parsed. Furthermore it provides the parsed node kinds which are used

to support syntax highlighting. Simply using a `DescendantAxis` from Treetank which traverses the nodes in preorder would not be sufficient, end-tags have to be emitted as well. Therefore it is much easier to develop a reusable `StAX`-implementation. Nontheless the `DescendantAxis` is used internally as part of the implementation to traverse the tree in preorder. The algorithm which implements a StAX-Parser for Treetank is outlined in appendix **??**. To adhere to the specifications of the methods which must be implemented and to keep the iterator-methods idempotent might have been the biggest challenge. In addition to generate events for end-tags the StAX-parser supports a `peek()`-method.
Figure 11 displays the *TreeView* and the *TextView* side by side. Note that the two views are kept in synchronization.
In order to support an analyst with the task of analysing differences in tree-structured the view supports another mode which incorporates the aggregation of the two tree-structures to compare described in section 4.3. Based on this aggregation another StAX-Parser called *StAXDiffSerializer* has been developed which receives a `DiffAxis` to iterate over `SunburstItems` created for the *SunburstView* which is described next. We support a preorder-traversal of the items to derive the stored diff-types as well as the depths in the items without the distortion of a semantic view for the *SunburstView* described later on in this chapter. The depth of the current item and the depth of the next item using a *peek()* method on the `DiffAxis` are used to determine if an end-tag must be emitted. Another check involves empty-Elements whereas an `EndElement` must be emitted immediately following the `StartElement` in the case of the next call to `nextEvent()`. In this case the parent node-IDs must match. Furthermore the depths are used in subsequent calls to *nextEvent()* to determine how many closing tags must be emitted. In case of `ElementNode` updates we immediately emit the new updated element and push the old element on an end-tag stack. In a subsequent call to `nextEvent()` the old value in case of a `TextNode` or the old element in case of an `ElementNode` are emitted. Changed nodes are highlighted with a background color which denotes the kind of change. The `TextView` itself must not change the indentation for the old value/old element if an UPDATE has been detected. Only the first emitted value (the new value) must change the indentation.
A legend which describes the color $\leftrightarrow$ change mapping is currently only available from within the *SunburstView* which is described next. However this is only an implementation detail and a help-dialog can be added easily. Exemplary a side-by-side view with the *SunburstView* is depicted in Fig. 12 whereas the first inserted subtree is also visible in the *TextView* area. Note that while the SunburstView provides a great overview about the whole tree-structure and subtrees, the *TextView* provides a better detailed view on selected parts of the whole tree-structure. Other deficiencies meantioned in the introduction regarding the boundary of nodes and XML-specific details do not apply as we compare the tree-structure, instead of a character based comparsion, with the ID-based diff-algorithm in the first place. Be-

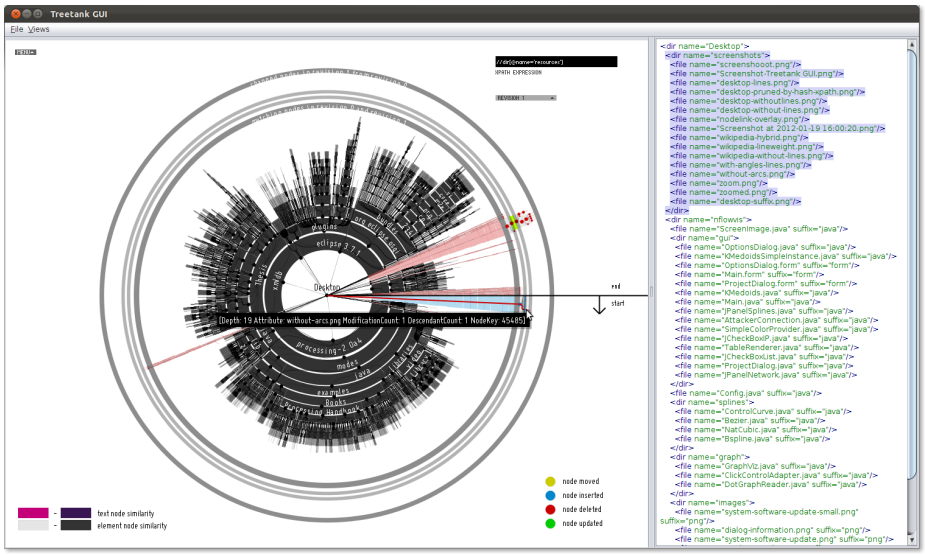

Fig. 12. SunburstView and TextView side-by-side

sides the lack of an appropriate overview, which is one of the advantages of the *SunburstView*, the *TextView* is an ideal partner to the *SunburstView* as the XML text-serialization is better readable than radial Sunburst labels and readers might be more familiar with pure XML.

Note that the diff-algorithm is only ever called once for every visible view (besides a smallmultiple variant which represents changes among several tree-structures or revisions in Treetank), whereas a simple iterator on the created *SunburstItems* is broadcasted to all other views which are currently visible to support the iteration over the agglomerated tree-structure.

- The *SunburstView* displays the tree structure in a radial arrangement (Fig. 13). It is a space filling approach and thus tries to maximize the usage of available screen space for the hierarichical visualization. Furthermore it is adjacency based, drawing child nodes next to their parent node. In contrast, Treemaps enclose child-nodes within parent nodes. Thus a Treemap utilizes available screen space to the full extend as the root node uses all available space recursively embedding descendants as rectangles. In contrast, in the Sunburst method corners are left empty due to the radial representation. While this alone on first glance might be a great drawback in addition to circular segments which are more difficult to read regarding node labels and comparisons of item-sizes, the layout is stable even with a lot of changes and the hierarchical structure is much better readable. The Spiral Treemap layout which has been described in chapter 2 is relatively stable but it is still very difficult to track changes which might be scattered through 90 degree changes in direction as well as the complicacy to follow nested rectangles

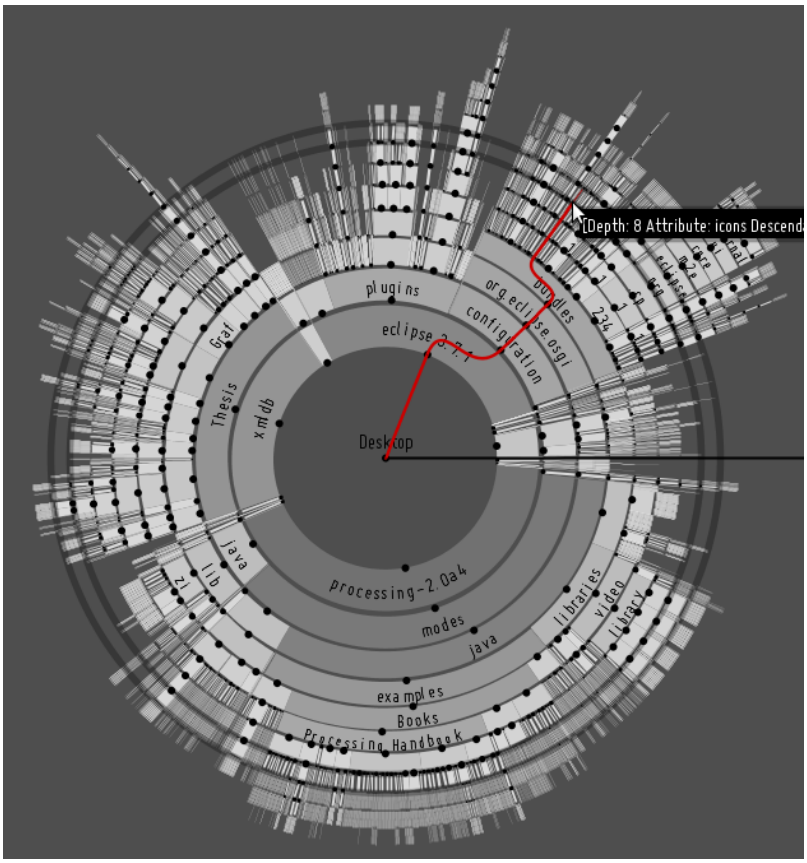

Fig. 13. SunburstView

which are arranged in spirals in comparison to the simplicity of a Sunburst layout.

The root node of a tree-structure in a Sunburst-layout is plotted in the middle of the screen depicted as a circle. Child-nodes of the root node are drawn in circular segments next to their parent. In our case the radius depends on the depth and shrinks with increasing depth such that the area of circular-segments between two levels does not change. Otherwise items toward the edge will occupy more space. However this behaviour can be changed to further visually emphasize changes which will be visualized along the edge (section 4.5). The extend of an item in the Sunburst layout which depicts one node depends on one or more attributes of the node whereas a relative measure in regard to the other children is used.

The subtree-sizes are mapped onto the extend of each segment such that nodes having more descendants get a greater arc and thus occupy more space. The formular is straight forward:

$$ext = \begin{cases} 2 \cdot \pi & \text{if } node\ is\ root\ node \\ parExt \cdot descs/(parDescs - 1) & \text{otherwise} \end{cases} \quad (3)$$

Note that we recently added the number of descendants of each structural node in Treetank itself to maximally speed up the creation of the visualization. Before, while creating the items we issued a preorder traversal on each node in parallel using a Java `ExecutorService` and a `BlockingQueue` which is also used in the comparsion view described later.
The color of each segment in case of internal nodes (element nodes) is mapped to the number of descendants of a node plus one as well. Having that said in future versions it will be possible to map another custom attribute available through a drop-down menu to the color. If such an attribute is not available for every node a default value can be assumed. Leaf nodes which are `text-nodes` are colored according to their text-length.
A node-link diagram is drawn on top of the *SunburstView* to further emphasize the hierarchical structure. Dots representing the node in addition to the SunburstItem-segments are depicted in the middle of the segment whereas either bezier curves or straight lines denote a child/parent-relationship between the nodes.
In order to support large tree-structures the generated Sunburst-segments are drawn into an offscreen buffer, whereas the actual items are only used to implement a mouseover effect and to enable XPath-queries and highlighting of result-nodes.

Interaction The view is highly customizable. Through checkboxes the dots/circles denoting a node in the node-link overlay and/or the arcs can be hidden. Furthermore the radius of the arcs is adjustable through sliders. Moreover the radius of the dots is manageable through a slider, whereas the connections between child/parent-nodes in the node-link overlay either can be disabled and a range denoting the line/curve thickness is adaptable. By default links at a higher level are drawn thicker depending on the range-value. On top of that it is possible to adjust certain values such that either the whole node-link diagram or the *SunburstView* disappears. More options are available. Some adjustments are depicted in Fig. 14.
Figure 15 illustrates the node-link overlay without coloring the arcs. The red curve marks a path up to the root-node through all ancestor-nodes for the current node which is highlighted by moving the mouse over the item.
Besides, mapped values to the color of Sunburst-segments or items, a term which is used interchangeably in the following sections, can be normalized according to a linear, squareroot or logarithmic scale.
Crucial to the interaction and the value of the visualization itself is the possibility to drill down into the tree. Clicking an item results in drawing the selected node with its subtree in a new Sunburst-diagram whereas the selected node is simply used as the new root-node. Stacks are used to implement a simple undo-operation which is very fast as we store the Sunburst-items as well as the background buffer-image.

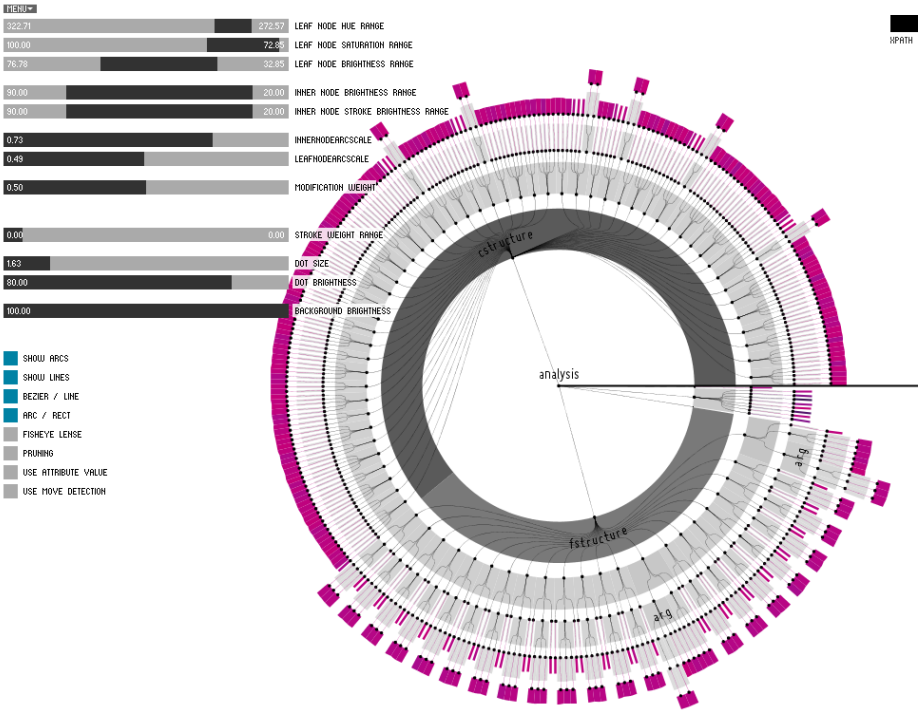

Fig. 14. SunburstView - adjusted arcs/dotsize parameters

A well known technique to enlarge small regions is to use transformations of the screen-space, as for instance a fisheye lense. The enlargement of small items via a fisheye lense is depicted in Fig. 16. Zooming and panning is also incorporated allowing affine transformations of the screen (Fig. 17) to analyse important regions. Note that the mouseover-effect displaying additional information about the node itself as well as the legends are not affected by the transformation. As the background-buffer cannot be used in this case zooming is restricted to smaller trees with an upper bound of about 10_000 to 15_000 nodes.
In order to manipulate Treetank resources it is even possible to insert XML fragments as right-siblings or first-childs as well as to delete nodes.

Querying XPath can be used to query the tree-structure for specific nodes. Result sequences are highlighted in a light green. Figure 18 displays the result of a simple `//*[text()='var:0']` query to highlight all nodes which have a text-node child with the value "var:0".

Labels If the Sunburst items are huge enough and the scale to draw the arcs for each depth is greater than a predefined value (currently 0.7, whereas

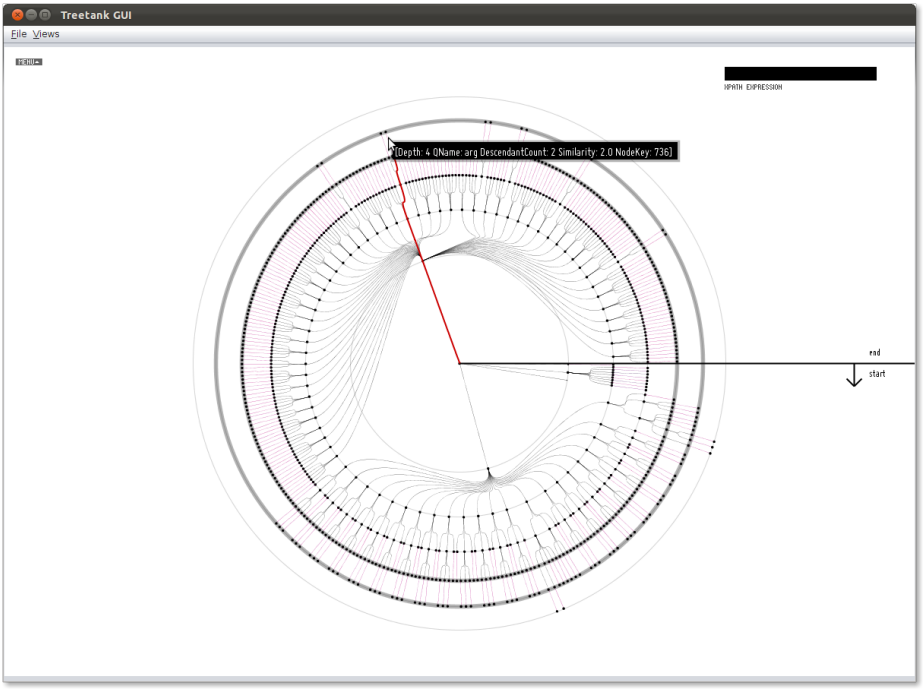

Fig. 15. node-link diagram

the scale ranges from 0 to 1) labels are drawn. Labels in the top half of the visualization, that is if the center of the item is greater than π, are drawn beginning at the startAngle in clockwise direction, otherwise they are drawn starting from the endAngle in counter clockwise direction. Furthermore the font-size ranges between two values, whereas the size is decreased with an increasing level.

Filtering/Pruning The standard *SunburstView* incorporates a method to filter the tree by level. While this filtering is not perfect in circumstances where the fanout is very large, it turned out to work very well to keep the number of generated Sunburst items small. Furthermore the view currently is used as an entry point to the comparsion view which is also based on the Sunburst-layout whereas it is planned to backport the filtering by itemsize. In the following we use the terms *filtering* and *pruning* interchangeably. Whereas it will be sufficient to use an XPath-query as for instance `//*[count(ancestor::*)<3]` to get a sequence including all nodes between level 0 and 3 we opted for a tree-traversal implementation, as the XPath query has to touch all ancestor nodes in the current Treetank implementation which doesn't incorporate hierarchical node-IDs as for instance the ORDPATH/Dewey-IDs where it's usually trivial to compute such queries

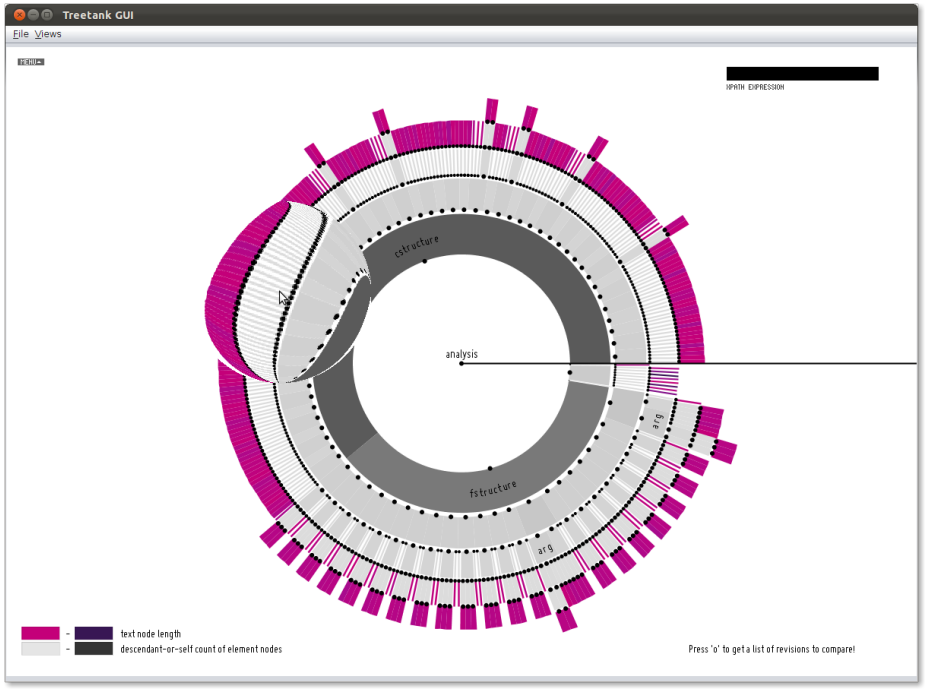

Fig. 16. Fisheye transformation

on the ID itself in an in-memory B*-tree or something alike. To support such queries efficiently it is scheduled to include the depth for each inserted node, which usually doesn't have to be adjusted in case of all operations but the movement of nodes/subtrees. In that case all moved nodes have to be adjusted instead of just the root node and it's former siblings/parent and the new siblings/parent. Furthermore the `DescendantAxis` and/or `VisitorDescendantAxis` can be adjusted to optionally use a maximum depth for the traversal.

4.5 Comparsion using a new Sunburst-layout algorithm

The standard *SunburstView* includes a comparison-mode. Once a base revision is opened and the *SunburtView* is enabled an analyst is able to choose another revision from a dropdown menu for comparison. Note that all interaction capabilities described earlier are also available in the comparison mode. Differences and additional capabilities are described in the following sections. In order to compare tree-structures in a radial arrangement similar to the described "usual" SunburstView to explore a single revision a new layout-algorithm has been developed. Next, we first describe the new layout.

