International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 07 Issue: 05 | May - 2023 SJIF 2023: 8.176 ISSN: 2582-3930

A SURVEY ON CHANGE DETECTION IN HIERARCHICAL DATA STRUCTURES FOR DATA WAREHOUSES

B Chirag Baliga[1], Uday A S, Dr. Rashmi R[3]

Information Science and Engineering, RV College of Engineering®, Bengaluru

---

Abstract: *Change detection algorithms play a crucial role in data warehousing applications, where efficiently identifying and tracking differences between data structures is essential. This paper presents a comprehensive review of existing change detection algorithms, focusing on their methodologies, assumptions, and performance characteristics in the context of data warehouse environments. The review emphasizes the applicability and results of these algorithms considering factors such as matching accuracy, handling of moved subtrees, support for subtree deletion, and overall effectiveness in capturing changes. Experimental evaluations and comparative analyses are conducted to assess the performance of the algorithms. Some other applications such as web publishing, query systems, and database management, version and configuration management are also discussed. The findings of this review provide valuable insights into the state-of-the-art change detection algorithms, enabling researchers and practitioners in the field of data warehousing to make informed decisions when selecting and implementing change detection solutions.*

Keywords: Change detection, XML, hierarchically structured data, data warehouse, version management, similarity search, tree edit distance, tree edit script

I. INTRODUCTION

In today's information-rich world, where data is constantly evolving and being updated, change detection within existing data is of utmost importance to track modifications and perform necessary actions. Whether it is monitoring updates in a web application, tracking modifications in a database, or synchronizing data across different systems, being able to identify and understand the changes within hierarchical structures is crucial. By comparing two files in formats such as XML, it becomes possible to detect insertions, deletions, and modifications of elements, attributes, and their values. This enables efficient tracking and management of changes, ensuring data integrity and consistency.

At its core, change detection involves the comparison of two or more data structures to identify added, deleted, or modified elements. The data structures can range from simple text documents to complex hierarchical data representations such as XML or JSON. The primary objective of change detection algorithms is to generate an edit script or a set of operations that succinctly describes the transformations required to transform one data structure into another. Hierarchically structured data are commonly utilized to store configuration settings, specifications, and other critical information related to software systems. By comparing different versions of XML files, it becomes possible to detect changes in configurations, allowing for the identification of discrepancies and conflicts. This enables efficient management of software versions and configurations, facilitating seamless deployment, troubleshooting, and maintenance.

One of the key challenges in change detection is handling large datasets efficiently. As the size of the datasets increases, the computational complexity of the algorithms becomes a crucial factor. Several algorithms address this challenge by employing optimized data structures, indexing techniques, and parallel processing to improve runtime performance and scalability. This review aims to serve as a valuable resource for researchers and practitioners in the field, facilitating the exploration and advancement of techniques for effective analysis and comparison of hierarchically structured data.

II. REVIEW OF ALGORITHMS

The basic algorithms which are typically used in change detection are discussed in [11]. The paper mostly focuses on **Outer Join Algorithms**, **optimization using compression** and the **Window algorithm**.

The **basic sort merge algorithm** takes advantage of the sorted order by making an optimization; when two records are being matched, the record with the smaller key is guaranteed to have no matching records. If the algorithm is executed as part of a differential process (where changes are detected between snapshots), it is possible to save the sorted file from the previous snapshot. This way, only the second file needs to be sorted. The second file (F2) can be sorted using the multiway merge-sort algorithm, which constructs runs (sequences of blocks) with sorted records. After several passes, the file is partitioned into progressively longer runs until there is only one run left. The complexity and IO cost of the basic sort merge join algorithm depend on the size of the input files and the available memory. In general, it takes approximately $2 * |F1| * \log_{M1}|F1|$ IO operations to sort a file of size F1. However, if enough memory is available ($M1^2 > |F1|$), the sorting can be done in $4 * |F1|$ operations (using two passes). The second phase of the algorithm, involving scanning and merging the two sorted files, requires $|F1| + 5 * |F2|$ IO operations.

The **partitioned hash join algorithm** partitions the input files into buckets using a hash function applied to the join attribute. The steps of this algorithm are as follows:

1. The input files are divided into multiple buckets based on the hash values of the join attribute.
2. Records from the buckets with the same hash value are compared to find matches.
3. The algorithm considers each pair of records from the matching buckets and produces the desired join results.