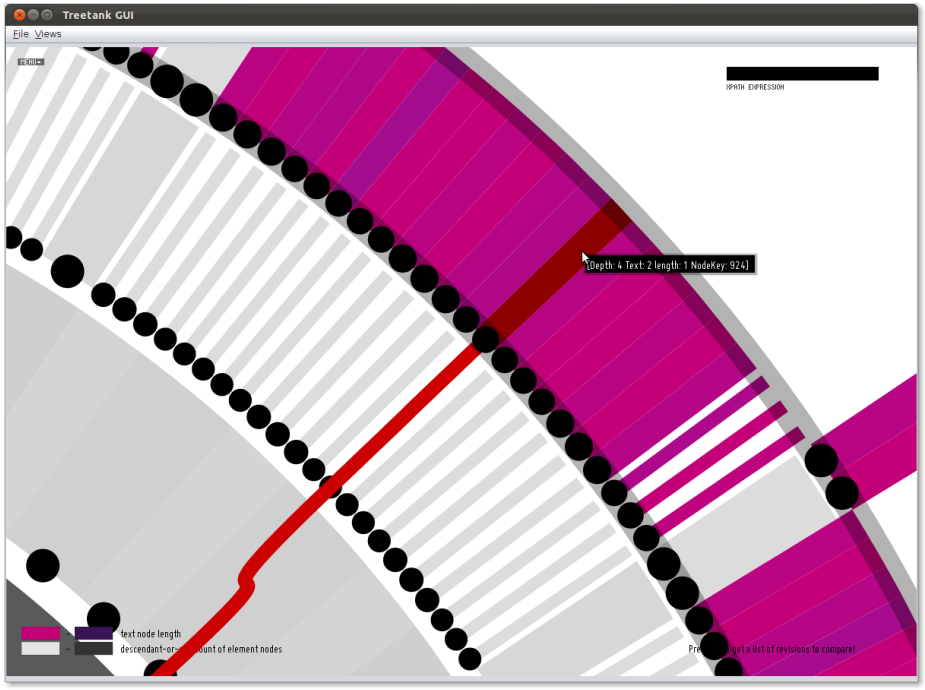

Fig. 17. Zooming into the visualization

Sunburst Comparsion-Layout The Sunburst comparison layout is illustrated in Fig. 19. Nodes are colored as depicted in the color legend in the bottom right corner. All nodes which have not changed as part of comparing the base revision to another revision[9] are plotted inside the inner circle which is labeled "matching nodes in revision 0 and revision 1". The circle itself is drawn between the maximum level of the unchanged nodes plus one and maxLevel plus two. Changed nodes are zoomed out from their original place and drawn between the two dark circles labeled "changed nodes in revision 0 and revision 1" and "matching nodes between revision 0 and 1". To demonstrate the area denoting the changes it is hatched in Fig. 19. Similarly the arrows emphasize the direction in which changed subtrees are zoomed/dislocated. This semantic zoom serves a double purpose. First, the visualization adheres to a Tree-Ring metapher depicting the evolution of a tree. Like the age of a tree in the nature is deducable by analysing rings in a cross-cut of the stem whereas the rings denote the age and each ring represents one year starting from the very middle to the outside, our representation aims at representing the changes between two rings. In our representation the unchanged nodes form the middle of the *SunburstView* whereby changed nodes are zoomed to the border between the outer and inner ring which

[9] in this case revision 1

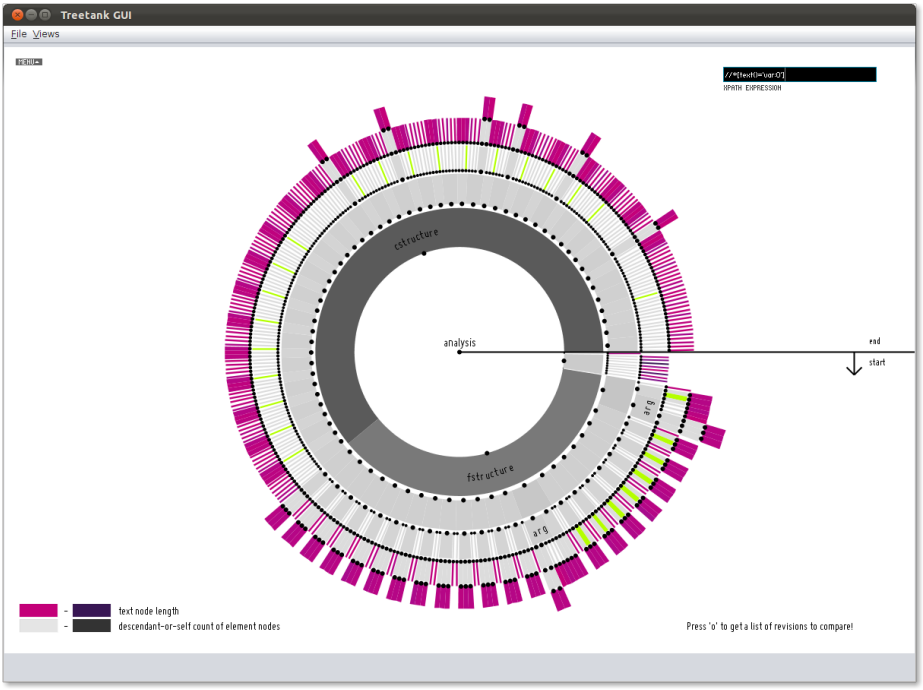

Fig. 18. XPath query results displayed in light green

is representing the growth of a tree in case of analysing temporal tree-structures. Additionally this transformation of changed subtrees displaces these subtrees to a prominent place. Small changed subtrees are thus much better noticeable as they are not surrounded by unchanged subtrees which might even be deeper. To depict changes between multiple trees first considerations involved the addition of changes from a sequence of sorted revisions in further circles around the one denoting the changed nodes. That is, the inner circle represents unchanged nodes between *all* compared revisions whereas changes between selected or consecutive revisions are drawn between new outer rings which are appended. According to the Tree-Ring metapher each comparison between a pair of trees would append a new rings for their changes. However this would affect the whole layout each time. The idea proved to be not viable because of the complexity this involves. To name a few

- The inner ring must keep space between unchanged nodes/subtrees for all changed subtrees in the outer rings.
- Consecutive calling the diff-algorithm and merging diffs into a diff-list with diff-tuples which has already been created.
- Keeping track of all opened transactions and resetting the transaction appropriately depending on the revision in which a change has occured.

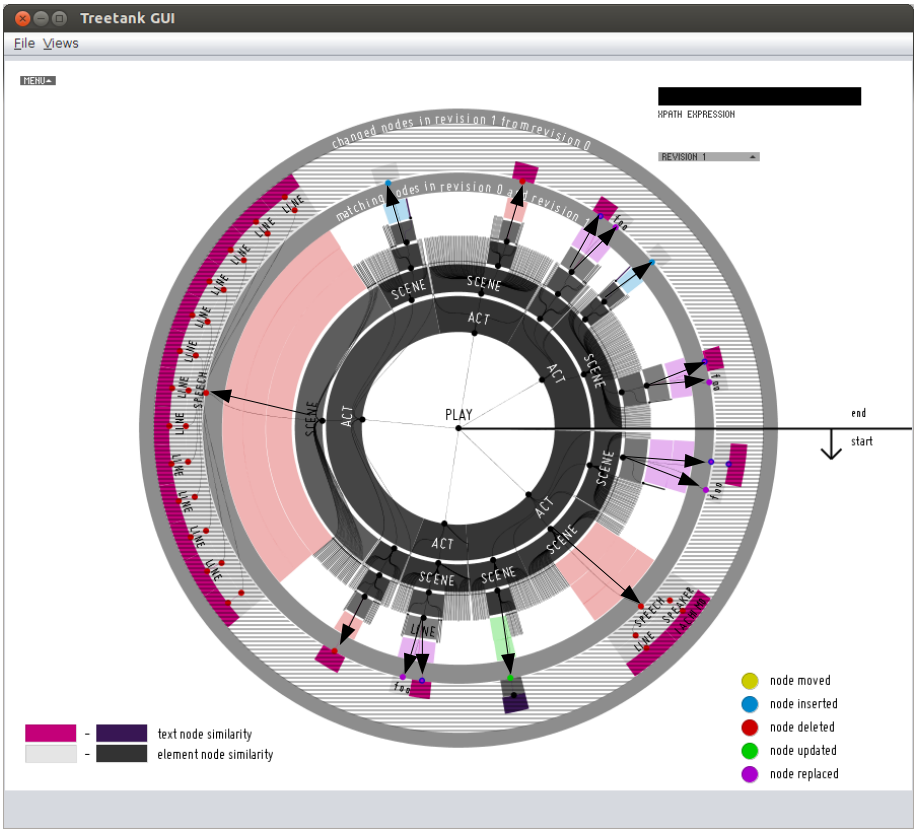

Fig. 19. SunburstView - comparison mode.

- Similarly the depth for items in the tree will change very often which involves further state and it is almost not possible to denote the current depth.
- The `descendant-or-self`- and `modification`-count of each nodes' subtree will be cumbersome to calculate.

These and similar complexity considerations formed the idea of introducing Small multiple visualizations of the current view which are described in section 4.8. Adhering to the Tree-Ring metapher, the semantic zoom surves the purpose that all changed nodes are visible on first glance. Otherwise it would be hard to track small changes.

Short animation In order to clearly demonstrate the semantic zoom a short animation has been implemented as test persons usually did not grasp the meaning without further explanation. Thus the transformation of changed subtrees is shown which dislocates the items to their dedicated positions along the arrows

in Fig. 19 [10]. However the animation is only drawn if the framerate will not drop due to too many items. Thus when a certain amount of items has been created the animation will be skipped.

Layout algorithm After invoking the ID-based diff-algorithm depending on the chosen filtering-method (described in the next section) the new Sunburst-layout has to be drawn. First of all, based on the generated aggregation of the trees, Sunburst items which denote the nodes in the aggregated tree have to be computed. The diff-tuple encountered through observing changes computed by the ID-based diff-algorithm is of the following form: key in first revision / key in second revision / depth in first revision / depth in second revision / kind of diff. The Sunburst items are created based on a special axis which implements the Iterator/Iterable interfaces from Java to traverse the diff-tuples. That is the following two methods are available: `hasNext()` and `next()`. Initially we developed an axis based on the idea of traversing the tree-structure just like in the `SunburstDescendantAxis` and to adjust a simple index-variable pointing to the next diff-tuple. The key idea is to traverse the tree-structure based on the transaction opened on the newer revision whereas usually the pointers of the nodes are used as a guidance to traverse the aggregated tree-structure in document-order. The transaction will be changed to the old revision whenever a diff of type `DiffType.MOVEDFROM`, `DiffType.DELETED` or `DiffType.REPLACEDOLD` is encountered. However the preorder-traversal using a pointer-based approach similar to the algorithms in the `SunburstDescendantAxis` or the `DescendantAxis` itself bares a lot of complexity as the node key of the next node to traverse has to be set in advance and the movement of the transaction in a lot of cases cannot be immediately reflected by adjusting certain stacks which are required to determine the start-Angle, extend, descendantCount, modificationCount and the parent index. In fact in some cases the adjustments can not be done until the next call to `hasNext()`. Figure 20 demonstrates a lot of this complexity on a simple tree-aggregation. Note that the terms "aggregation" and "agglomeration" are used exchangeable in this thesis.

If the transaction-cursor is located at the node with the nodeKey 4 in the new revision the next nodeKey will be the node denoted by nodeKey 6 in the `Descendant-` and the `Sunburst`-axis as node 5 is deleted. As such the simple expression $rtx.getStructuralNode().hasFirstChild()$ to test for a child node must be extended to keep track of these issues, for instance by receiving the next element in the diff-list and compare the diff to the type `DELETED` and `MOVEDFROM` whereby the depth must be greater than the current depth. Furthermore the kind of movement must be tracked just like in the `SunburstDescendantAxis` to adjust the stacks with the start-angle, extension, depths and descendant- as well as modification-number accordingly. Everytime before a new SunburstItem is added to a list which contains all resulting items in the end, the values as for instance the descendant-or-self number and the modification-number as well as

[10] remember, the arrows are not drawn in the visualization, they have been added to the screenshot to emphasize the transformation

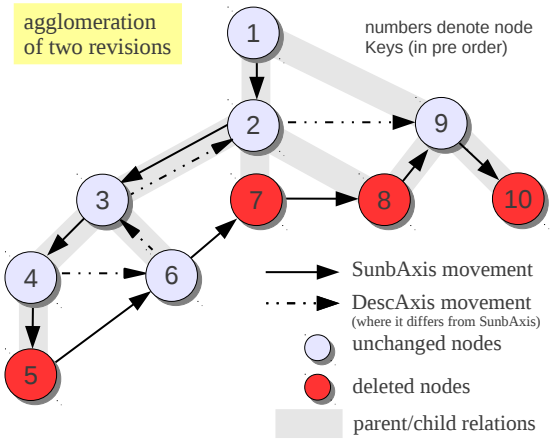

Fig. 20. SunburstCompare-Axis based on the Sunburst-Axis

the startAngle/extension must adapted based on the stacks which have to be adjusted themselfes depending on the transaction movement.

Another example of the complexity is the movement from node 6 to node 9 in the normal `DescendantAxis` whereas in our case the next node is the deleted node 7. Thus the `pop()`-method on the stacks must be called once instead of two times. Each time a `DELETED` or `MOVEDFROM` diff kind is encountered the transaction is temporarily changed along with the nextNodeKey which denotes the key the $hasNext()$-method must move to the next time called and a right-sibling node stack which is used in cases where the node has no first child and no right sibling. In this case the next nodeKey is popped from the right-sibling stack (a nodeKey is pushed on the stack for each child-movement whenever the node before the move to the first child has a right sibling) and as in the algorithm 4 the depth must be considered too, that is like in the example scenario described before such that the pop()-method is called the right number of times. This method is invoked for the first `DELETED` or `MOVEDFROM` after another diff-kind has been encountered. If the movement of the transaction before calling this method was of kind `EMoved.ANCHESTSIBL`, that is the transaction will be moved to the right sibling of the first ancestor which has one. This cannot be done until the next call to $hasNext()$ as otherwise the stacks will not be adapted appropriately. Note that the first while-iteration doesn't call pop() as no values have been pushed on the stacks for leaf-nodes. Another remark concerns the else-branch which is executed if the depth of the current node in the list is equal or even greater than the last depth.

Similarly if the movement-type was to a right sibling of the first anchestor-node which has one and the next node will not be of the kind `DELETED` and

`MOVEDFROM`, the actual movement cannot be done before the next call to $hasNext()$ and the method is very similar to algorithm 4.

The last pitfall in Fig. 20 is after traversing the node denoted by nodeKey 9. Usually $hasNext()$ will return false but in this case it must return true, as the deleted node "10" follows.

We observed that the overall algorithm is very complex in terms of a lot of special cases as described in the last paragraphs have to be considered which directly results in many comparsions and instructions and therefore might be a performance issue as well.

The complete algorithm therefore is not described as recently a new, in comparsion, lightweight algorithm has been developed. Instead of using a pointer based traversal it became apparent that it is easier to directly use the diff-tuple and move either the transaction on the old revision or the transaction on the new revision depending on the diff-kind. In case of a `DELETED` or `MOVEDFROM` the working transaction is based on the transaction opened on the old revision, otherwise the transaction opened on the new revision is used.

The algorithm skeleton for $hasNext()$ is described in 5.

The method `setupDiffs` takes care of setting the current depth which depends on the diff-kind/type, the upcoming next depth, if the list has more diff-elements as well as setting the boolean instance variable `mIsOldTransaction` which is `true` if the diff-kind is of type `DELETED`, `MOVEDFROM` or `REPLACEDOLD` and `false` otherwise. After determining which transaction must be used based on `mIsOldTransaction` if more elements are in the diff-list the transaction is moved to the next key determined by the diff-kind and obtained from the diff-element, the tuple described earlier which incorporates the nodeKey of both nodes which have been compared in the diff-algorithm as well as the depths of both nodes. In order to provide both the old value and new value[11] or QName[12] in the case of updates and similarly in case of replaces for the next SunburstItem, the transaction opened on the base revision which is always the older revision is moved to the node key of the node in the old revision. Note that this information is available from the diff-tuple. Thus both transactions are located at the right nodes. The following three `if`-clauses ensure the preorder traversal of the axis. If the next depth is greater than the current depth the next node is a child node in the agglomerated tree, thus the variables of the last created Sunburst item which depend on the stacks must be pushed on top of these. As in all cases the variable which denotes the movement must be adapted. Otherwise if the next depth is equal to the current depth the next node is a right sibling. In this case the extension of the last emitted Sunburst item has to be added to the startAngle and the movement variable must be adjusted accordingly. Likewise if the next depth is lower than the current depth the next node is a right sibling of the first ancestor which has an appropriate pointer (!= $NULL\_NODE\_KEY$ which denotes that no right sibling is available). Thus the stacks must be adapted sim-

[11] in case of a `TextNode`
[12] in case of an `ElementNode`

Algorithm 4: adjusting stacks and depths for movement to next following node for the first DELETED or MOVEDFROM node after another type has been encountered

input : instance variabled denoted by a trailing m, int initDepth, Diff lastDiffCont, EMoved moved
output: nothing, the instance variables are directly modified (method/algorithm has side effects)

```
1  int tmpDepth = 0;
2  if mDepth == 0 then
3      tmpDepth ← initDepth;
4  else
5      tmpDepth ← lastDiffCont.getDepth().getNewDepth();
6  if mMoved == EMoved.ANCHESTSIBL then
7      if tmpDepth - mInitDepth > diffCont.getDepth().getOldDepth() - initDepth
       then
8          // Must be done on the transaction which is bound to the new
              revision.
9          boolean first ← true;
10         while tmpDepth - initDepth > diffCont.getDepth().getOldDepth() -
           initDepth do
11             if first == true then
12                 // Do not pop from stack if it's a leaf node.
13                 first ← false;
14             else
15                 mDiffStack.pop();
16                 mAngleStack.pop();
17                 mExtensionStack.pop();
18                 mParentStack.pop();
19                 mDescendantsStack.pop();
20             tmpDepth–;
21             mDepth–;
22     else
23         moved ← EMoved.STARTRIGHTSIBL;
24         mAngle += mExtension;
```

Algorithm 5: Diff-Axis hasNext()-skeleton

input : instance variabled (denoted by a trailing m)
output: true, if more diffs are in the diff list, false if index == size

```
1  // Fail if there is no node anymore.
2  if !mHasNext then
3      return false;
4  // Setup everything.
5  setupDiffs();
6  if mIsOldTransaction == true then
7      setTransaction(mOldRtx);
8  else
9      setTransaction(mNewRtx);
10 // Move to next key.
11 getTransaction().moveTo(mNodeKey);
12 if mHasMorDiffKinds == true then
13     if mDiff == DiffKind.UPDATED OR mDiff ==
       DiffKind.REPLACEDOLD then
14         // For DiffKind.UPDATE or DiffKind.REPLACED the transaction
              needs to be on the right node.
15         mOldRtx.moveTo(mDiffCont.getOldNodeKey());
16     // Always follow first child if there is one.
17     if mNextDepth > mDepth then
18         return processFirstChild();
19     // Then follow right sibling if there is one.
20     if mDepth == mNextDepth then
21         return processRightSibling();
22     // Then follow next following node.
23     if mNextDepth < mDepth then
24         return processNextFollowing();
25 // Then end.
26 processLastItem();
27 mHasNext ← false;
28 return true;
```

ilar to algorithm 4. Algorithm 6 describes how the stacks, depending on the last movement, are used to adapt dependent variables for the next Sunburst item.

Algorithm 6: Stack adaptions for next Sunburst item depending on transaction movement in last call of `hasNext()`

input : IReadTransaction pRtx, Item pItem, Stack pAngleStack, Stack pExtensionStack, Stack pParentStack, Stack pDescendantStack
output: nothing, executed for it's side effects, that is adapting stacks and an item to create a Sunburst Item afterwards

```
1  if lastMovement == START_OR_RIGHTSIBL then
2      // Do nothing.
3  else if lastMovement == CHILD then
4      pItem.mAngle ← pAngleStack.peek();
5      pItem.mExtension ← pExtensionStack.peek();
6      pItem.mIndexToParent ← pParentStack.peek();
7      pItem.mParentDescendantCount ← pDescendantsStack.peek();
8  else if lastMovement == ANCHEST_RIGHT_SIBL then
9      pItem.mAngle ← pAngleStack.pop();
10     pItem.mAngle + = pExtensionStack.pop();
11     pItem.mExtension ← pExtensionStack.peek();
12     pParentStack.pop();
13     pItem.mIndexToParent ← pParentStack.peek();
14     pDescendantsStack.pop();
15     pItem.mParentDescendantCount ← pDescendantsStack.peek();
```

Semantic zoom implementation The implementation of the *semantic zoom* with changes highlighted in a dedicated place requires an adaption of the depth of the first changed node in every subtree in a preorder-traversal of the agglomerated tree to reflect the changes in its dedicated position between the two rings. Three cases have to be distinguished in which the depth must be adapted (Fig 21).

1. Transition from an unchanged node to the first changed node in a subtree (1) in Fig. 21).
2. Transition to another changed node once a (changed) subtree has been traversed and a node on the XPath `following::`-axis follows whereas its original depth would be less than the depth of the inner ring ($pMaxDepth + 2$) which only ever includes unchanged nodes (3) in Fig. 21).
3. Transition from a changed node back to an unchanged node in case the unchanged node is not in the changed nodes' subtree. An unchanged node might only be in a changed nodes' subtree if the changed node has been updated (2) in Fig. 21).

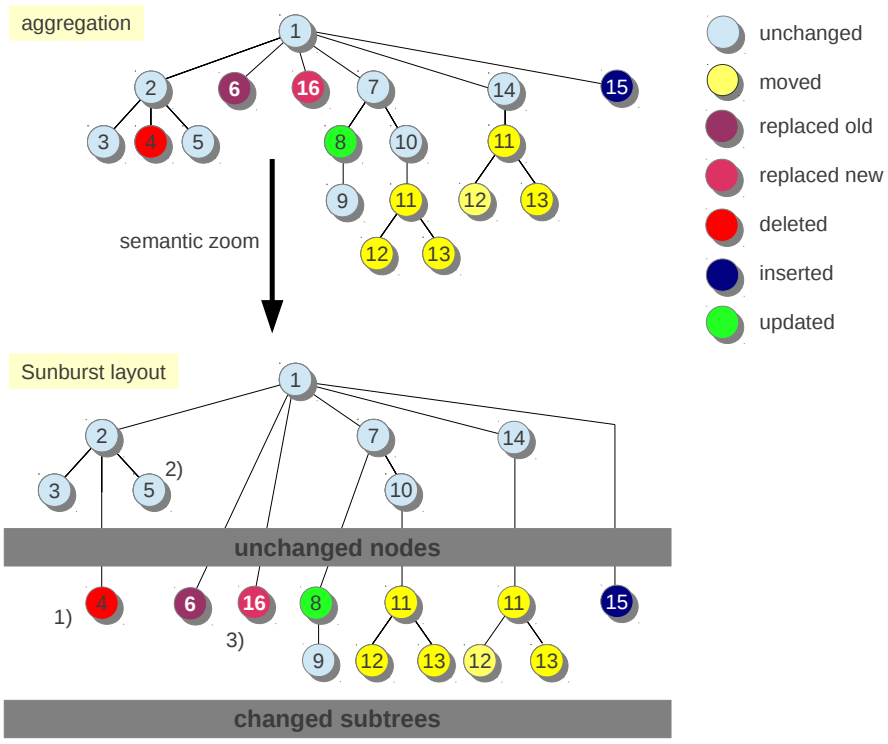

Fig. 21. Sunburst-layout depicting changes in the depth. All nodes above the grey rectangle labeled "unchanged nodes" are unchanged whereas the area between the rectangle named "changed subtrees" includes all changed subtrees. However it also includes changed nodes below an updated node as for instance node 9.

Algorithm 7 determines if and how the depth must be adjusted. The first and second case is handled by the first `if`-clause, whereas the transition back to the original depth is handled by the `elseif`-clause. An instance variable *mTempKey* is used to determine when to switch back. It is the node-ID of the first node in the XPath `following::`-axis which is not changed. However the variable always refers to the first node in the `following::`-axis which is changed in the second case. An `UPDATED` node most probably incorporates unchanged nodes if it is an internal `ElementNode`. In these cases we do not use the original depth of the node, but the current depth based on the tree traversal in the `Diff`-Axis which reflects the original depth plus the additional depth-difference to transform the depth of the first updated node.

Besides adjusting the depth the semantic zoom requires some form of highlighting. In our case we opted for a global "distortion" which enlarges modified subtrees and shrinks subtrees of unchanged nodes. Thus, the extend of a Sun-

Algorithm 7: Calculate depth

input : int pDepth, DiffType pDiff, DiffTuple pDiffCont, int pMaxDepth, PruningType pPrune, instance variables denoted by a trailing "m" (long mTempKey, long mInitDepth)
output: new depth

```
1  int depth ← pDepth;
2  if pDiff != DiffType.SAME AND pDiff != DiffType.SAMEHASH AND pDepth
   ≤ pMaxDepth + 2 then
3      depth ← pMaxDepth + 2;
4      int index ← mIndex + mPrunedNodes + mDescendantCount;
5      boolean isOldTransaction ← (pDiff == DiffType.DELETED OR pDiff ==
       DiffType.MOVEDFROM OR pDiff == DiffType.REPLACEDOLD);
6      DiffDepth depthCont ← mDiffCont.getDepth();
7      int depth ← 0;
8      if isOldTransaction == true then
9          depth ← depthCont.getOldDepth();
10     else
11         depth ← depthCont.getNewDepth();
12     if index < mDiffs.size() then
13         Diff nextDiffCont ← mDiffs.get(index);
14         DiffType nextDiff ← nextDiffCont.getDiff();
15         boolean nextIsOldTransaction ← (nextDiff == DiffType.DELETED
           OR nextDiff == DiffType.MOVEDFROM OR nextDiff ==
           DiffType.REPLACEDOLD);
16         DiffDepth nextDepthCont ← nextDiffCont.getDepth();
17         int nextDepth ← 0;
18         if nextIsOldTransaction == true then
19             nextDepth ← nextDepthCont.getOldDepth();
20         else
21             nextDepth ← nextDepthCont.getNewDepth();
22         mTempKey ← 0;
23         if nextIsOldTransaction == true then
24             mTempKey ← nextDiffCont.getOldNodeKey();
25         else
26             mTempKey ← nextDiffCont.getNewNodeKey();
27 else if (pDiff == DiffType.SAME OR pDiff == DiffType.SAMEHASH) AND
   pDiffCont.getNewNodeKey() == mTempKey then
28     depth ← pDiffCont.getDepth().getNewDepth() - mInitDepth;
29 return depth;
```

burst item depends on two variables, the `descendant-or-self`-count and the number of `modification`s in a nodes' subtree.

$$ext = \begin{cases} 2 \cdot \pi & \text{if } node\ is\ root\ node \\ parExt \cdot ((1-\alpha) \cdot descs/(parDescs - 1) & \\ \quad +\alpha \cdot mods/(parMods - 1)) & \text{otherwise} \end{cases} \quad (4)$$

The modification-count in the forumla is derived from the `descendant-or-self`-count added to the `modification`-count of the current node plus a constant. The addition of the `descendant-or-self`-count is needed to handle nodes, which do not contain any modifications in its subtree and the node itself is not changed, too. In order to further emphasize and enlarge subtrees with a small number of modifications a constant is added which has proven useful in empirical studies (Chapter 5). Futhermore note that if the parent node has been modified the constant must be subtracted from the parent `modification`-count in advance. Otherwise the equation will not work.

The two variables are computed in parallel to the `Diff-Axis`-traversal whereas the results are appended to a `BlockingQueue`, which is a thread safe queue designed for producer/consumer relationships. Modifications for the root node are gathered while observing diff-tuples, thus the modifcation-count does not need to be computed afterwards as for the other nodes in the agglomerated tree-structure. A simple heuristic determines depending on a depth-threshold if tasks are executed in the calling thread instead of another thread, as context switches for very small subtrees are too costly. Observe that the added descendant-count for each node can not be used, because the aggregated tree-structure is traversed which incorporates deleted and moved nodes. Algorithm 8 depicts how the two variables are derived from traversing the diff-datastructure at a specified index until the depth of a diff in the agglomerated tree-structure either is less than or equal to the start depth or no more diff-tuples are following. Note that the depth depends on the diff-type. In case of a `DELETED`, `MOVEDFROM` or `REPLACEDOLD` diff-type the depth of the node in the older revision is used, otherwise the depth of the node in the new revision. Furthermore recapitulate that the depths are computed by the ID-based diff-algorithm instead of stored directly in the node by Treetank.

Filtering/Pruning Providing an initial Sunburst overview of huge tree-structures in reasonable time, ranging from a few seconds to a few minutes, requires pruning techniques to filter nodes of no or least interest. Therefore three types of filtering are provided. Changes in a nodes' subtree are always guaranteed to be visible. The following screenshots in this section are related to Fig. 22 which depicts the same tree comparison in the SunburstView without filtering nodes.

- *by itemsize* Sunburst items which have no changes in its subtree and are too thin to perceive individually or too thin to select even with the fisheye transformantion are pruned based on a predefined threshold-value. An example is depicted in Fig. 23. This type of filtering is useful wherever nodes

Algorithm 8: Derives `descendant-or-self`-count as well as the `modification`-count

input : int pIndex, List pDiffs
output: CONSTANT_FACTOR * diffCounts, descendantCounts, subtract

```
1  int index ← pIndex;
2  DiffTuple diffCont ← pDiffs.get(index);
3  DiffType diff ← diffCont.getDiff();
4  int rootDepth ← diffCont.getDepth().getNewDepth();
5  if diff == DiffKind.DELETED OR diff == DiffKind.MOVEDFROM OR
   currDiff == DiffType.REPLACEDOLD then
6      rootDepth ← diffCont.getDepth().getOldDepth();
7  int diffCounts ← incrDiffCounter(index);
8  int descendantCounts ← 1;
9  boolean subtract ← false;
10 diffCounts = incrDiffCounter(index);
11 index ← index + 1;
12 if diffCounts == 1 AND index < mDiffs.size() then
13     DiffTuple cont ← mDiffs.get(index);
14     int depth ← cont.getDepth().getNewDepth();
15     if cont.getDiff() == DiffType.DELETED OR cont.getDiff() ==
       DiffType.MOVEDFROM OR currDiff == DiffType.REPLACEDOLD then
16         depth ← cont.getDepth().getOldDepth();
17     if depth == rootDepth + 1 then
18         // Current node is modified and has at least one child.
19         subtract ← true;
20 boolean done ← false;
21 while !done AND index < mDiffs.size() do
22     DiffTuple currDiffCont ← mDiffs.get(index);
23     DiffType currDiff ← currDiffCont.getDiff();
24     DiffDepth currDepth ← currDiffCont.getDepth();
25     int depth ← currDepth.getNewDepth();
26     if currDiff == DiffType.DELETED OR currDiff ==
       DiffType.MOVEDFROM OR currDiff == DiffType.REPLACEDOLD then
27         depth ← currDepth.getOldDepth();
28     if depth ≤ rootDepth then
29         done ← true;
30     if !done then
31         descendantCounts ← descendantCounts + 1;
32         if currDiff != DiffKind.SAME AND currDiff != DiffKind.SAMEHASH
           then
33             diffCounts ← diffCounts + 1;
34         index ← index + 1;
```

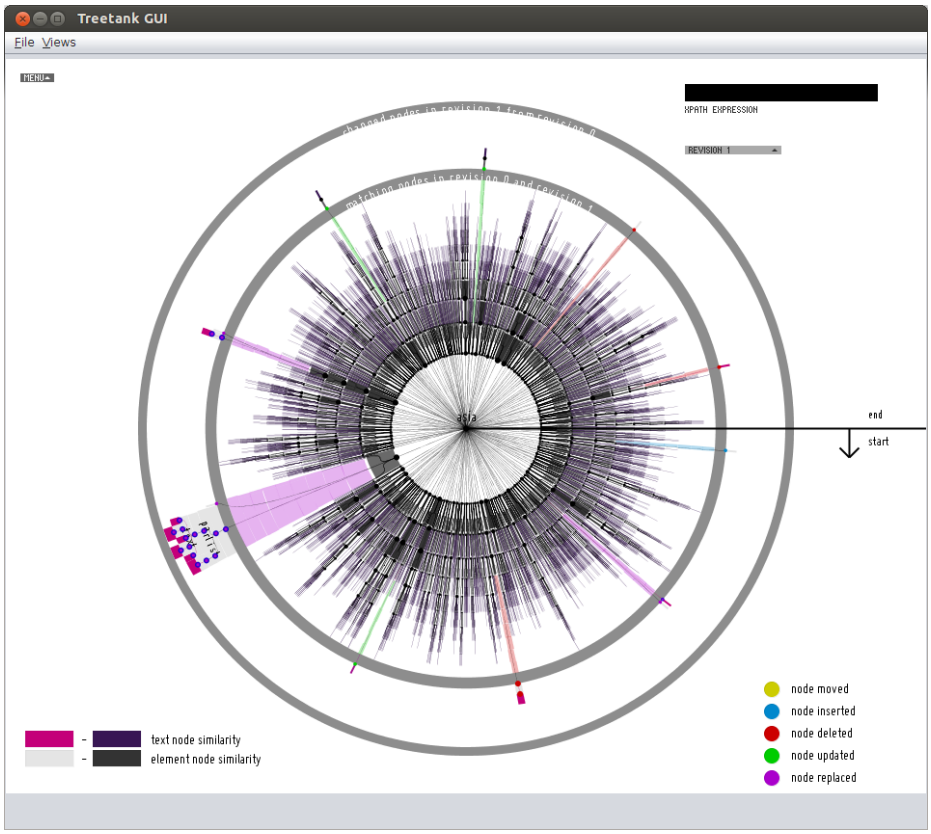

Fig. 22. Comparison without pruning.

which do not differ are of value but depicting the whole tree will not add any significant value and/or does not . It considerably speeds up the generation of the Sunburst items in large tree-structures, however it does not effect the diff-calculation. Furthermore if a new node is selected to drill down into the tree the item creation must be redone, based on the `Diff`-axis, thus no optimization which just creates new items based on the initial set with adjusted angles can be used. The `descendant-or-self`-count as well as the `modification`-count must be recalculated.

- *by hash-based diff-algorithm* The diff-algorithm is called with the option to utilize persisted hashes which are created for every resource based on a database-configuration paramete. Per default rolling hashes which are very fast are used. The hashes of ancestor-nodes are recomputed for all edit-operations. As described in Chapter 3 everytime the hash values during node-comparisons are identical the whole subtree is skipped in both revisions. Thus, only items are created for nodes, which contain changes in their subtree including subtree-roots for which identical hash-values are deter-

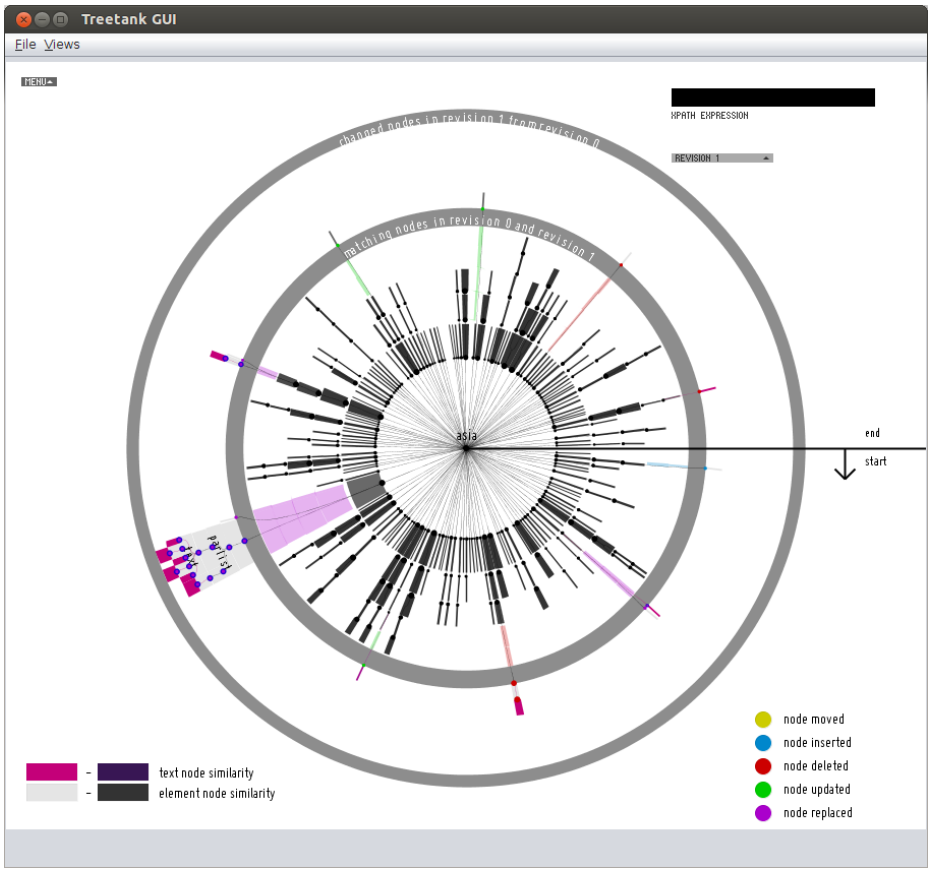

Fig. 23. Pruned by itemsize.

mined (Fig. 24). This type of pruning is especially useful for large tree-structures, whereas in contrast to the pruning by itemsize it speeds up the diff-computation as well as the item creation, as in both cases subtrees of nodes with the same hashes are skipped. However, it might be that more items in comparison with the itemsize-based approach are created as nodes with identical hash-values are always included.

- *by hash-based diff-algorithm without nodes which have the same hash* This type of pruning is closely related to the filtering-method described above. In addition to filtering nodes in the subtree of nodes which have the same hash-value, items for this type of "difference" are not created at all. A consequent is a much better readability of the differences in case of many consecutive nodes with the same hash-value (Fig. 25).

Usually the maximum depth of unchanged nodes in the aggregated tree-structure is computed in parallel using a `BlockingQueue` if more than two cores

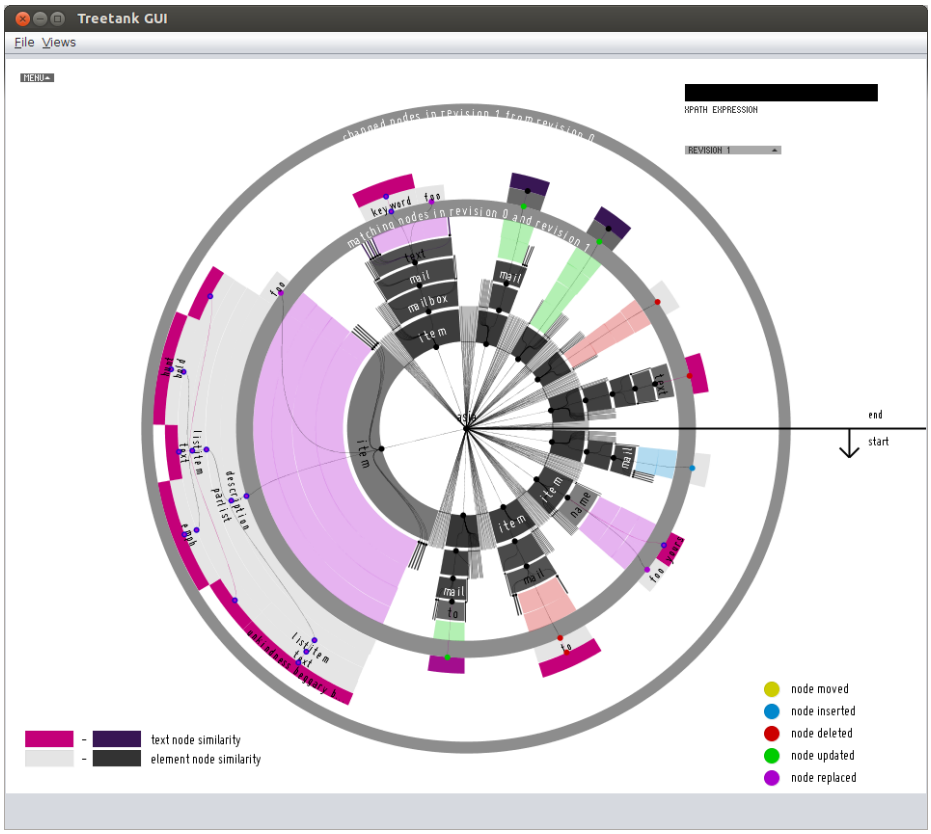

Fig. 24. Pruned by same hashes.

are available taking diff-tuples off the head of the queue during observation. However, all filtering types require postprocessing as maximum depth nodes might either be within the subtree of an unchanged node and in case of the itemsize-based filtering might be too thin regarding a threshold value. Thus, the maximum depth must be recomputed and the depth of unchanged nodes must be adapted and reduced accordingly. Note, that otherwise too much space would be wasted as the root of changed nodes is plotted at *maxDepth of unchanged nodes* plus two.

Having described several filtering methods and briefly meantioned a general zooming/panning technique based on affine transformations which is inherited from the *SunburstView* layout depicting one revision. Besides it is of the utmost importance to enlarge specific subtrees in the layout itself. The next section gives some insight on how to the selection of new root-nodes to dig deeper into the tree-structure is handled.

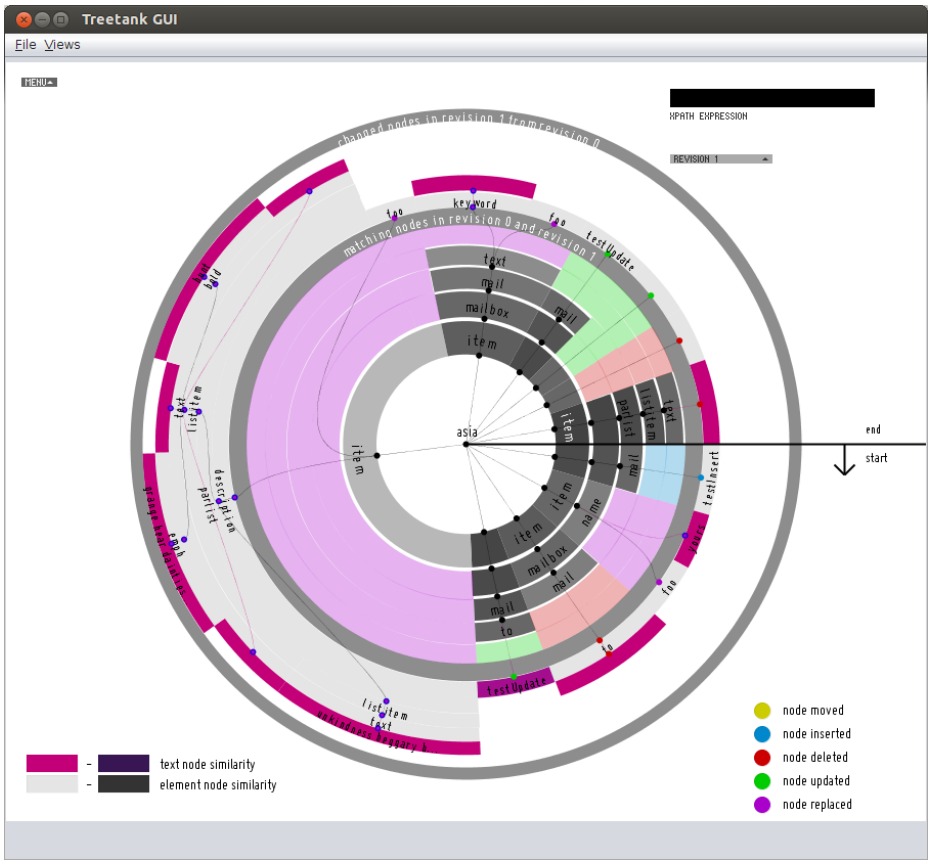

Fig. 25. Pruned by hash-value without building items for nodes with same hash-values.