This algorithm offers advantages in terms of parallelism and reducing the need for sorting. However, the specific details of the algorithm's implementation and its IO cost are not provided in the given text. A detailed discussion of the partitioned hash algorithm is given where they show that the IO cost incurred is |F1| + 3 * |F2|.

The **Window Algorithm** assumes that matching records are physically close to each other in the files. While the matching records may not be in the same position due to possible reorganizations at the source, it is expected that they would still be within a relatively small area, such as a track. File reorganization algorithms typically rearrange records within a physical sub-unit, leading to this expectation. The algorithm takes advantage of this assumption and the increasing capacity of main memory. It maintains a moving window of records in memory for each snapshot. Only the records within the window are compared, with the hope that the matching records will occur within the window. Unmatched records are reported as either inserts or deletes, which may occasionally result in useless delete-insert pairs. However, it is stated that a small number of such pairs can be tolerated. To implement the window algorithm, the available memory is divided into four distinct parts: input buffers and aging buffers for each snapshot. The input buffer is responsible for transferring blocks from disk, while the aging buffer serves as the moving window mentioned earlier. Since the files are read through only once, the IO cost for the window algorithm is only |Fl| + |F2| regardless of snapshot size, memory size and number of updates. Thus the window algorithm achieves the optimal IO performance among the other algorithms in [11].

In the paper [12], the same authors proposed yet another algorithm, **MH-DIFF** for change detection in two hierarchical structure snapshots such as trees. This algorithm gives the changes in a descriptive way with more complex operations, unlike previous counterparts, which give it as a sequence of simple insert and delete operations, which do not convey an intuitive understanding of the changes. The algorithm works by first creating a hash table of the nodes in the tree. The hash table is used to quickly find the nodes that have changed since the last time the tree was checked. The algorithm then uses a series of heuristics to determine the type of change that has occurred. The heuristics are based on the following.

1. The number of nodes that have changed.
2. The location of the nodes that have changed.
3. The values of the nodes that have changed.

The complexity of the MH-Diff algorithm is O(ND), where N is the number of nodes in the tree and D is the depth of the tree. The O(ND) complexity is due to the fact that the algorithm has to create a hash table of all the nodes in the tree, and then it has to compare the hash table of the old tree to the hash table of the new tree. The O(ND) complexity can be reduced by using a more efficient data structure to store the nodes in the tree, such as a balanced binary tree.

The paper in [15] discusses the **RWS-Diff** algorithm, which uses random walks similarity (RWS) measure to find similar subtrees rapidly. It is able to compute a cost-minimal edit script in log-linear time while having the robustness of a similarity based approach. This algorithm involves the following steps:

1. Simple Matching Step: This initial step attempts to identify obvious common structures in both versions of the tree. The nodes that are successfully mapped in this step are excluded from further matching steps, leading to improved efficiency.
2. Construction of Feature Vectors: For the unmapped subtrees in both trees, fixed-length feature vectors are created. These feature vectors serve as representations of the subtrees and exhibit similarity if the subtrees themselves are similar. The similarity measure used is the squared Euclidean distance between the feature vectors.
3. Index Structures for Nearest Neighbors Queries: In this step, appropriate index structures are constructed to facilitate nearest neighbors queries among the feature vectors. These index structures help in efficiently identifying potential candidates for mapping based on similarity.
4. Mapping of Unmapped Subtrees: Using the constructed index structures, previously unmapped subtrees are mapped by searching for possible candidates through nearest neighbors queries. This step aims to find similar subtrees in the two versions of the tree and establish corresponding mappings.
5. Generation of the Edit Script: Finally, based on the edit mapping obtained from the previous steps, an edit script is generated. The edit script represents the operations required to transform one version of the tree into another. The goal is to create an edit script that minimizes the overall cost of the required edits.

The generation of random walk feature vectors for all subtrees in a tree has a complexity of O(n), where n represents the number of nodes in the tree. Since there can be O(n) subtrees in both trees that need to be mapped, mapping one subtree may only require O(log n) operations. The nearest neighbors lookup in the index structures typically has a complexity of O(log n). The index structures are adjusted to ensure worst-case O(log n) behavior by sacrificing some approximation quality in extreme cases. This means that a single RWS mapping operation remains within O(log n) complexity. Insertions or lookups in the mapping 'M' are also in O(1) since dense integers can be assigned to each node in tree B, and 'M' can be implemented as an array indexed by these integers. Finally, the edit script generation loops only twice over both trees, resulting in an overall complexity of O(n). This meets the desired complexity bound of O(n log n) for the entire RWS-Diff algorithm.