Zooming / Details on demand Supporting analysts with the ability to dig deeper into the tree-structure for most but the tiniest tree-structures is crucial. Thus we support the selection of a new root node by a mouse-click on the desired item. In our first version this triggered the recalculation of the diffs for the nodes' subtree as well as the the maximum depth of unchanged nodes, the modification- and the descendant-count for each node in the subtree to build new enlarged items. The former items are pushed on a stack to support a simple undo-operation.

However due to unnecessary recalculations despite in case of the item-based pruning our simplyfied approach just recalculates the maximum depth of the unchanged nodes in parallel utilizing all available cores and subsequently builds new items based on a simple upscaling.

Having described the new Sunburst-layout tailored to comparison of tree-structures with all available pruning methods and the ability to select a new

root-node in detail the next section describes how the similarity score between different types of nodes is measured, which is used to map colors to the items.

Similarity measures Similarities for `Element`- and `Text`-nodes are measured differently. The only inner nodes of an XML-document are element nodes (which can also be leaf nodes).

`Element`-node similarity is measured based on overlapping subtrees. Consider the comparison of tree-structures T1 and T2. The similarity score for an element in the aggregated tree-structure T_{aggr} is defined as

$$Sim(node_{T_{aggr}}) = \frac{descs(node_{T_{aggr}}) - mods(node_{T_{aggr}})}{descs(node_{T_{aggr}})} \tag{5}$$

The similarity score is normalized between $[0, 1]$. The nodes of type `INSERTED/DELETED/REPLACED` always are scored 0 as they do not have any equivalent in the other tree. In case of `SAME`-nodes the similarity depends on overlapping subtree-structures. If no modifications are in the subtree the score is 1. `UPDATED` nodes are counted as a modifications and therefore add to the dissimilarity.

`Text`-nodes are leaf nodes which therefore have no child. The similarity score is defined as

$$Sim(node_{T_{aggr}}) = \begin{cases} Levenshtein(node_{T_1}, node_{T_2}) & \text{if } node \text{ } is \texttt{ UPDATED} \\ 0 & \text{if } node_{T_{aggr}} \text{ } is\texttt{INSERTED} \\ & \texttt{/DELETED/REPLACED} \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

Note that the similarity score is computed based on the diff-tuple which incorporates both nodeKeys. In case of an `UPDATED text`-node the Levenshtein algorithm is used, which defines a similarity score based on per-character edit-operations to change one string-value into the other one and is normalized between 0 (no similarity) and 1 (same string-value).

4.6 Querying

XPath 2.0 queries are currently partly supported by our XPath 2.0 engine. We also examined our Saxon binding, which was surprisingly rather slow. However we did not measure time differences which is out of scope of this thesis. Future work might include a Brackit[25] binding which is a "flexible XQuery-based query engine" developed at the TU Kaiserslautern. In contrast to Saxon it is specifically designed to work on top of databases and adding specific indexes for instance will be easy. However this feature is currently reevaluated.

We provide query capabilities on top of the agglomerated tree-structure and therefore query both compared revisions in parallel. The items are sorted by node-ID in parallel. Once all query results have been collected they are also

sorted (the result is a sequence of node-IDs). A subsequent traversal of the items highlights items which are included in the query results.

4.7 Visualization of moves

Hierarchical Edge Bundles[26] are used to avoid visual clutter of subtree moves. The technique creates a path up to but usually not including the Lowest Common Ancestor (LCA) of a source-node down to the destination-node. The path is used to define control points for plotting a curve.

The LCA is defined as:

$$lca(a,b) = min\{c|a \prec c, b \prec c\} \tag{7}$$

A simple algorithm computes the LCA through the `push()`-operation on two stacks following the first node (moved from) and the second node (moved to) up to the root-node. In a next step the two stacks are processed in a loop using `pop()` as long as identical node-IDs are found. The LCA is the first node-pair for which the node-IDs do not match.

Fig. 26 illustrates two move-operations on a subtree in a shakespear XML-document.

Despite comparing two tree-structures we furthermore aim to support a broad overview about the changes (and similarities) of currently at most five trees.

4.8 Small multiple

Small multiple visualizations of the *SunburstView* are therefore used to provide an overview about the changes between several tree-structures, whereas the restriction of comparing five tree-structures is merely an implementation detail which in future versions will be configurable. Two variants based on same-titled well known revisioning strategies as well as a similarity based variant are described next.

- *differential* The differential variant displays changes related to a base revision. This is especially useful if several tree-structures have to be compared to a common base revision. The direction is clockwise. That is the upper left corner displays a SunburstView comparison between revision 0 and 1 (if zero is the base revision which is loaded), the right upper corner displays changes between revision 0 and revision 2 et cetera (Fig. 27).
- *incremental* The incremental variant displays changes related to the last revision in increasing order. Suppose we have loaded revision 2, then in the upper left corner revision 2 and 3 is compared, the upper right corner contains the comparison between revision 3 and 4 et cetera (Fig. 28).

The implementation parallizes the computation of the small multiple displays and stores each Sunburst-visualization in an offscreen image, which is appended to a list in the view. This list must be sorted after all visualizations have been

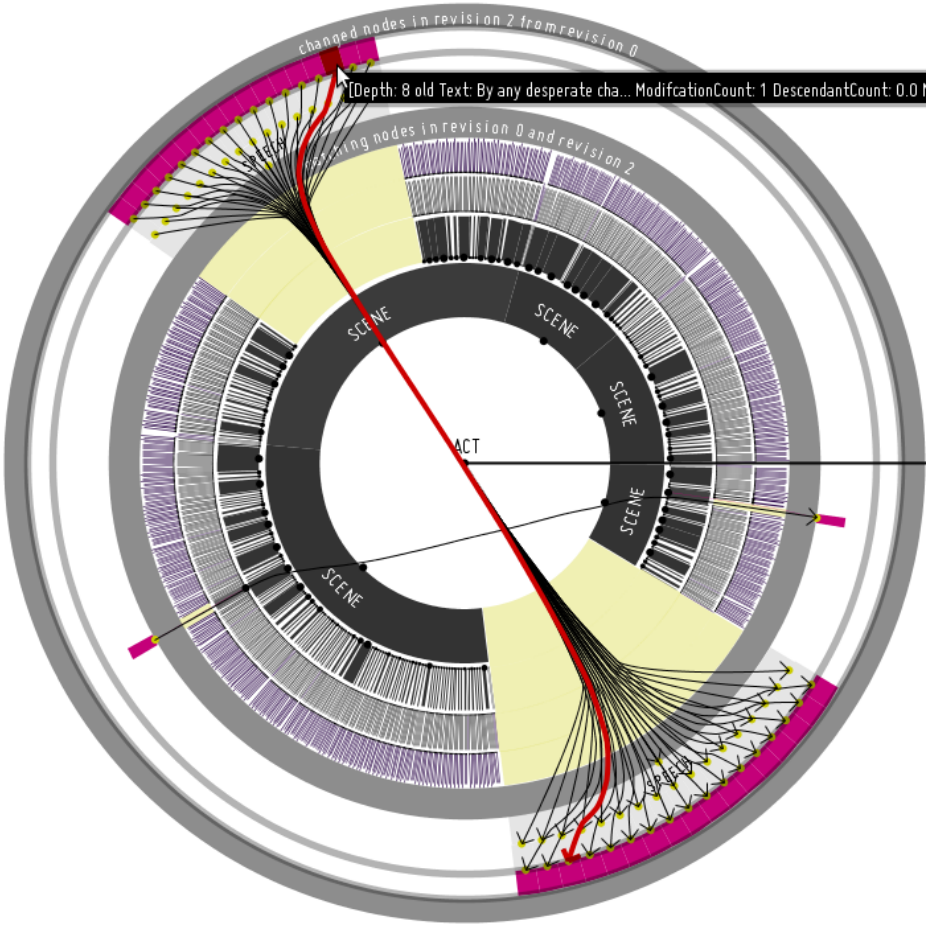

Fig. 26. Moves visualized using hierarchical edge bundles.

computed accoring to a revision which is saved along with the buffered offscreen image to support the clockwise order. The items of each visualization are saved in a simple datastructure in the model to support highlighting of items on mouse over through brushing and linking. The technique is illustrated in Fig. 27 and 28. Items are highlighted in red just like in the SunburstView but the same item based on a node-ID equivalence relation is linked and highlighted in all small visualizations. Note that we aim to highlight whole subtrees on hovered nodes in a future version such that moves in the subtree during upcoming comparisons in the incremental variant will be easily visible resulting in forests.

The two variants support all filtering methods described earlier. Next we provide a short asymptotic runtime- and space-analysis backed by performance measures.

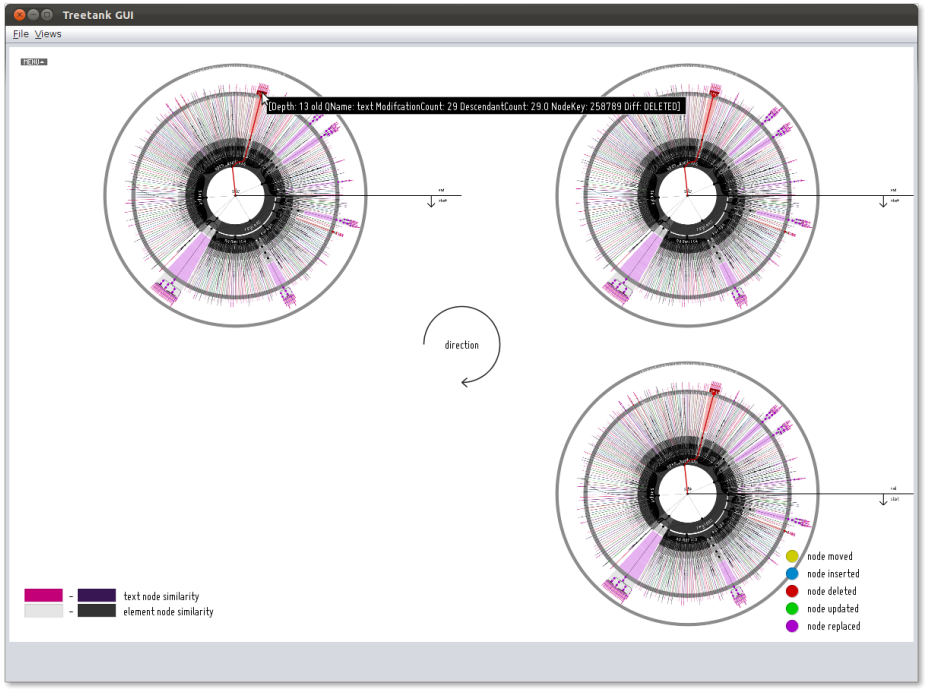

Fig. 27. Small multiple - differential variant.

4.9 Runtime/Space analysis and scalability of the ID-based diff

The runtime of the algorithm currently is bound by determining the subtree-size and modifications in the agglomerated tree-structure for each node. The runtime complexity thus is $O(n^2)$ whereas n is the sum of changed and unchanged nodes between two trees $T1$ and $T2$. However by storing the diff-tuples which only include the depth of the two compared nodes to form a very simple tree-structure we are limited to a preorder traversal. Building a more sophisticated tree-structure based on pointers or sequences denoting child nodes for instance in an in-memory Treetank structure will support a subsequent postorder traversal to determine the subtree-size and modifications for each node reducing the runtime to $O(n)$. Due to using Java7 which is not available for OS X we were not able to measure the performance on computers with four or more cores. However, counting the subtree-size and modifications is started in parallel to building Sunburst items which consumes these through a Java `BlockingQueue`. The `descendant-count/subtree-size` of the root-item (plus one) is the size of the accumulated List or Map of diff-tuples observed from the ID-based diffing algorithm and thus does not need to be computed. The `modification-count` of the root-item is also determined on the fly while observing diff-tuples. Thus we assume adding more cores will speed up the creation of the items considerably as tree-structures often times are rather flat and have a large fan-out

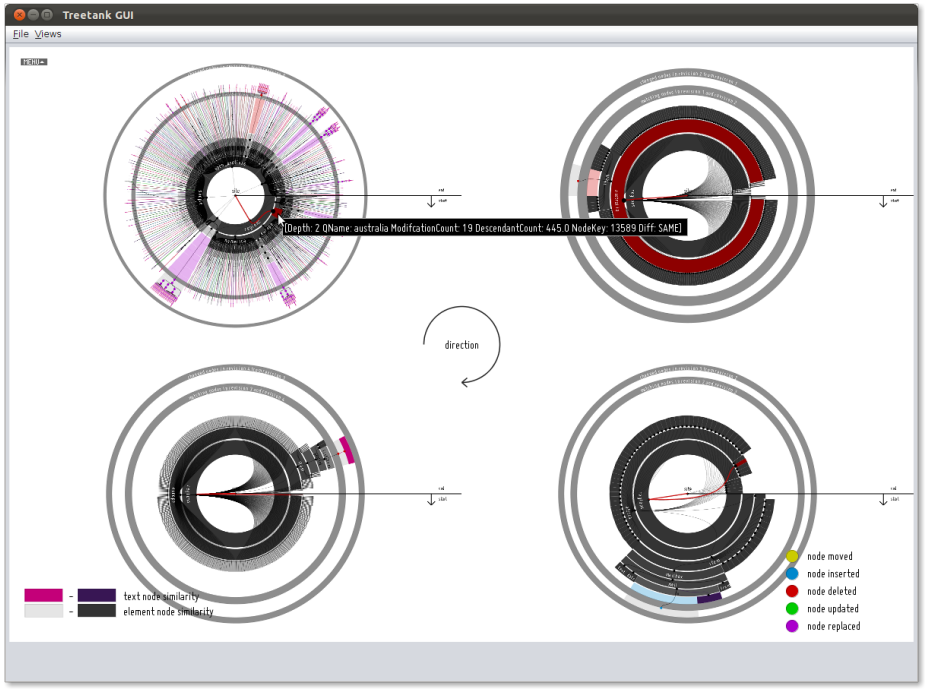

Fig. 28. Small multiple - incremental variant.

instead of being deep with high average levels/depths of the nodes, especially considering document-centric XML [6]. Fig. 29 shows performance measures, the average of ten runs of the Sunburst-visualization on an 11MiB-document and an 111MiB-document of the XMark-benchmark. The documents are identical to the documents used in Chapter 3 for benchmarking. Thus we randomly modified the instances after every 1000st node in the 11Mib-document and after every 10000st node in the 111MiB-document. The hardware used is also identical (Core 2 Duo 2666Mhz, 4Gb RAM). It is obvious that the exponential growth in case no pruning is enabled is unacceptable. Showing the 111MiB-document without pruning lasted too long (¿ 15min for each run) such that we aborted the execution. By reducing the number of Sunburst-items which have to be created considerably each one of the pruning-mechanisms reduces the runtime tremendously. Usually the number of Sunburst-items to create is reduced so much that the exponential time to compute the modifications in each nodes' subtree plus the subtree-size itself is not measurable and the runtime reduces to a linear scale. Furthermore as we were not able to measure the impact of parallelizing this task with four and more cores it might also considerably speed up the computation in the general case without pruning. We assume that the context switches with only one or two cores in fact slow down the computation.

Fig. 29. GUI-performance.

	1000mods	**5000mods**	**10000mods**
min	36265.14	35717.34	33948.96
max	52459.38	61860.29	44451.06
average	39167.75	37650.67	36579.37

Table 6. Comparsion of different modification-schemas of a 111 MiB XMark instance (change every 1000st, 5000st and 10000st node).

Fig 30 shows benchmarking results comparing the runtime of the fastest pruning, pruning-by-hashes without creating items for identical hashes with move-detection enabled and disabled. We are able to determine that the move-detection usually is fast. Asymptotically it is bound by the size of the agglomerated tree-structure n, $O(n)$.

In summary we are able to conclude that the hash-based-pruning without creating Sunburst items is by far the fastest and that the move-detection usually does not inhibit the runtime of our algorithms considerably. Furthermore it is required to use pruning whenever the exponential time required to compute subtree-sizes and modifications therein significantly inreases due to a large aggregated tree. Adding more CPUs however should decrease the runtime as the computation of subtreesizes and modifications therein are computed in parallel.

4.10 Conclusion and Summary

We have introduced a multitude of visualizations ranging from a simple *TextView* displaying syntax highlighted serialized XML in the viewport appending new text during scrolling to a *SunburstView* and several *Smallmultiple* display variants

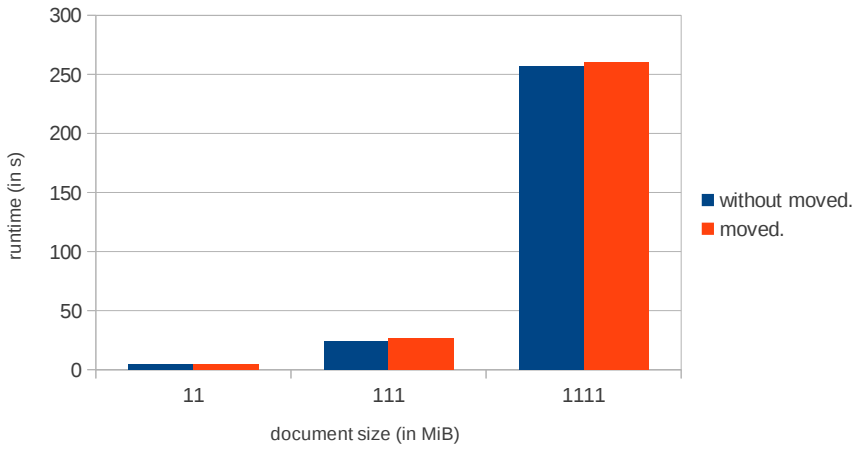

Fig. 30. GUI-performance using hash-based pruning without adding identical hash-values and move-detection enabled/disabled.

based on the *SunburstView*. The *TextView* supports the visualization of structural differences based on the aggregated tree structure and through a color-coded background depending on the diff-type. However non-structural changes, for instance the similarity between updated String values can not be visualized. Our main contribution is a new Sunburst layout based on the idea of a semantic zoom, which places all changed nodes prominently between two rings and facilitates a global distortion to enlarge changed-subtrees and thus to shrink unchanged subtrees accordingly which are potientially uninteresting. Structural changes are visualized through color coding of the node in the overlaying node-link diagram. The type of diff is indicated through a color-encoding. Furthermore the similarity of `TextNode` values and `ElementNode`s are depicted based on predefined functions. Large matched subtrees result in a higher similarity score for element nodes, few character replacements/inserts/deletes increases the similarity score for text nodes.

Moreover three filtering techniques facilitate the analysis of large tree-structures. Pruning by itemsize is useful if changed *and* and unchanged nodes are of importance speeding up the creation of items considerably. However it does not affect the diff-computation. Thus, we also provide hash-based filtering techniques, which utilize the diff-algorithm with the optimization to skip the traversal of subtrees of nodes with identical hash-values. These filtering types usually reduce the number of items even more and accelerate the diff-computation. Note that the hashes include the unique node identifiers and other node-specific content, thus by using an exchangeable cryptographic hash-function th.

Move-detection is enabled on demand. Furthermore inserted/deleted subtrees in a row are summarized as replace-operations.

In addition to the *TextView* and *SunburstView* small multiple variants support the comparison of multiple trees (> 2). The number of comparsions is only limited by our implementation (which is only an implementation-detail) and the available screen space. Note, that the small multiple displays currently use too much unused screen space due to storing the whole visualization except the legends and menus in an offscreen image which are downscaled afterwards. As the extends of the main GUI-window usually are not squarified the space usually used for drawing menu-components and legends in the *SunburstView* is multiplied and wasted. However this is merely an implementation detail which will be fixed in a future version. The filtering techniques are also available in the small multiple variants.

5 Applications

5.1 Introduction

The last chapters described in detail the different components involved in the Visual Analytics approach of detecting and visualizing differences in tree-structures. This chapter exemplary studies three usage scenarios and describes additional preprocessing steps if necessary. To demonstrate the feasability of our approach with real world data several the following three applications are studied:

- Import of Lexical Functional Grammar (LFG) XML exported files.
- Import of sorted Wikipedia articles by revision-timestamps.
- A Directory recursively watched for changes within a filesystem. The XML representation thereof is based on FSML[27].

5.2 LFG

Linguists often face the problem of tree-structure comparisons as for instance Abstract Syntax Trees (ASTs). The LFG (Lexical functional grammar) in particular differenciates between two structures:

1. Constituent structures (c-structures) which "have the form of context-free phrase structure trees".
2. Functional structures (f-structures) "are sets of pairs of attributes and values; attributes may be features, such as tense and gender, or functions, such as subject and object."[28]

We obtained an XML-export of a collection of different versions of a c-structure/f-structure combination. To import differences between these XML documents in Treetank the FMSE-algorithm described in Chapter 2 and 3 is used as the XML-export does not include unique node identifiers. Thus we can not rely on node-IDs and have to Fig. out differences using our similarity metrics defined for leaf- and inner-nodes. Fig. 31 illustrates the visualization of a diff between two revisions.

It is immediately obvious by investigating the original XML-documents, that the FMSE-algorithm mismatched nodes in the first place which in effect causes a lot of edit-operations. Nodes are touched which have not changed whereas changed nodes might or might not be concerned by edit-operations. Thus the FMSE-algorithm executes too many edit-operations especially in case of the deletion and reinsertion of the old- respectively the new-`cstructure` subtree. In doing so, a subsequent visualization is impractical, for the simple reason that it almost mirrors the update-operations for consecutive versions. As stated in Chapter 2 ID-less algorithms work best if leaf nodes and therefore especially text-nodes are very well distinguishable. That is the initial matching of leaf nodes in most if not all cases really matches unchanged nodes, which is the desired behavior.

Having a closer look at the input documents reveals changed text-values which due to their short length and to sometimes numerical values have changed completely.

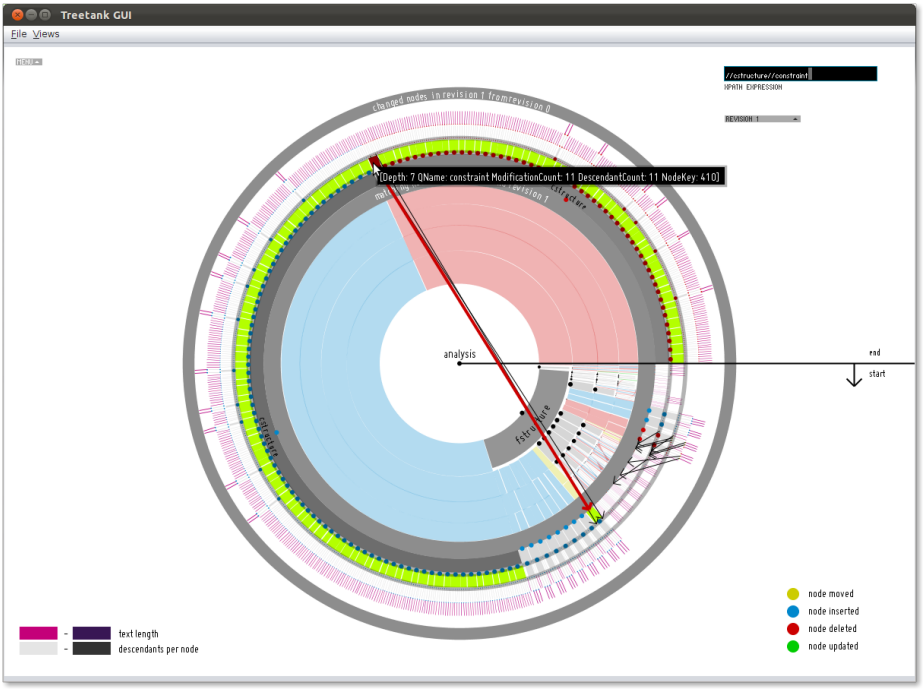

Fig. 31. LFG comparsion

Listing 1. CStructure/FStructure comparsion

```
<cstructure>
  <constraint>
    <label>subtree</label>
    <arg no="1">102</arg>
    <arg no="2">ROOT</arg>
    <arg no="3">-</arg>
    <arg no="4">83</arg>
  </constraint>
  <constraint>
    <label>phi</label>
    <arg no="1">102</arg>
    <arg no="2">var:0</arg>
  </constraint>
   ...
</cstructure>
```

```
<cstructure>
  <constraint>
    <label>subtree</label>
    <arg no="1">163</arg>
    <arg no="2">ROOT</arg>
    <arg no="3">-</arg>
    <arg no="4">145</arg>
  </constraint>
  <constraint>
    <label>phi</label>
    <arg no="1">163</arg>
    <arg no="2">var:0</arg>
  </constraint>
   ...
</cstructure>
```

Thus we changed the leaf-node comparison slightly. Instead of just relying on the Levenshtein-distance for text-node values we also try to match all ancestor `QNames` [13] if the returned normalized value is <= the threshold value defined for

[13] ancestors are always element nodes, the document-root node is not visited

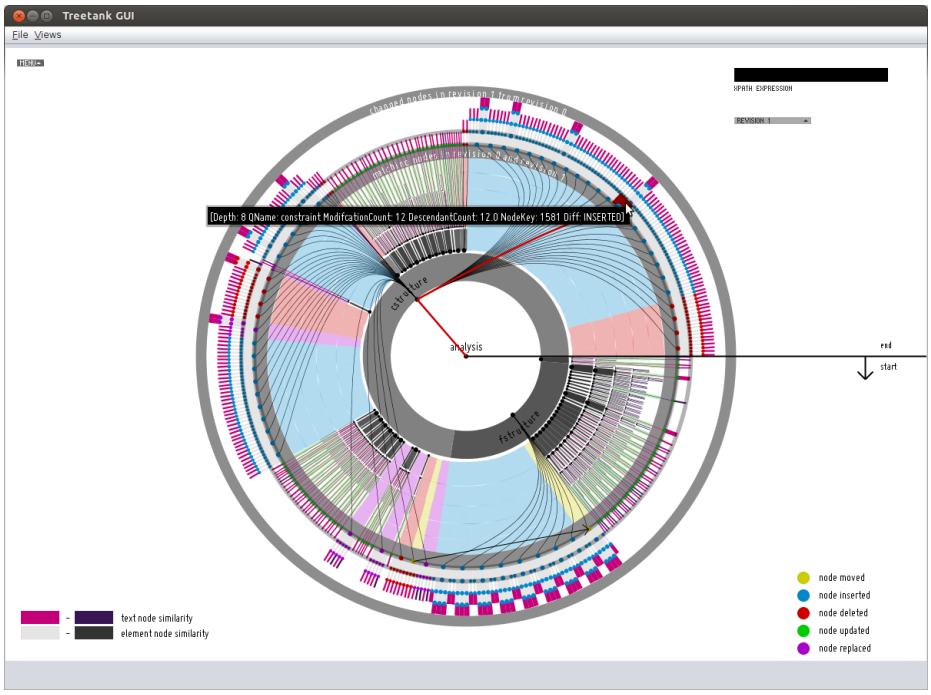

Fig. 32. LFG comparsion revised

the similarity of leaf nodes. Note that only leaf nodes are considered to match if the threshold is $>$ than the predefined threshold.

The result is depicted in Fig. 32.

We observe that still many nodes are changed, but by looking at the original XML-documents we observe that indeed a lot of nodes have been updated/deleted/inserted or replaced. For instance by scrolling both XML-documents at the end of the fstructure-subtree a lot of subtrees must have been inserted.

However as a direct consequence of this short evaluation it is inevitable to rely on unique node identifiers if we want to visualize minimal edit-scripts in case of updated leaf node values which are very distinct to the other trees' leaf node values or worse the leaf nodes are very similar. Comparisons based on similarity-metrics are always based on heuristics. Assuming the export of the XML-documents includes unique identifiers of the nodes, importing requires utilization of the internal diff-algorithm to simply store incremental changes, the changes from the old- to the new-revision. The following steps have to be implemented:

1. Import of the initial XML-document.
2. Import the first updated XML-document to a temporary resource.
3. Utilize the internal Treetank diff-algorithm to determine changes based on unique node-IDs. Whenever an edit-operation is encountered the appropriate

transaction-method has to be executed. In case of `element`- and `text`-nodes it requires how to insert nodes[14] and where, which is very simple due to the `transaction.moveTo(long)`-method which can move the transaction/cursor directly to the left-sibling-node of the node to insert if it is available or to the parent. In the first case the node has to be inserted as a right sibling, in the latter case as a first child of the parent node.

Furthermore this approach applies to every XML-document which incorporates unique node identifiers.

Next we study the import of several articles from Wikipedia ordered by timestamp of their revisions.

5.3 Wikipedia

Wikipedia is studied as an application to demonstrate the feasability of our approach on a large text corpora. However several issues have to be solved during preprocessing.

- Wikipedia is dumped as a very large XML-file. "'The XML itself contains complete, raw text of every revision"' [29] and thus is a full dump instead of an incremental- or differential-dump describing the changes an author performed. Listing 5.3 depicts a small example of the structure of an Wikipedia XML-dump. Moreover WikiText, which is a proprietary markup format is stored as plain text and not replaced by XML-markup. Thus, the content of an article in each revision is a huge `TextNode` which includes the full text instead of just denoting the changes and its context in some form. Differences between the actual content of several revisions of an article on a node-granular level can not be found with a state of the art native XML database system, which supports revisioning of the data, as a direct consequence. The database systems' best bet is to generate a character based delta between the two large text-nodes.

Listing 2. Wikipedia Dump - XML representation

```
1 <mediawiki xml:lang="en">
2   <page>
3     <title>Page title</title>
4     <id>67365</id>
5     <restrictions>edit=sysop:move=sysop</restrictions>
6     <revision>
7       <timestamp>2001-01-15T13:15:00Z</timestamp>
8       <contributor><username>Foobar</username></contributor>
9       <comment>I have just one thing to say!</comment>
10      <text>A bunch of [[text]] here.</text>
11      <minor />
12    </revision>
```

[14] remember that Treetank currently is able to insert a node as a right sibling or first child

```
13      <revision>
14        <timestamp>2001-01-15T13:10:27Z</timestamp>
15        <contributor><ip>10.0.0.2</ip></contributor>
16        <comment>new!</comment>
17        <text>An earlier [[revision]].</text>
18      </revision>
19    </page>
20
21    <page>
22      <title>Talk:Page title</title>
23      <id>127455</id>
24      <revision>
25        <timestamp>2001-01-15T14:03:00Z</timestamp>
26        <contributor><ip>10.0.0.2</ip></contributor>
27        <comment>hey</comment>
28        <text>WHYD YOU LOCK PAGE??!!! i was editing that jerk</text>
29      </revision>
30    </page>
31  </mediawiki>
```

- Furthermore due to the fact that XPath 2.0 should be usable without requiring an XPath Full Text 1.0 implementation and our overall goal is to analyse the temporal evolution of the stored data, WikiText markup has to be converted into semistructured XML fragments. To accomplish this the Wiki2XML parser from the MODIS team [30] is used.
- Another issue arises because the Wikipedia dump is not sorted and we want to analyse and visualize the temporal evolution during several snapshots. Neither articles are sorted by date/time nor revisions of the articles are. Thus for the subsequent revisioned import the revisions have to be sorted with their associated article metadata (page id, author...). For the simple reason that we have to deal with large amounts of data an external sorting algorithm has to be used. Instead of implementing an own approach, Hadoop is a natural choice. Its *MapReduce* framework is a "programming model and software framework for writing applications that rapidly process vast amounts of data in parallel on large clusters of compute nodes". The overall process of the programming model is divided into two functions.

1. `Map` is a function that has to split the problem into subproblems which can be distributed among worker nodes in a cluster. Logically it is a function of the form $Map(k1, v1) \rightarrow list(k2, v2)$ where $k1$ and $v1$ is an input *key* and *value* and the output from the function is a list of *key*/*value* pairs ($list(k2, v2)$). After that the MapReduce framework groups the values according to the keys.
2. `Reduce` is a function of the form: $Reduce(k2, list(v2)) \rightarrow list(v3)$. It receives a *key* and a list of grouped *values* and returns performs any computation which might be feasable and returns another list of *values*.

Using the `map`-function means input data has to be split into records. Most MapReduce frameworks rely on a programmable mechanism to do this. In Hadoop

the `InputStream` in conjunction with a `RecordReader` takes this responsibility. The following steps describe the implementation of sorting Wikipedia by Hadoop.

1. `XMLInputStream` in conjunction with an `XMLRecordReader` splits records on *revision-elements* with all the *page*-metadata. The **timestamp** of each revision denotes the key, the whole subtree of each *revision*-element is saved as the value for the `map` input. The algorithm is straight forward. A `SAX-Parser` is used to determine starting and ending of each record.
   The implementation utilizes a configurable record identifier such that it is adjustable to schema changes in the Wikipedia dump. Furthermore it can simple be adapted to split other XML files based on other record identifier nodes as well.
2. `XMLMap` just forwards the received key/value pairs.
3. `XMLReduce` finally merges consecutive revisions with the same *page-id* and **timestamp** by means of Saxon, an XSLT/XQuery-processor. The XSLT stylesheet used is pretty small yet maybe not straight forward. It is shown in listing 3.

Listing 3. XSLT stylesheet to combine consecutive pages/revisions

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs">

  <xsl:output method="xml" indent="no"
    omit-xml-declaration="yes" />
  <xsl:strip-space elements="*" />

  <xsl:template match="/">
    <xsl:copy>
      <xsl:for-each-group
        select="descendant-or-self::page"
        group-by="concat(id, revision/timestamp)">
        <page>
          <xsl:copy-of select="* except revision" />
          <xsl:for-each-group
            select="current-group()/revision"
            group-by="xs:dateTime(timestamp)">
            <xsl:apply-templates
              select="current-group()" />
          </xsl:for-each-group>
        </page>
      </xsl:for-each-group>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="revision">
```