[13] discusses the **KF-Diff+** algorithm, which is specifically tailored for XML documents where each node has a unique key among its siblings. In this scenario, KF-Diff+ allows move operations only between nodes with the same parent, and it can compute the diff in linear time, O(n), where n is the size of the input. [18] proposed the **XyDiff** algorithm that does not rely on strong assumptions about the XML document structure. It utilizes tree hashes that are invariant to the sibling order, enabling efficient identification and mapping of moved subtrees. XyDiff first maps the moved subtrees and then proceeds to map nodes in the vicinity of these mapped subtrees. The overall runtime complexity of XyDiff is O(n log n), where n is the size of the input. XyDiff has shown good performance when there are large unchanged subtrees in the XML documents.

These algorithms have their implementations in programming languages like Python and Java as described in TABLE 1. The Javascript implementations are often used to evaluate change in DOM structure of web pages.

TABLE 1. Implementation of some algorithms

Algorithm	Language	Link	Last update
XmlDiff [3]	Python	Pypi.org	present
DeltaXML	Java	Commercial	present
XyDiff [7]	C++	Github	2015
Xdiff [31]	C++	Github C++	2015
DiffXml [4]	Java	Github	2018
XOp [11]	Java	Living-pages.de	2009
FC-XmlDiff [12]	Java	Github	2009
DiffMK [18]	Java	Sourceforge	2015
JXyDiff [24]	Java	Github	2009
XCC [22]	Java	Launchpad	2009
JNDiff [5]	Java	Sourceforge	2014
Node-delta [13]	JavaScript	Github	present

III. IMPLEMENTATION USING XML DATA

XML, the widely adopted eXtensible Markup Language, has emerged as the de facto standard for document publishing and transport on the web.. Given the dynamic nature of online information, there arises a need for a tool that can effectively detect changes in XML documents. Efficient operation of this tool becomes paramount, especially when handling large volumes of evolving documents. Efficient operation of this tool becomes paramount, especially when handling large volumes of evolving documents. To illustrate the significance of such a tool, [1] introduced a scenario where a parent intends to purchase books for their children from an online auction site, relying on a search engine equipped with this change-detection capability. During the parent's initial visit, a list of currently available books and their associated information is obtained. Two hours later, when the search engine retrieves updated data, the change-detection tool comes into play to discern the alterations that have occurred during this time frame. Firstly, the tool determines whether the two versions are identical or not. If disparities exist, it proceeds to match each book segment in the previous version with those in the new version, discerning which books are still available, which have been sold, and which ones are newly listed. Despite the change in the ordering of the two books, both of them remain available in this example.

For each book that is still available, the change-detection tool further analyzes the modifications made to the associated information. The tool would notify the consumer that there are now two fewer hours remaining to submit a bid for both books. Specifically, the Harry Potter book currently has a bid price of $10, with Mark as the bidder, who possesses a rating of 125. As for the Tom Sawyer book, it currently stands at a bid price of $4.50, and there have been no changes in the bidder's identity. The paper also introduces an algorithm called X-diff for computing the differences between two versions of an XML document.

The key features of the algorithms are:

1. XML Structure Information: X-Diff introduces the notion of node signature and a new matching between the (XML) trees corresponding to the two versions of a document. Together, these two features are used to find the minimum-cost matching and generate a minimum-cost edit script that is capable of transforming the original version of the document to the new version.
2. Unordered Trees: Since XML documents can be represented as trees, the change detection problem is related to the problem of change detection on trees. For database applications of XML the authors believe that the unordered tree model is more important. Thus, X-Diff is designed to handle unordered tree representations of XML documents.
3. High Performance: Change detection on unordered trees is substantially harder than that on ordered trees, which. has been shown to be NP-Complete in the general case. By exploiting certain features of XML documents, a polynomial algorithm is presented to compute the “optimal” difference between two XML documents.

[1] also discusses the tree representation of the XML documents which is applied when detecting changes in them. To create an efficient algorithm for detecting changes in XML documents, it is essential to have a grasp of the hierarchical structure within XML. According to the Document Object Model (DOM) specification, an XML document can be represented as a tree. The paper explores three types of nodes found in the DOM tree: element nodes, text nodes, and attribute nodes. Element nodes are non-leaf nodes with a single label (name), text nodes are leaf nodes with a single label (value), and attribute nodes are leaf nodes with two labels (name and value). As per the DOM specification, element and text nodes have a specific order, while attribute nodes do not. In many cases, XML documents can be treated as unordered trees, where only the relationships between ancestors matter, and the left-to-right order among siblings is insignificant. In the X-Diff approach, the focus is on detecting changes in unordered trees. Most correction methods designed for ordered tree-to-tree comparisons are not suitable for unordered trees because their accuracy usually relies on preserving the left-to-right order when matching nodes. Two trees are considered isomorphic if they are identical except for

the arrangement of siblings. In X-Diff, equivalence between two trees is determined based on their isomorphism.