```
29      <xsl:copy-of select="." />
30    </xsl:template>
31 </xsl:stylesheet>
```

Once the dump is sorted with MapReduce it has to be imported into Treetank. First of all, as we have splitted the data on `revision`-elements and prepended the `page`-metadata we have to construct a new root node, which is simply done by prepending a `mediawiki` start-tag, then writing the result of the Hadoop-run and appending a corresponding end-tag to a new file. Otherwise it is no valid XML-document and can not be parsed.

Next, to import the revisioned file which is now in ascending timestamp order of all revisions from all pages/articles a special Wikipedia-Importer is responsible. It utilizes the FSME-implementation described in detail in Chapter 2 and 3 to just import incremental changes between the latest stored revision in Treetank and a shredded List of `XMLEvent`s in a temporary database/resource. Therefore as data is read-in from the XML document with a StAX-Parser the page-metadata as well as the whole revision-subtree is saved in a simple main-memory collection, a `List`, assuming the XML-content of an article can be put in main memory. The assumption holds true as every revision can be parsed and rendered in a web-browser. Everytime a `page` end-tag is encountered the page-id is searched in the already shreddered revision. If it is found the FMSE algorithm is called for the found `page`-element, such that the encountered differences are shreddered subsequently. In case the XPath expression returns no results, that is the page-ID is not found a new page is going to be appended as a right sibling to the last `page`-element.

The Importer for Wikipedia allows the usage of a timespan (hourly, daily, monthly, yearly) which is used for the revisioning. If a new timestamp is encountered, which differs from the current timestamp regarding the timespan the transaction is commited.

Fig. 33 demonstrates the result of importing 50 articles of Wikipedia sorted by article-revisions and an hourly commit [15]. Revisions 70 and 71 are compared. The *edit-distance* used by our FSME-implementation to determine and update the stored Treetank-data with the encountered differences in the first place works reasonably well, as most text-nodes can be distinguished very well. We directly used the hash-based pruning to enlarge regions of interest. The *TextView* on the right side is particular useful to quickly reveal several changes whereas in the *SunburstView* only the new updated text content in case of `TextNode` updates is used for the label (and usually the content is too big to be displayed within the arc, especially if the content contains whole paragraphs as is usually the case for Wikipedia). We quickly detect that a lot of paragraphs have changed by looking at the *SunburstView*. Details are depicted by either clicking on an item or scrolling down in the *TextView*.

[15] only hours in which changes occured are reflected

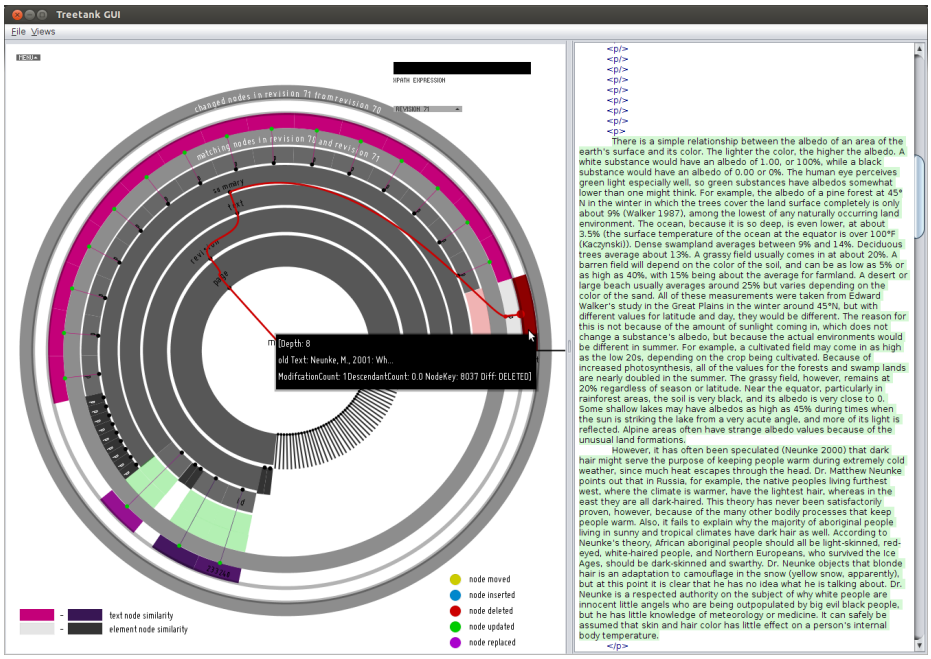

Fig. 33. Wikipedia comparsion

Another visualization (Fig. 34) depicts what happens if we do not prune and do not include the number of modifications of a node in its subtree to scale the SunburstItems, the segments for each node.

The changes in the last page/article are almost non visible. Most items denote unchanged nodes with no modifications in their subtree and thus do not carry any useful information, if we just want to quickly determine changes. This and the fact that we do not include the modification weight, that is just the subtree size of each node matters, results in considerably reduced itemsizes of changed items.

Despite viewing changes between subsequent revisions which in case of importing only 50 articles most often results in appending or updating one article it is possible to view changes between arbitrary revisions. Fig. 35 reveals changes between revision 90 and 106. Thus seven articles have changed and it is very easy to determine regions of interest and to further drill down into the tree through selection of a new root node in the *SunburstView*. The page/article with the name "ArgumentForms" has been changed enormously. The page-nodes do not have enough matching subtree-nodes in common between the two revisions, such that the page-nodes are not matched, too. The FMSE-algorithm thus deletes the whole article and inserts the article as a new first child of the "wikimedia" root-element. The changes in the other articles mainly either include single paragraphs which are updated or larger subtrees which are replaced. Detailed changes are

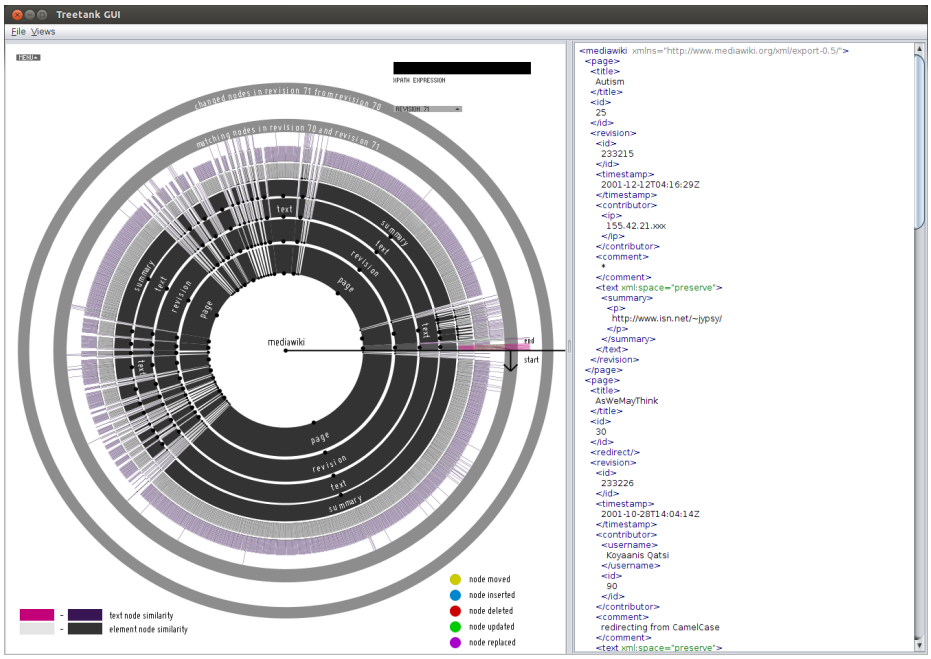

Fig. 34. Wikipedia comparsion without pruning and including the modification weight to determine the size of a SunburstItem

depicted on demand, that is if we drill down into the tree and/or scroll down in the *TreeView*. The transparent red square in the *SunburstView* in Fig. 35 illustrates the article/page to which we have scrolled in the *TextView*.

Besides comparing only two revisions the smallmultiple variants facilitate the comparison between several revisions (currently at most five revisions). Fig. 36 illustrates changes between revisions 70,71 (upper left), 71,72 (upper right), 72,73 (bottom right) and 73,74 (bottom left). We are quickly able to determine that except in the SunburstView comparing revisions 73 and 74 all other small multiples change or delete/insert the same article. The comparison between revision 70,71 and 72,73 reveal a lot of updated paragraphs. Between revisions 71 and 72 the same article has been deleted and inserted. We are quickly able to determine that a lot of paragraphs have been added as the inserted article includes more descendants than the article in the old revision. Thus the import FMSE-algorithm can not match the page-nodes due to very different subtrees. Repacitulate that in order to match inner nodes, at least half of the nodes in both subtrees have to be matched. To gain a better understanding about the differences between revision 73 and 74 we switched to the *SunburstView* in conjunction with the *TextView*. We are able to quickly determine that the deleted and inserted page denotes the same article (autism) once more (Fig. 37). The

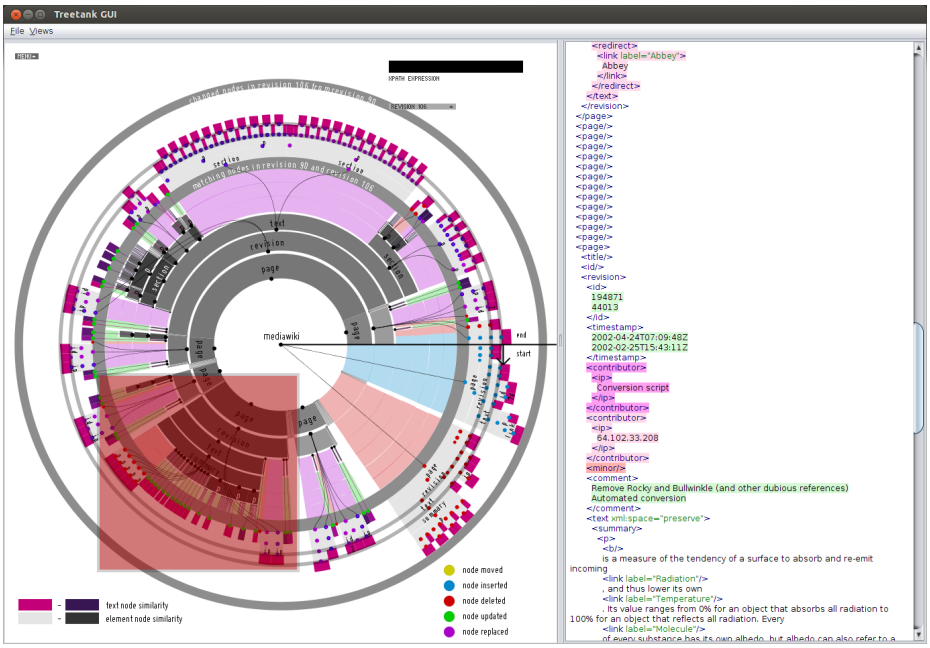

Fig. 35. Wikipedia comparsion pruned by itemsize

article changed from a single paragraph which just included a URL to a slightly more profound description of autism.

Fig. 38 finally reveals that between revision 10 and 13 articles are added which is expected. Articles are most probably going to be altered in later revisions once most or all 50 articles have been prepended. However revision 14 updates an article added in one of the first 10 revisions.

5.4 Import of Filesystem-based tree-structure

Filesystems usually organize data in a tree-structured hierarchy whereas a unique *Path* denotes how a file can be located. As a direct consequence it is an optimal use case to demonstrate the feasability of our proposed approach as the tree-structure ca not be neglected if we want to quickly detect differences in the folder/file-structure based on snapshots.

Use cases are manyfold, ranging from monitoring software-evolution based on the package/class hierarchy to monitoring if employees stick to organizational policies.

In order to take snapshots of a directory with all subdirectories we study two approachs:

1. FSML representation based on a python script [27], which has been executed several times to obtain snapshots based on an XML-representation which are

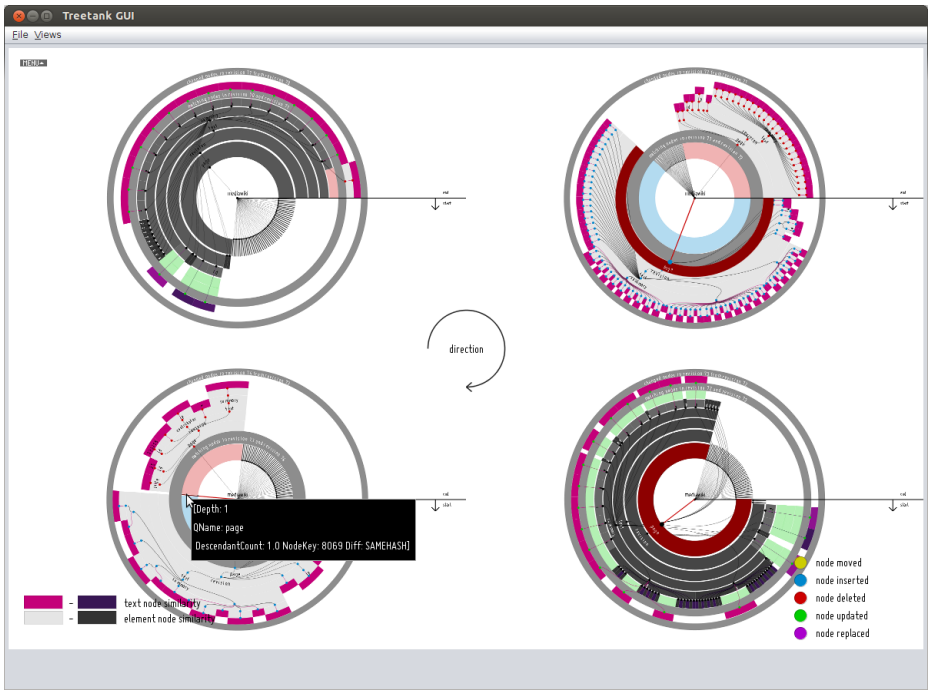

Fig. 36. Wikipedia comparsion - depicting differences through the incremental smallmulltiple variant

afterwards imported in Treetank. The differences between the snapshots are calculated based on the FMSE-algorithm described in chapter 2 and 3.

2. FSML representation based on an initial import of a directory with all its subdirectories. Changes in that directory and all subdirectories are afterwards monitored.

The File System Markup Language[27] is an XML-dialect developed by Alexander Holupirek to represent the tree-structure of filesystems by folders and files as well as metadata of certain file-types.

The first approach is considerably simpler than the import of Wikipedia due to the snapshot-creation which results in different files. Therefore we do not have to reorder whole subtrees according to timestamps in the first place to obtain a subsequent import ordered by time. Instead applying the FMSE-algorithm on the latest imported revision and the new XML document is sufficient.

Fig. 39 reveals the differences between revision 0 and 4 using the hash-based pruning on manually taken snapshots of the src-folder of our GUI project (slightly outdated) with a Python-script, developed by Alexander Holupirek, too. We are able to detect possible hotspots of development on the package/class-level. It is immediately obvious that a ForkJoinQuicksort implementation has been deleted. Another interesting observation is the recent work on BSplines

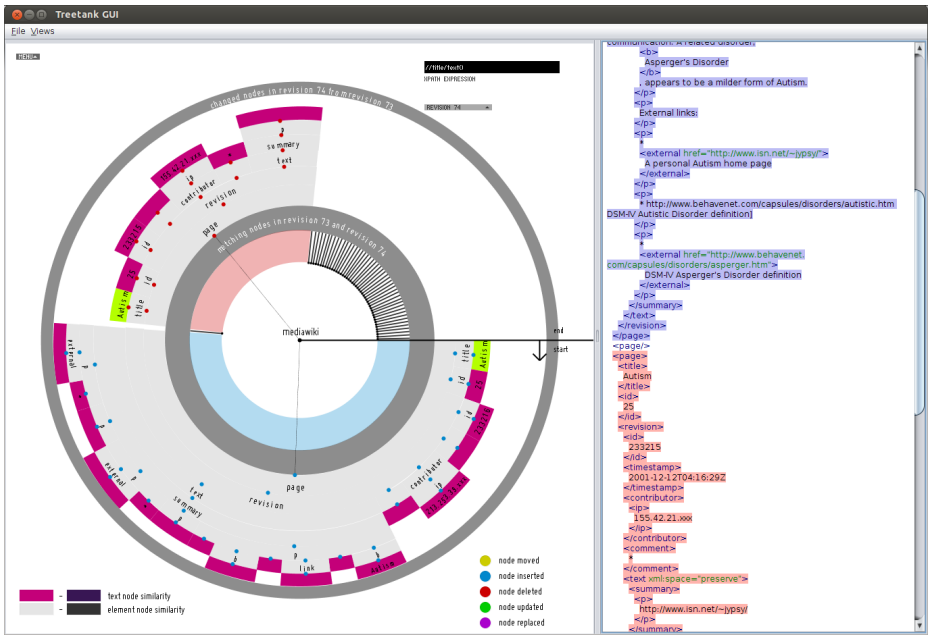

Fig. 37. Wikipedia comparsion - depicting differences between revision 73 and 74

which have been added to facilitate the Hierarchical Edge Bundling technique described in Chapter 4. Furthermore we are able to spot recent work on a new *TreeMap*-view. The XPath-query `//element()[@st_mode="0100664"]` highlights nodes whith the appropriate mode.

The alert reader might think that some of the nodes for instance the subtree of the "sunburst"-subtree which are plotted should not have been processed by the ID-based diff algorithm but by close inspection it is revealed that these nodes have different hash values (on mouseover the details are revealed, that is type `DiffType.SAME` instead of `DiffType.SAMEHASH`). Moreover up until now we have always used the ID-based diff mode which does not include namespace/attribute comparisons. However once including checks for namespace/attribute equivalence we are able to detect updates of certain nodes (Fig. 40) due to last access time of a file/directory which is denoted through the `st_atime`-attribute. Each time either a namespace- or an attribute-node differs the parent `ElementNode` is emitted as being updated.

We conclude, that the result represents the real changes in the GUI src-folder. That is the preprocessing step of matching nodes in the FMSE-algorithm works reasonably well on FSML-data as no similar `TextNode`s are included. However we aim to support the extraction of metadata for several kinds of files which might introduce the possibility of mismatches in the future.

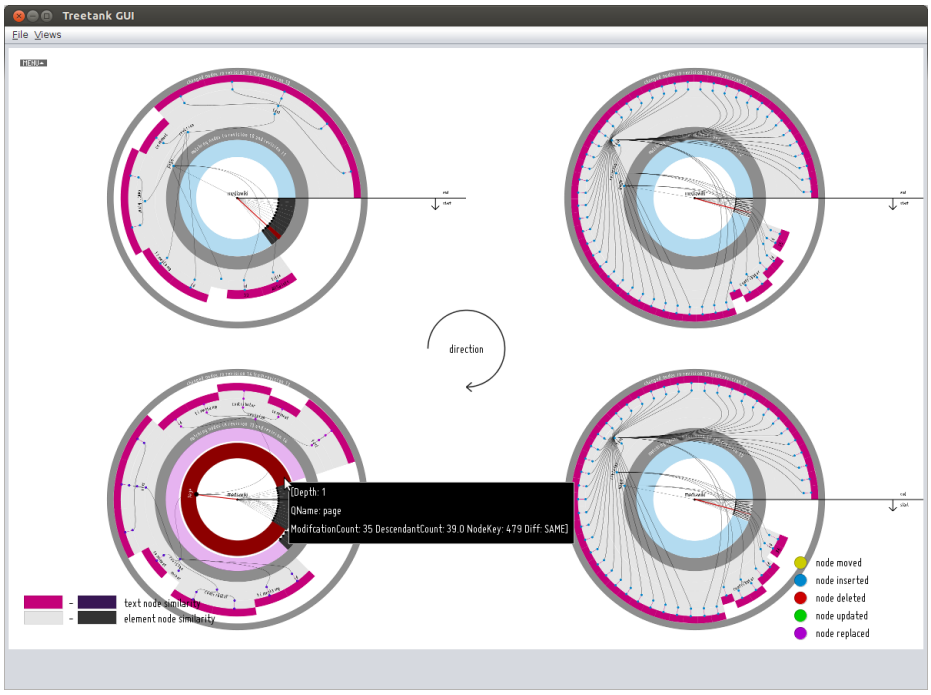

Fig. 38. Wikipedia comparsion - incremental smallmultiple variant depicting changes between revisions 10,11,12,13 and 14 from the upper left to the bottom left in clockwise order.

In order to avoid any mismatches and therefore too many update-operations on nodes which have not changed at all as well as to avoid the costly execution of the FMSE-algorithm in the first place the capabilities of current filesystems to register for modification-events are additionally used. The steps are as follows:

1. To obtain the hierarchical structure of filesystems the Java7 Filesystem-Walker API is used instead of the Python-Script. A new database is created in Treetank with a standard resource named "fsml" whereas the hierarchical structure is mapped to the resource while traversing a directory.
2. The new `WatchService` is used to detect subsequent changes in a watched directory. In order to support watching all subdirectories as well a suitable datastructure as for instance an associative array has to be used in the first place. Java does not permit recursive watching of all subdirectories, as it is not supported by some filesystems.

   The `WatchService` detects the following events:

- `ENTRY_CREATE`: New file or directory has been created.
- `ENTRY_DELETE`: File or directory has been deleted.

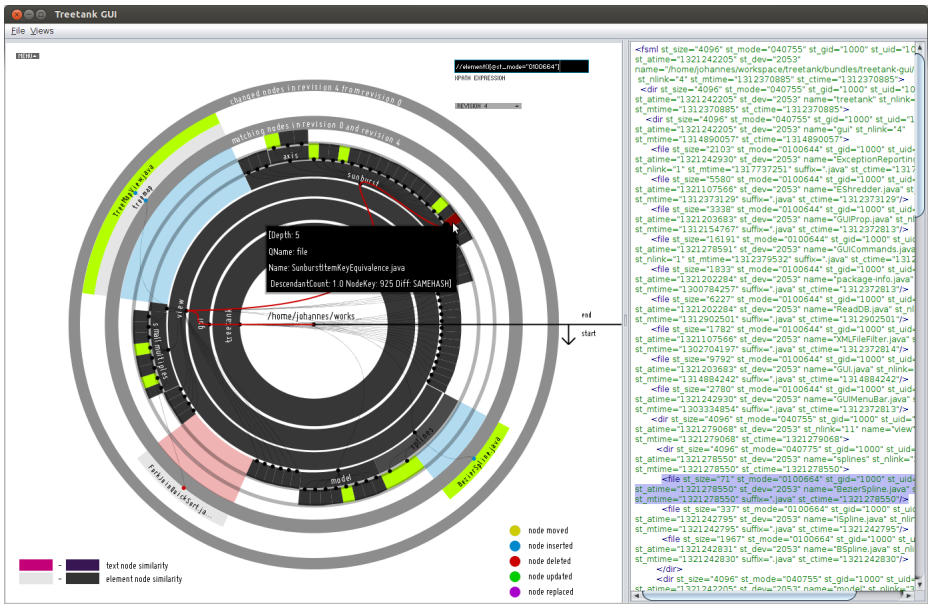

Fig. 39. FSML comparsion on the GUI src-folder

- `ENTRY_MODIFY`: File has been modified.

Therefore moves and renames are not supported out of the box. The path to the directories and files is translated to an XPath query, which locates the appropriate node in the database/resource. In case a new file or directory has been created a new `ElementNode` is prepended as a first child of the parent node as filesystem-trees are unordered. In this case the XPath-query translates the parent-path into the appropriate XPath-query, that is it is of the form `/dir[@name='pathComp1Name']/dir|file[@name='pathComp2Name']`. The path component usually denotes a directory whereas the last component might be a directory or file in case of an `ENTRY_DELETE` event. Deleted directories have to be removed from a WatchService/Path-mapping in an associative array. Furthermore all paths have to be saved in another datastructure to denote if the deleted path pointed to either a file or directory. Thus another associative array is used to save a Path ⇔ EPath mapping for inserted nodes. EPath is a Java enum to determine the type.

The FSML-subdialect currently used is very simple. Listing 5.4 provides an example of the simple structure. Directories are mapped to `dir`-elements, whereas files are mapped to `file`-elements. Note that the names can't be used to denote the elements, as for instance whitespaces are not permitted in QNames. Thus the labels in the visualizations are optionally based on `name=""`-attributes.

Listing 4. FSML structure

```
1 <fsml>
```

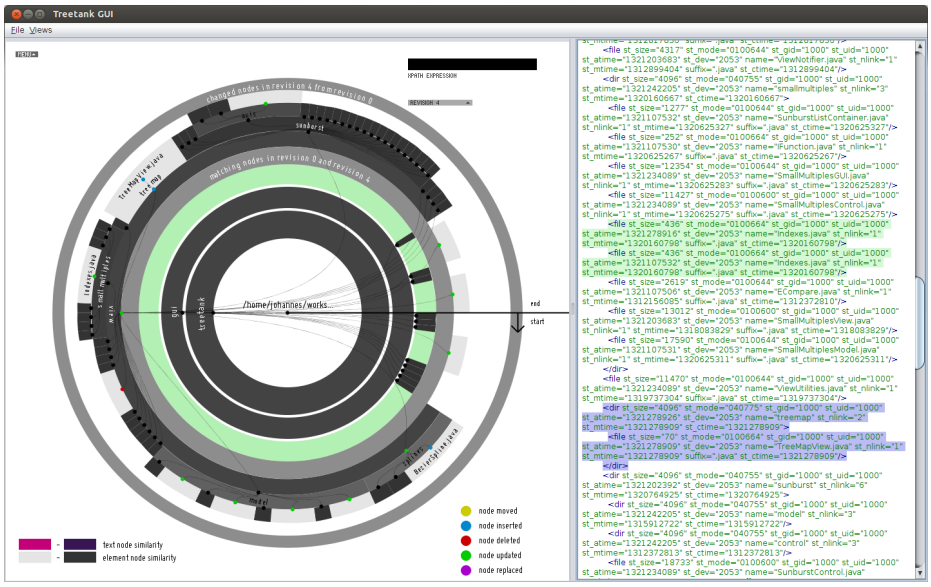

Fig. 40. FSML comparsion on the GUI src-folder using a full diff including namespaces and attributes

```
<dir name="Desktop">
  <dir name="Lichtenberger">
    <dir name="Bachelor">
      ...
    </dir>
    <dir name="Master">
      <dir name="Thesis">
        <dir name="Fig.s">
          <file name="fsml-incremental.png" suffix=".png"/>
          ...
        </dir>
        <dir name="results">
          <file name="1gb-400" suffix=""/>
          ...
        </dir>
        <file name="thesis.tex" suffix=".text"/>
        <file name="motivation.tex" suffix=".text"/>
        ...
      </dir>
      ...
    </dir>
  </dir>
  ...
</dir>
```

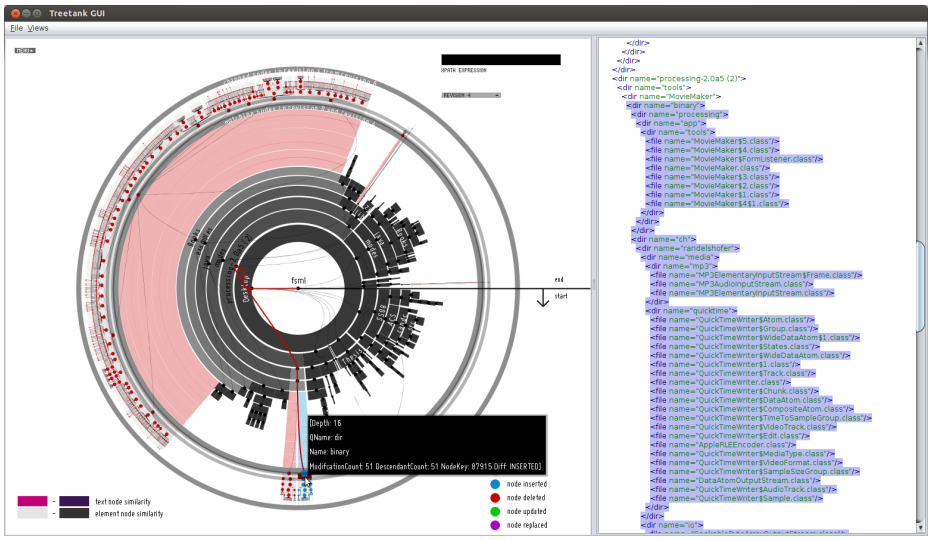

Fig. 41. FSML comparsion of "/home/johannes/Desktop"

```
26 </fsml>
```

While this representation currently does not incorporate most of the strengths FSML usually provides it is easy to add metadata about files and to incorporate text-files. Optionally instances of custom classes are pluggable to provide any type of extensibility. Thus it is possible to provide extractors for certain types of files and to incorporate text-files into the FSML-representation itself, possibly by adding a link to another resource in the fsml-database.

Fig. 41 is an example of mapping a Desktop-folder to a database in Treetank (move- and replace-detection is disabled). Subsequent revisions are commited every five minutes.

Most files and directories are unchanged. Usually changed items (besides the processing-subfolder which has been deleted) are rather small, as the subtrees are rather small compared to most unchanged nodes. Pruning by same hashes instead of the itemsize are displayed in Fig. 42. The changed subtrees are much better visible.

A sequence of revisions can be viewed with the incremental smallmultiple variant (Fig. 43). We used the hash-based pruning without keeping nodes with same hashes, to keep the number of items to a minimum. By hovering the items and compare the values and subtree sizes we are able to determine that between revision 2 and 3 in the bottom right view a subtree has been renamed. A `rename`-operation however is not supported by the Java `WatchService` such that by using an ext3-filesystem we receive a delete and an insert event.

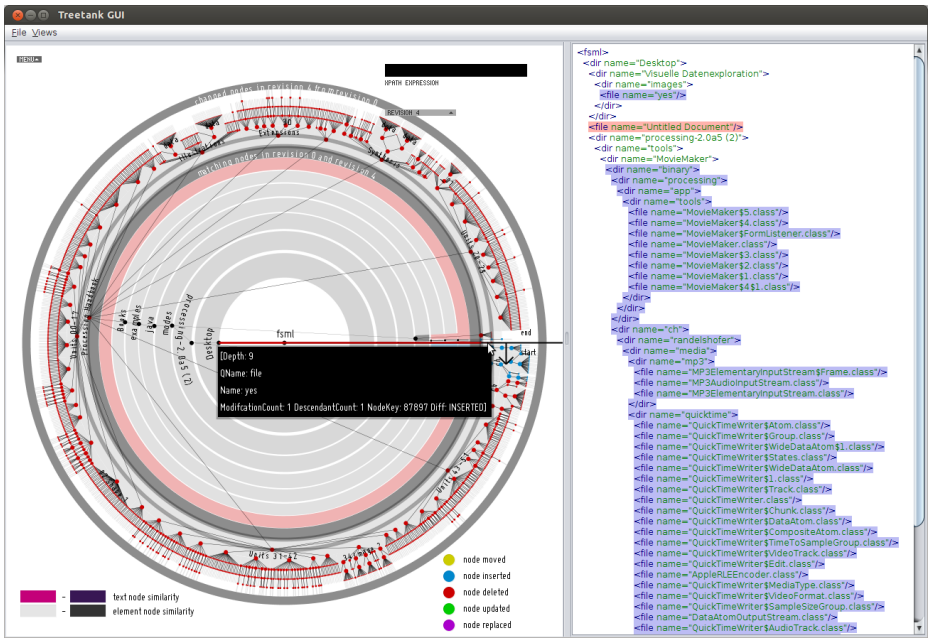

Fig. 42. FSML comparsion of "/home/johannes/Desktop" pruned by same hashes

5.5 Summary

This chapter introduced a few use cases for comparisons of tree-structures. Depending on the activated views different characteristics are revealed.

The smallmultiple differential-variant should be used to compare similar, different tree structures. Choosing the incremental variant reveals changes in temporal evolving trees. In case the tree is rather small (for instance less than 5000 nodes) it is arguable that the views can be used without pruning interesting nodes. However, once more items have to be created it is necessary to choose one of the filtering methods to keep the numer of items small. Hovering through linking and brushing allows to keep track of changed nodes. When enabling pruning inserted nodes might not be visible in upcoming comparisons. However this indicates that the items have not been changed within later revisions.

The standard *SunburstView* reveals all kinds of differences and furthermore allows two zooming operations, the normal zoom which transforms the viewing-coordinates and can be used if the number of items is small and therefore animations are possible whereas a left-mouseclick on an item transforms the item into a new root-node. As already explained in Chapter 4 this involves either a real transformation based on node-copies and adjusting of angles or in some circumstances the algorithm to create the items in the first place must be used.

We have merely scratched the surface of the possibilities of our current approach. Future work might include The SmallMultiple-Views will greatly benefit

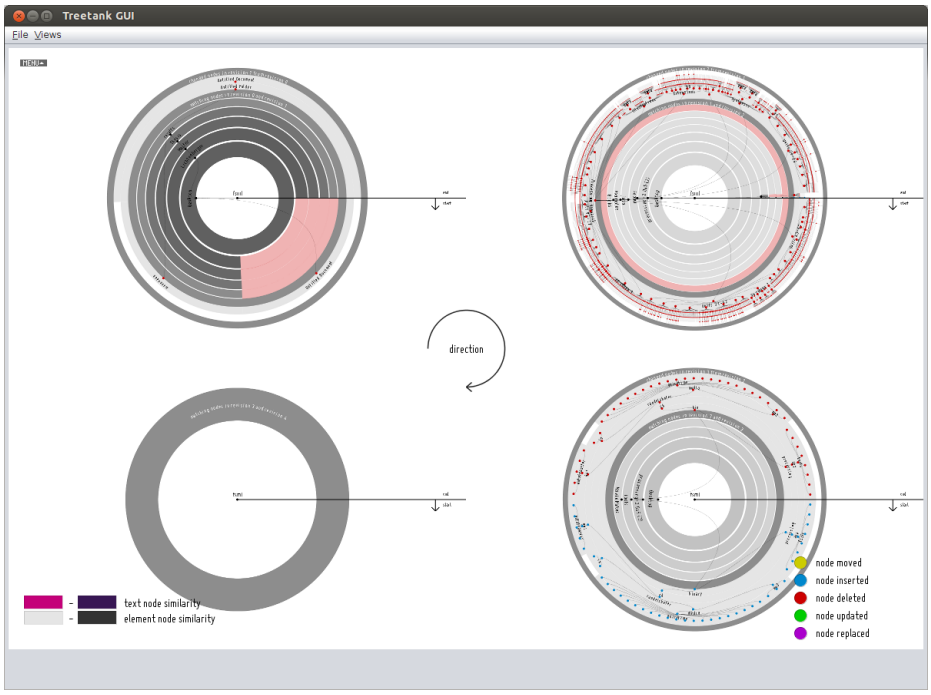

Fig. 43. FSML comparsion of "/home/johannes/Desktop" (SmallMultiples incremental view)

from a temporal XPath-extension. Some proposed axis as `next::`, `previous::`, `future::`, `past::`, `revision::` already have been implemented but have to be incorporated in either the Saxon-Parser or our XPath 2.0-parser. Furthermore adding the possibility to execute temporal XPath-queries and the highlighting of resulting nodes in the SmallMultiple-Views will allow to track specific kinds of nodes over several revisions. The simple query
`//dir[@name='wiki-sorted']/future::node()` will reveal nodes with the attribute `name='wiki-sorted'` in all future revisions. The context will be based on the currently opened base-revision.

6 Discussion

6.1 Introduction

This Chapter concludes the main contribution of this thesis with a discussion of the work presented in relation to the state-of-the-art.

The presented approach facilitates the comparison of tree-structures through various linked visualizations showing the tree-structures at varuous levels of detail. The Small multiple variants of a *SunburstView* provide a high level view of the differences between many tree-structures. Linking and brushing is used to highlight nodes in all small visualizations. The *SunburstView* itself incorporates both a mode to visualize one tree-structure and a mode to visualize comparisons between two tree-structures. A new layout-algorithm has been developed to highlight changes in a prominent place. Furthermore different filtering methods allow the comparison of large trees. Details are shown on demand through a mouseover effect and a linked `TextView`. A selection of a new root-node togehter with a simple undo-operation allows to drill down into the tree as known from other Sunburst-visualizations. However costly recalculation of diff-tuples, modifications and descendants of each node is omitted in all cases but the itemsize-based pruning. Instead we recalculate the maximum depth of unchanged nodes and scale a new list of items to fit the new boundaries. The `TextView` provides a low-level serialization of (sub)trees linked with the `SunburstView` also tailored to large tree-structures.

Unlike other visualizations of tree comparisons we have found and described in Chapter 2 our approach is entirely database driven. The DBMS is tailored to temporal tree-structures which are stored as snapshots. Each node in the database/resource is unique and remains stable throughout all revisions. As most trees do not inhibit node-IDs we have implemented an ID-less diff-algorithm called FMSE to import differences. Afterwards we are able to use a faster diff-algorithm based on the node-IDs and furthermore are able to utilize hashes to further speed up the algorithm. Due to FMSE our similarity-measure between trees is based on a matching algorithm which computes the longest common subsequence (LCS) bottom up and furthermore tries to find cross-matches of unmatched nodes.

Most proposed visualizations in contrast rely on stable unique IDs (the Contrast Treemap[16], Code Flows[19], TreeJuxtaposer[18] (unique node labels)) and do not include the detection of `replace`- and `move`-operations. However Code Flows visualizes moves due to spline connections between matching nodes with the same ID in two adjacent icicle plots. Due to the reliability on unique node IDs these visualizations do not cover trees which do not include unique node-IDs. In contrast our prototype is able to determine and visualize these tree-structures utilizing the FMSE-algorithm to import differences through similarity metrics for leaf- and inner-nodes.

The next section evaluates our approach similar to Chapter 2 according to several attributes.

6.2 Evaluation Criteria

- *SpaceFilling* Space filling techniques try to maximize the usage of available screen space and thus facilitate a higher information density. Sunburst layouts are generally space filling, but in contrast to Treemaps lack space filling properties in the corners. However in our case the corners are used to display GUI components and legends. The small multiple variants currently have too much unused space which is due to the often times non squarified screen viewport depending on the extends of the main window. As we just scale each visualization of the comparison of each two revisions the small multiple variants currently include too much unused space due to often times non squarified screen viewport which depends (1) on the size of the main window and (2) how many visualizations are currently enabled. Thus, in a future version we will use a squarified clip of the view depending on the maximum depth to generate minimized offscreen buffers for the four regions. Other space filling approaches include Icicle plots (Code Flows) which are comparable in space consumption as always the two complete tree-structures to compare are plottet. In our case unchanged nodes are not plotted twice. Furthermore a special stable Spiral-Treemap layout has been proposed which utilizes all available screen space and remains relatively stable after changes. Usual Treemap layouts suffer from abrupt layout changes during modifications of the underlying data.
- *Hierarchy* The Hierarchy in Sunburst layouts is very well depicted due to the adjacency based layout whereas it is not as obvious in Treemaps which encapsulate child items. Cushion Treemaps have been developed for better readability of hierarchical relationships using shading. However they are still not as good readable as in adjacency based layouts. Code flows utilizes Icicle plots which are rectangular views of the radial Sunburst layout, thus adjacency based and very well readable. Our approach optionally plots a node-link diagram on top of the Sunburst layout to further illustrate the relationships such that the hierarchy is at least as well depicted as in other node-link diagrams (for instance in TreeJuxtaposer and Treevolution).
- *Readability* To both use node-link diagrams and a Sunburst layout facilitates a higher information density as individual nodes can be color-coded as well as their child/parent relationship, the links between the nodes. Thus we are able to utilize to map certain attributes to the extension of the Sunburst items, the color of the arcs, the color of the dots/nodes in the node-link diagram. To maximize the information density we could also use histograms in the Sunburst items instead of just using one color for the whole item. However we assume that usually the items will be too small, such that a whole histogram would possibly not be readable in many cases. To support better readability of item-labels our visualization is able to be rotated. Node labels of Treemaps and Icicle plots however usually are better readable due to their rectangular display in comparison to circular plotted labels. Thus the *TextView* has been extended to take the agglomerated tree-structure into account. It is an ideal partner of the *SunburstView* as it provides better readability of small

subtrees but lacks the overall overview about all differences provided by the *SunburstView*.

- *Similarity of ID-less tree-structures* Some proposed tree-to-tree comparsion visualizations depend on node-IDs (Spiral-/Contrast-Treemap[16]) or the comparison technique has not been meantioned. Others depend on domain characteristics (Treejuxtaposer[18], Code Flows[19]). Juxtaposer seems to rely on unique leaf node labels. Otherwise it is not obvious how to map node labels to their postorder rank on a region plane. Furthermore phylogenetic trees are leaf labeled, whereas we also consider labeled inner nodes. In contrast to Treejuxtaposer our prototype is able to compare every kind of tree-structure.
- *Structural changes* Our visualizations, in particular the *SunburstView* and the small multiple variants support the highlighting of all kind of structural-changes (inserts, deletes, updates, replace and move-operations) through color-coded nodes. In case of moves- links, currently depicted as arrows, denote the movement from their original- to their target-place using hierarchical edge bundling to avoid or at least reduce visual clutter due to overlapping lines or curves. Furthermore the extend of the sunburst items is based on the nodes' subtree size *and* the number of modifications in the subtree. Most other visualizations do provide a global distortion to further emphasize changes except Treejuxtaposer to the best of our knowledge.
- *Non structural changes* in `TextNode`s are color coded to denote that the value is `UPDATED` through coloring the node in the overlapping node-link diagram accordingly. Furthermore the color of the SunburstItem reflects their Levenshtein String-similarity.
- *Filtering* is one of our primariy concerns. The focus of this thesis evolved around comparing Treetank-resources. As Treetank is a secure storage system it naturally often times stores very large tree-structures. Thus our first consideration was to filter items which can be perceived, that is too small items are not created. However this only effects the creation of SunburstItems and does not concern the ID-based diff-algorithm. To speed up both, the ID-based diff algorithm as well as the construction of SunburstItems we developed the diff-algorithm which skips subtrees with same hashes and moves both transactions to the next node on the XPath-following axis. Thus, in case large subtrees can be skipped the diff-algorithm using hash-comparisons is much faster. Besides even less items have to be created and in case of a subtree-selection only the itemsizes have to be recalculated instead of reinvoking the diff-algorithm, recalculate the descendant- and modification-count, and the whole preorder traversal of the agglomerated tree-structure with all stack-adjustements. Pushing the idea of filtering by hashvalues to its limits involves to avoid the creation of SunburstItems for the nodes, which have the same hashvalues. These filtering techniques furthermore enlarge changed subtrees naturally as the extend of the items is based on the *descendant-count* of each node and the *modification-count*. Note that the *descendant-count* includes the node itself, that is all nodes in the subtree plus one. TreeJuxtaposer once more along with Code Flows to the best of our

knowledge seem to be the only systems which are able to handle large tree-structures but the diff-algorithm of TreeJuxtaposer relies on unique node labels. Code Flows does not provide a global filtering method such that their filtering is comparable with our method to drill down into the tree. However, due to filtering by queries the filtering is mightier. Implementing such a behavior in our prototype will be straight forward allowing the user to specify an XPath-query and invoke the diff-algorithm for each result on both revisions. Our current approach is able to filter differences and related context nodes on a global basis and on subtrees once a user drilled down into the Sunburst visualization with a guaranteed visibility of modifications much like in TreeJuxtaposer.

Nochmal Tabelle mit Vergleich zu allen anderen im related work kapitel beschriebenen visualisierungen

7 Summary, Conclusion and Future Research

7.1 Summary

A full pipeline has been implemented backed by the secure tree-storage system Treetank ranging from (extensive) preprocessing to ID-less and ID-based diff-algorithms to new layout algorithms developed for

Almost all data is subject to preprocessing. In the case of Wikipedia we aimed at comparing the temporal evolution of several articles. The Wikimedia Foundation provides only full Wikipedia dumps which aren't sorted by revision-timestamps. Therefore it had to be sorted at first using Hadoop. Importing structural changes which should be revealed by the visualizations afterwards requires a diff-algorithm which doesn't operate on unique node IDs. Thus the FMSE-algorithm described in Chapter 2 and 3 has been implemented as well as several missing edit-operations in Treetank (`copy`, `move`, `replace`), whereas others have been extended to adhere to the XPath/XQuery Data Model (XDM) to simplify the internal ID-based diff-algorithm and to add to the visualizations' expressiveness (no two adjacent text-nodes are ever created). The FMSE-algorithm is furthermore useful to compare generic trees. In this case structural changes between a base tree and several other trees are imported consecutively which are visualized by the SmallMultiples differential view in a subsequent step. Other studied applications include the comparison of snapshots of a directory in a filesystem and two revisions of XML-exported LFG-grammars.

An ID-based diff-algorithm has been developed to compare imported data based on snapshots/revisions, whereas the comparison is *not* restricted to both subsequent revisions and whole documents. A fast version utilizes hash-values of each node generated during the import. These are derived from the nodes' descendants and used to reduce the run-time of the algorithm. The algorithm can skip the comparison of the whole subtree of every node with a matching hash-value.

The visualizations trigger the ID-based diff-algorithm if needed. A *SunburstView* based on aggregated tree-structures is one of the major contributions. Drawing a new Sunburst-layout for a selected-node to drill down into the tree doesn't require the invocation of the diff-algorithm. A notable exception is the itemsize-based pruning because the diffs are usually not saved to avoid their constant in-memory space consumption. Instead they are usually subject to garbage collection. The type of diff however is saved as part of the created Sunburst item and used by the visualization.

A new Sunburst layout algorithm has been designed to specifically support the analysis of structural changes, which are featured prominently between two rings whereas the first inner ring denotes the maximum depth of unchanged nodes and the second denotes the maximum depth of changed nodes. The first changed node in a subtree is semantically zoomed from its original place (the depth is increased) to reside between the two rings supporting a tree-ring metapher based on the analogy of an aging tree in the nature which adds growth rings every year.

The variant of the *SunburstView* which has been designed to compare tree-structures furthermore incorporates three filtering methods. First, an itemsize based approach to filter nodes based on their extend is provided. It is used to speed up the creation of the Sunburst items, but does not effect the run-time of the diff-algorithm used in the first place. Second, a method utilizing the hash-based diff-variant skips whole subtrees of unchanged nodes and thus builds solely interesting items to the task at hand (which is analysing structural changes). A third variant even skips the creation of building items for nodes with unchanged content altogether. This variant is especially interesting for very large tree-structures to keep the memory consumption to a minimum and to provide maximum space for subtrees which contain changed nodes.

Besides filtering uninteresting nodes automatically reducing the run-time of the diff-algorithm and/or the creation of Sunburst items and reducing in-memory space consumption sometimes it is interesting to view the whole aggregated tree in an overview. To still emphasize structural changes the extend of an item is based on the `descendant-or-self` count of a node as well as of its modification count, whereas the amount which one affects the extend at most can be chosen by a slider. Note that the modification-count is based on the number of modification in the nodes' subtree as well as on a constant factor the which is multiplied as modifications are usually small compared to the whole tree plus the addition of the `descendant-or-self` count as a fallback which needed in case of zero modifications in the subtree. The addition is needed in case of the slider to adjust how much the modifications contribute to the extension is set to zero, such that the extend is solely based on the `descendant-or-self` count.

In addition to the enhanced `SunburstView` supporting comparisons several *SmallMultipleView* variants, based on the *SunburstView* have been developed to support an overview about the changes ranging up to the comparsion of currently at most five revisions. An incremental version displays changes related to a sliding window of two subsequent revisions. The left upper view shows a comparison between the loaded baseRevision and $baseRevision + 1$, the right upper view displays the comparision between $baseRevision + 1$ and $baseRevision + 2$ whereas the comparsion between two subsequent revisions is done clockwise until either no more revisions are available or five revisions have been compared (all four squarified regions are filled with Sunburst views). A differential version compares subsequent revisions to the loaded baseRevision. This version is ideal to compare different tree-structures to a reference tree. A hybrid view first compares the baseRevision with the last revision which can be drawn in the bottom left square of the visualization. Subsequently it computes differences just like in the incremental version but blackens changes which occur in subsequent incremental comparisons.

7.2 Conclusion

In retrospective the goals described in the introduction have been achieved. The GUI-frontend embedding the visualizations aids analysts in comparing tree-structures ranging from temporal to generic similar trees. Its main strength is

the comparison of structural changes, whereas a smilarity-measure for `element`-nodes depends on the amount of overlapping nodes in its subtree. In contrast `text`-nodes are compared based on the Levenshtein-algorithm as `text`-nodes are always leaf-nodes in a tree. Other algorithms might be easily included for instance to compare numerical values.

Our approach based on a tight storage integration integrates several edit-operations utilizing an expressive aggregated tree-structure build through observing diffs from the ID-based diff algorithm. The ID-based algorithm supports two modes. Either only structural nodes (`DocumentNode`, `ElementNode` and `TextNode`s) are compared or structural and non structural nodes (adding `NamespaceNode`s and `AttributeNode`s). The encountered diffs are saved in an associative array and further processed to construct a novel Sunburst layout tailored to tree-comparisons. In contrast to other visual tree-comparison tools our prototype incorporates several edit-operations including moved and replaced nodes which are rarely seen in other visualizations. Filtering-mechanisms facilitate the comparison of very large tree-structures. Note that no other approach described in Chapter 2 to the best of our knowledge incorporates such filters and therefore most likely is not useful to inspect large datasets. Two filtering mechanisms depend on hashes which are used to compare the whole subtree of a node including itself. Subtrees are skipped from comparison if the hashes are identical. Thus they are neither traversed nor items are build subsequently. Besides building an aggregated tree-structure from observing changes by the ID-based diff-algorithm no state is involved. Unlike the straight forward approach of building two in memory datastructures with nodes of both revisions which are going to be compared our algorithm uses less space and is much faster. Furthermore a new Guava based cache preloads node pages in a dedicated thread and thus facilitates faster traversal of large tree-structures.

In addition to a novel Sunburst layout we implemented three different *SmallMultiplesView* variants. Two of them, the incremental- and the differential-view compute the diffs and the subsequent visualizations in each of the four screen regions (top-left, top-right, bottom-right, bottom-left assuming five subsequent revisions of the tree-structure to compare exist) in parallel. On mouse-over items are highlighted in each region in which they are present. Compared to an icicle-view which connects unchanged items in each revision by splines our approach features changes much more prominently through the semantic zoom/the tree-ring metapher as well as a modifyable modification-rate. A hybrid variant currently suffers from the lack of visual

7.3 Future Research

Many topics are subject to further research.

- First of all we want to incorporate the already developed temporal XPath axis in an XQuery processor, most probably Brackit[25], which is especially important to further analyse temporal evolving tree-structures. Therefore it furthermore will be inevitable to provide indexes for fast response-times of the visualizations.

- Extend the Hierarchical Edge Bundles to use a gradient color to indicate the direction of moves instead of arrows.
- Building an index structure for visualizations of consecutive revisions (the SmallMultiple incremental-variant and the SunburstView comparing consecutive revisions).
- Evaluation and integration/implementation of various other tree-similarity measures.
- Support queries to select interesting regions before invoking the ID-based diff-algorithm.

A Treetank

A.1 Persistent storage enhancements

While not exclusively developed for our tree-to-tree comparsion for completeness we want to meantion several techniques which have recently been developed to support the persistent storage of large tree-structures efficiently (with a minimum space overhead).

- Values of `TextNodes` are compressed by using the Deflate-algorithm which combines the LZ77 algorithm and Huffman coding. Decompressed values are cached in memory once they are requested. Furthermore only values which are greater than a certain length-threshold are compressed.
- Pointers to neighbour nodes, the first child and the parent node are persisted as ranges.
- As ranges are persisted node-IDs are efficiently compressed.
- Whole `NodePage`s are compressed using the very fast Snappy-algorithms due to a lot of compression/ decompression in case of many modifications of the same NodePage in the BerkeleyDB log.
- In case of the `DocumentNode` only the firstChild-pointer and descendantcount has to be persisted. Similar the serialization of `TextNode`s doesn't include the childCount and descendantCount as well as the firstChild-pointer.
- The `NamePage` which is used to store String-names which are commonly repeated in XML-documents now contains index-mappings whereas we opted for different indexes based on an element-QName index, an attribute-QName index and a namespace/URI-index. This allows an XPath- or XQuery- Optimizer to use the index for queries as `count(//@attribute='foo')` or `count(//element()='bar')`. We furthermore aim to provide backreferences to the nodes as we encountered far too long response-times on larger tree-structures. A path-like index therefore might be inevitable in the long term.
- Deletion of `attribute`- and `namespace`-nodes in subtrees.

A.2 ACID properties

Consistency rules have been enhanced while developing our prototype. The following provides a brief overview about the ACID-properties of Treetank.

1. *Atomicity* is ensured through the transaction layer and the interchangeable backend (currently BerkeleyDB). As such atomicity is even guaranteed in case of power failures, errors and crashes.
2. *Consistency* as of now involves the checking of `QName`s for validity. At all times no adjacent text nodes are created which is consistent with the XPath/XQuery Data Model (XDM). We also introduced XML entity encoding for the XML characters serialization (in the `XMLSerializer` as well as the new `StAXSerializer` and `SAXSerializer`). Furthermore we check attribute `QName`s for duplicates and throw an appropriate runtime exception if a new

attribute insertion would yield duplicates. Similarly insertion of duplicate namespace prefixes for namespaces of the same parent element-node are prohibited.

3. *Isolation* is guaranteed through *Snapshot-Isolation* which is based on the transaction-, page- and I/O-layer through versioning. Furthermore currently only one write transaction is allowed per resource. To maximize the properties of tree-structures the implementation of concurrent write-transactions on different subtrees with appropriate locking is in development.
4. *Durability* is guaranteed through the backend. The transaction log created by Treetank's BerkeleyDB-binding implemented as a cache to store all changed pages and nodes is written and flushed on transaction commit.

A.3 Axis

The axis in Treetank have been changed to adhere to the `hasNext()` and `next()` specifications of the `Iterable` interface. The check if `getNext()` is true is added to all axis such that `hasNext()` is idempotent. It simply checks a flag which is set in `resetToLastKey()` to make sure the transaction points to the node after the last call to `hasNext()` without changing the node-ID to which to move in the next call to `next()`. Furthermore the transaction now is not moved forward in `hasNext()` anymore which is done only when calling `next()`. Instead a variable denoting the next node-ID is set which is used by the `next()` implementation to move to the next node. Furthermore `next()` now also is idempotent, simply checking if it has been called before. When true and `hasNext()` has not been called immediately before it is first called by `next()`.

Levelorder-Axis The `LevelOrderAxis` is described in algorithm 9. Just like other axis to traverse certain regions or the whole tree-structure in Treetank it is based on the `Iterator/Iterable` Java interfaces to support the `foreach`-loop and iterator-based iteration. All other axis have been extended as described in appendixA. *mFirstChilds* is a double ended queue to remember all first childs of each node for a subsequent new depth ($depth+1$). `processElement()` is invoked to add non structural nodes, that is `attributes` and `namespaces` to the queue. After initialization the queue is empty and *mNextKey* is initialized to either the current key (if self is included), the right sibling node key if there is one or the first child node key. The NULL_NODE_KEY is a special node key to denote that the traversal is done.

A.4 Edit operations

The `copy`-operation adds the capability to add whole subtrees of another resource or revision to the currently opened *resource/revision*. Actually three `copy`-operations exist. Either the subtree is inserted as a `firstChild`, `rightSibling` or `leftSibling` of the currently selected node. The node to copy must be a

Algorithm 9: LevelOrderAxis (hasNext())

input : boolean mFirst, Deque mFirstChilds, long mKey
output: node key of next node

```
1  if getNext() then
2      return true;
3  resetToLastKey();
4  // Setup.
5  INodeReadTrx rtx ← getTransaction();
6  IStructNode node ← rtx.getStructuralNode();
7  // Determines if it is the first call to hasNext().
8  if mFirst == true then
9      mFirst ← false;
10     return processFirstCall();
11 // Follow right sibling if there is one.
12 if node.hasRightSibling() then
13     processElement();
14     // Add first child to queue.
15     if node.hasFirstChild() then
16         mFirstChilds.add(node.getFirstChildKey());
17     mKey ← node.getRightSiblingKey();
18     return true;
19 // Iterate over non structural nodes (attributes/namespaces).
20 if mInclude == EInclude.NONSTRUCTURAL then
21     processElement();
22 // Add first child to queue.
23 if node.hasFirstChild() then
24     mFirstChilds.add(node.getFirstChildKey());
25 // Then follow first child on stack.
26 if !mFirstChilds.isEmpty() then
27     mKey ← mFirstChilds.removeFirst();
28     return true;
29 // Then follow first child if there is one.
30 if node.hasFirstChild() then
31     mKey ← node.getFirstChildKey();
32     return true;
33 // Then end.
34 resetToStartKey();
35 return false;
```

structural node, that is either an `ElementNode` or a `TextNode`. In case the transaction is located at a `DocumentRootNode` which is a special document node, which can not be deleted and exists in every revision the read transaction has to move to the first child in the first place.

Algorithm 10 describes the handling of `TextNode`s. However, more interesting are `ElementNode`s (algorithm 13). The algorithm recursively calls itself (`mRtx.getNode().acceptVisitor(this)`) to copy the whole subtree. To ensure that only subtrees are copied and no other nodes in document order, the depth starting at zero must be at all times > 0 except for the root of the subtree to insert.

Algorithm 10: visitNode(TextNode pNode))

input : INodeReadTrx mRtx, EInsert mInsert, TextNode pNode
output: void (none)

```
1 mRtx.moveTo(pNode.getNodeKey());
2 mInsert.insertNode(mWtx, mRtx);
3 if !mFirst and mRtx.getStructuralNode().hasRightSibling() then
4 |   mRtx.moveToRightSibling();
5 |   mInsert ← EInsert.ASRIGHTSIBLING;
6 |   mRtx.getNode().acceptVisitor(this);
7 else if !mFirst then
8 |   insertNextNode();
```

The `move-` operation just like the `copy-` operation is implemented in three different versions, `moveSubtreeToFirstChild(long)`, `moveSubtreeToRightSibling(long)` and `moveSubtreeToLeftSibling(long)`. The third operation is a simple wrapper for the other two, depending on the fact if a left sibling of the current node exists or not. The only parameter is the unique node ID of a node to move along with its descendants. Adjacent text-node merging adds a lot of complexity which affects the childCount and descendantCount of both the old parent node before moving a subtree and the new parent after moving. The move-operations preserve the unique node-ID and therefore are not atomic wrappers for delete- and subsequent insert-operations or vice versa.

We examine both operations independently as they require different node manipulations of neighbour nodes including the new and old parent node and the first child node. Both operations are defined in terms of moving structural- and non-structural nodes (the latter in subtrees of the moved node), whereas the root-node to move must be a structural node.

First, the following constraints are checked:

- The node/subtree to move must be an `element-` or a `text`-node.

Algorithm 11: visitNode(ElementNode pNode))

input : INodeReadTrx mRtx, EInsert mInsert, int mDepth, ElementNode pNode
output: void (none)

```
1  mRtx.moveTo(pNode.getNodeKey());
2  mInsert.insertNode(mWtx, mRtx);
3  mInsert ← EInsert.ASNONSTRUCTURAL;
4  for int i ← 0, nspCount ← pNode.getNamespaceCount(); i < nspCount; i++ do
5      mRtx.moveToNamespace(i);
6      mInsert.insertNode(mWtx, mRtx);
7      mRtx.moveToParent();
8  for int i ← 0, attrCount ← pNode.getAttributeCount(); i < attrCount; i++ do
9      mRtx.moveToAttribute(i);
10     mInsert.insertNode(mWtx, mRtx);
11     mRtx.moveToParent();
12 if pNode.hasFirstChild() then
13     mFirst ← false;
14     mInsert ← EInsert.ASFIRSTCHILD;
15     mRtx.moveToFirstChild();
16     mDepth+=1;
17     mRtx.getNode().acceptVisitor(this);
18 else if !mFirst and paramNode.hasRightSibling() then
19     mInsert ← EInsert.ASRIGHTSIBLING;
20     mRtx.moveToRightSibling();
21     mRtx.getNode().acceptVisitor(this);
22 else if !mFirst then
23     insertNextNode();
```

Algorithm 12: insertNextNode()

input : INodeReadTrx mRtx, INodeWriteTrx mWtx, EInsert mInsert
output: void (none)

```
1  while !mRtx.getStructuralNode().hasRightSibling() and mDepth > 0 do
2      mRtx.moveToParent();
3      mWtx.moveToParent();
4      mDepth-=1;
5  if Depth > 0 then
6      mInsert ← EInsert.ASRIGHTSIBLING;
7      if mRtx.getStructuralNode().hasRightSibling() then
8          mRtx.moveToRightSibling();
9          // Recursion.
10         mRtx.getNode().acceptVisitor(this);
```

- It is not permitted to move a node to one of its descendants which would introduce , therefore it is checked if the node to move is one of the ancestors of the anchor node.

The two operation require the following node manipulations. The location the node is moved away is subject to the following changes:

- In case of `moveSubtreeToFirstChild(long)`: Parent node must point to the right sibling node key of the current node if it was the first child.
- Decrement child count of parent node.
- Collapse text nodes if both the left- and right- sibling are text nodes, therefore remove the right node and append its value to the left sibling. Furthermore adapt its right sibling pointer to the former right sibling of the right sibling text node which has been deleted.
- If no text node collapsing involved: adapt right sibling key of former left sibling to the right sibling key of the moved node provided one exists.
- If no text node collapsing involved: adapt left sibling key of former right sibling to the left sibling key of the moved node provided one exists.

The node to which the new subtree is inserted is subject to the following changes.
In case of `moveSubtreeToFirstChild(long)`:

- Increment the child counter of the node to which the new subtree is inserted.
- Adapt the first child key of the node to which this node is inserted to point to the new node.
- Adapt the left sibling pointer of the former first child to the root node of the inserted subtree.

In case of `moveSubtreeToRightSibling(long)`:

- Increment the child counter of the parent node.
- Adapt the right sibling key to the nodeKey of the moved node.

The node which is moved is changed in the following ways.
In case of `moveSubtreeToFirstChild(long)`:

- Adapt the left sibling pointer to `NULL_NODE_KEY`.
- Adapt the right sibling pointer to the former first child of the node to which this node is inserted or `NULL_NODE_KEY` if there was not one.
- Adapt the parent pointer to point to the node to which this node is inserted.

In case of `moveSubtreeToRightSibling(long)`:

- Adapt the left sibling pointer to the nodeKey of the node where this node is inserted.
- Adapt the right sibling pointer to the right sibling key of the node where this node is inserted.

The implementation of the replace-operation is straigth forward using a delete- followed by an insert-operation chaining the two to provide an atomic operation.

Besides, the `remove`-operation has been modified to merge adjacent `TextNode`s if the deleted node has two neighbour text nodes to adhere to the XPath/XQuery Data Model (XDM) and to provide meaningful visualizations. `insert`-operations as of now check if the new `TextNode` would yield two consecutive `TextNode`s and, if so, update the value of the existing node with a concatenation of the old- and new-values instead. Therefore the storage consistently avoids consecutive `TextNode`s.

Furthermore a new bulk-insertion method supports the fast insertion of subtrees based on a component which already existed. Hashes are computed in a subsequent postorder traversal of the inserted subtree thus reducing the runtime from $O(n^2)$ to $O(n)$.

A.5 Visitor implementation

The axis executes a visitor specific implementation for each visited node. The return value of the methods the visitor has to implement (a visitor specific implementation for each node-type) guides the traversal in the axis. The following result types are currently available:

- `EVisitResult.TERMINATE`, terminates the traversal of the (sub)tree immediately and returns false for upcoming `hasNext()` calls.
- `EVisitResult.CONTINUE`, continues preorder traversal.
- `EVisitResult.SKIPSUBTREE`, signales that the axis skips traversing the subtree of the current node.
- `EVisitResult.SKIPSIBLINGS`, signales that the axis should move to the next following node in document order which is not a right-sibling of the current node.
- `EVisitResult.POPSKIPSIBLINGS`, is a special type which signals that the element on top of the internal right sibling stack must be popped, which is needed to implement for instance the deletion-visitor for the second FMSE-step.

An implementation must implement a method `EVisitResult visiNode(NodeType pNode)` for each node type. The deletion-visitor implementation of the method `EVisitResult visitNode(ElementNode pNode)` is described in algorithm 13. Note that all `attribute`- and `namespace`-nodes, which are going to be deleted, must be temporally saved in a sequence. Afterwards they have to be deleted through a sequence traversal. If the nodes instead are deleted in place the `attribute`- and/or `namespace`-counter of the parent `element`-node is decreased immediately and possibly unmatched nodes with the highest index(es) are not going to be deleted. The `delete(INode)` method described in 14 deletes the element node along with all its `attribute`- and `namespace`-nodes as well as its subtree. During the removal of a structural node (`element`- or `text`- node) and

its subtree the transaction is either moved to the right sibling of the deleted node, to the left sibling or to the parent, if it exists in the order described. Note that the parent must exist. Treetank moves the transaction either to the right-sibling of the current node to delete if it exists, to the left-sibling if it exists or to the parent node in the given order. For the simple reason that the transaction is moved to the next node in preorder after invoking the visitor which actually deletes a node, the transaction must be moved to the last node in the `previous::` axis. Otherwise it will be moved by the remove-operation in the first place *and* the axis subsequently. The `VisitorDescendantAxis` moves the transaction to the next node in preorder afterwards. The movement is determined and executed after the deletion of the `element`-node in the method `delete(pWtx, pNode)` outlined in algorithm 14.

Algorithm 13: FMSEDeleteVisitor: EVisitResult visit(ElementNode pNode)

```
   input : WriteTransaction mWtx, Matching mMatching, long mStartKey,
           ElementNode pNode
   output: EVisitResult type
 1 Long partner ← mMatching.partner(pNode.getNodeKey());
 2 if partner == null then
 3     EVisitResult retVal ← delete(mWtx, pNode);
 4     if pNode.getNodeKey() == mStartKey then
 5         retVal = EVisitResult.TERMINATE;
 6     return retVal;
 7 else
 8     long nodeKey ← pNode.getNodeKey();
 9     List<Long> keysToDelete ← new ArrayList<>(pNode.getAttributeCount()
       + pNode.getNamespaceCount());
10     for int i = 0; i < pNode.getAttributeCount(); i++ do
11         mWtx.moveToAttribute(i);
12         long attNodeKey ← mWtx.getNode().getNodeKey();
13         if mMatching.partner(attNodeKey) == null then
14             keysToDelete.add(attNodeKey);
15         mWtx.moveTo(nodeKey);
16     for int i = 0; i < pNode.getNamespaceCount(); i++ do
17         mWtx.moveToNamespace(i);
18         long namespNodeKey ← mWtx.getNode().getNodeKey();
19         if mMatching.partner(namespNodeKey) == null then
20             keysToDelete.add(namespNodeKey);
21         mWtx.moveTo(nodeKey);
22     foreach long keyToDelete : keysToDelete do
23         mWtx.moveTo(keyToDelete);
24         mWtx.remove();
25     mWtx.moveTo(nodeKey);
26     return EVisitResult.CONTINUE;
```

Algorithm 14: FMSEDeleteVisitor: delete(pWtx, pNode)

```
   input  : NodeWriteTrx pWtx, ElementNode pNode
   output: EVisitResult type
 1 long nodeKey ← pWtx.getNode().getNodeKey();
 2 // Case: Has no right and no left sibl. but the parent has a right
      sibl.
 3 pWtx.moveToParent();
 4 IStructNode node = pWtx.getStructuralNode();
 5 if node.getChildCount() == 1 AND node.hasRightSibling() then
 6 |   pWtx.moveTo(nodeKey);
 7 |   pWtx.remove();
 8 |   return EVisitResult.SKIPSUBTREEPOPSTACK;
 9 pWtx.moveTo(nodeKey);
10 // Case: Has left sibl. but no right sibl.
11 if !pWtx.getStructuralNode().hasRightSibling() AND
   pWtx.getStructuralNode().hasLeftSibling() then
12 |   pWtx.remove();
13 |   return EVisitResult.CONTINUE;
14 // Case: Has right sibl. and left sibl.
15 if pWtx.getStructuralNode().hasRightSibling() AND
   pWtx.getStructuralNode().hasLeftSibling() then
16 |   boolean removeTextNode ← checkIfTextNodeRemove();
17 |   if removeTextNode then
18 |   |   pWtx.remove();
19 |   |   return EVisitResult.CONTINUE;
20 |   else
21 |   |   pWtx.remove();
22 |   |   pWtx.moveToLeftSibling();
23 |   |   return EVisitResult.SKIPSUBTREE;
24 // Case: Has right sibl. but no left sibl.
25 if pWtx.getStructuralNode().hasRightSibling() AND
   !pWtx.getStructuralNode().hasLeftSibling() then
26 |   // similar to above case (omitted)
27 // Case: Has no right and no left sibl.
28 mWtx.remove();
29 return EVisitResult.CONTINUE;
```

B Acknowledgements

I would like to thank Sebastian Graf for his guidence as well as .

FERTIG MACHEN

References

[1] D. Keim, G. Andrienko, J. Fekete, C. Görg, J. Kohlhammer, and G. Melançon, "Visual analytics: Definition, process, and challenges," *Information Visualization*, pp. 154–175, 2008.

[2] "Moore's law." [Online]. Available: http://en.wikipedia.org/wiki/Moore's_law

[3] J. J. Thomas and K. Cook, "Illuminating the path: The r&d agenda for visual analytics." [Online]. Available: http://nvac.pnl.gov/agenda.stm

[4] F. J. Anscombe, "Graphs in statistical analysis." [Online]. Available: http://www.jstor.org/discover/10.2307/2682899?uid=3737864&uid=2&uid=4&sid=56171835643

[5] "Xmark benchmark." [Online]. Available: http://www.xml-benchmark.org/

[6] S. Rönnau, G. Philipp, and U. Borghoff, "Efficient change control of xml documents," in *DocEng*, vol. 9, 2009, pp. 3–12.

[7] "Delta xml." [Online]. Available: http://www.deltaxml.com

[8] G. Cobena, S. Abiteboul, and A. Marian, "Detecting changes in xml documents," in *Data Engineering, 2002. Proceedings. 18th International Conference on.* IEEE, 2002, pp. 41–52.

[9] K. Tai, "The tree-to-tree correction problem," *Journal of the ACM (JACM)*, vol. 26, no. 3, pp. 422–433, 1979.

[10] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *ACM SIGMOD Record*, vol. 25, no. 2. ACM, 1996, pp. 493–504.

[11] T. Lindholm, J. Kangasharju, and S. Tarkoma, "Fast and simple xml tree differencing by sequence alignment," in *DocEng*, vol. 6, 2006, pp. 75–84.

[12] Y. Wang, D. DeWitt, and J. Cai, "X-diff: An effective change detection algorithm for xml documents," in *Data Engineering, 2003. Proceedings. 19th International Conference on.* Ieee, 2003, pp. 519–530.

[13] G. Cobéna, T. Abdessalem, and Y. Hinnach, "A comparative study for xml change detection," *Research Report, INRIA Rocquencourt, France*, 2002.

[14] R. Therón, "Hierarchical-temporal data visualization using a tree-ring metaphor," in *Smart Graphics*. Springer, 2006, pp. 70–81.

[15] S. Bremm, T. von Landesberger, M. Heß, T. Schreck, P. Weil, and K. Hamacherk, "Interactive visual comparison of multiple trees," in *Visual Analytics Science and Technology (VAST), 2011 IEEE Conference on*. IEEE, 2011, pp. 31–40.

[16] Y. Tu and H. Shen, "Visualizing changes of hierarchical data using treemaps," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 6, pp. 1286–1293, 2007.

[17] M. Bruls, K. Huizing, and J. Van Wijk, "Squarified treemaps," in *Proceedings of the joint Eurographics and IEEE TCVG Symposium on Visualization*. Citeseer, 2000, pp. 33–42.

[18] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Z. Zhou, "Treejuxtaposer: scalable tree comparison using focus+context with guaranteed visibility," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 453–462, 2003.

[19] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," in *Computer Graphics Forum*, vol. 27, no. 3. Wiley Online Library, 2008, pp. 831–838.

[20] M. Ishihara, K. Misue, and J. Tanaka, "Ripple presentation for tree structures with historical information," in *Proceedings of the 2006 Asia-Pacific Symposium on Information Visualisation-Volume 60*. Australian Computer Society, Inc., 2006, pp. 153–160.

[21] "Logilab - xmldiff." [Online]. Available: http://www.logilab.org/859

[22] D. Hottinger and F. Meyer, "Xmldiff report." [Online]. Available: http://archiv.infsec.ethz.ch/education/projects/archive/XMLDiffReport.pdf

[23] O. D. T. Committee, "Docbook." [Online]. Available: http://www.docbook.org/

[24] "Basex." [Online]. Available: http://basex.org

[25] T. Kaiserslautern, "Brackit." [Online]. Available: http://code.google.com/p/brackit/

[26] D. Holten, "Hierarchical edge bundles." [Online]. Available: http://www.win.tue.nl/~dholten/papers/bundles_infovis.pdf

[27] A. Holupirek, "Filesystem markup language." [Online]. Available: https://github.com/holu/fsml

[28] D. Arnold, "Lexical functional grammar." [Online]. Available: http://www.essex.ac.uk/linguistics/external/lfg/FAQ/WhatIsLFG.html

[29] Wikimedia, "Wikipedia data dumps." [Online]. Available: http://meta.wikimedia.org/wiki/Data_dumps

[30] M. Team, "Wiki2xml." [Online]. Available: http://www.modis.ispras.ru/texterra/download/wiki2xml.zip