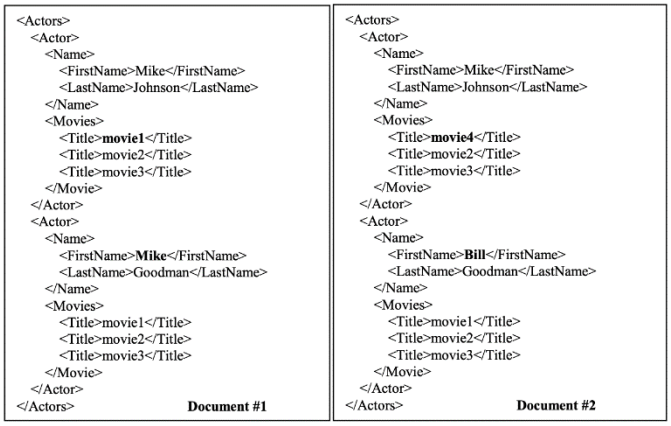

FIGURE 1. Two sample XML documents

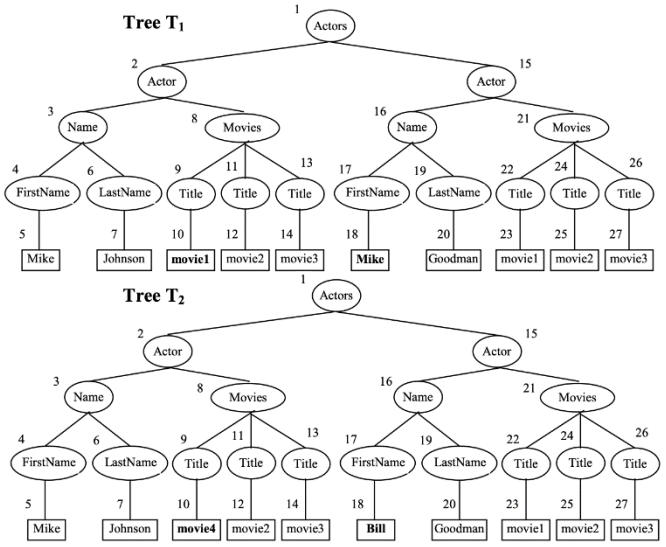

FIGURE 2. Tree representation of the documents in Figure 1

[2] utilized an example in configuration management, consider the correlation of data stored in an architect's database with data stored in an electrician's database, where both databases are for the same building project. For autonomy reasons, the databases are updated independently. However, periodic consistent configurations of the entire design must be produced. This can be done by computing the deltas with respect to the last configuration and highlighting any conflicts that have arisen.

The work on change detection reported in the paper has four key characteristics:

1. Nested Information: The paper's focus was on hierarchical information, not "flat" information. With flat information deltas may be represented simply as sets of tuples or records inserted into, deleted from, and updated in relations. In hierarchical information, we want to identify changes not just to the "nodes" in the data, but also to their relationships. For example, if a node (and its children) is moved from one location to another, we would like this to be represented as a "move" operation in the delta.
2. Object Identifiers Not Assumed: For maximum generality the authors do not assume the existence of identifiers or keys that uniquely match information fragments across versions. For example, to compare structured documents, we must rely on values only since sentences or paragraphs do not come with identifying keys. Similarly, objects in two different design configurations may have to be compared by their contents, since object-ids may not be valid across versions.
3. Old, New Version Comparison: Although some database systems—particularly active database systems—build change detection facilities into the system itself, the paper focuses on the problem of detecting changes given old and new versions of the data. They believe that a common scenario for change detection—especially for applications such as data warehousing, or querying and browsing over changes—involves "uncooperative" legacy databases (or other data management systems), where the best one can hope for is a sequence of data snapshots or "dumps".
4. High Performance: The goal was to develop high performance algorithms that exploit features common to many applications and can be used on very large structures. In particular, present algorithms that always find the most "compact" deltas, but are expensive to run, especially for large structures. (The running time is at least quadratic in the number of objects in each structure compared. The properties of these algorithms are described in more detail in Section II.) The algorithms discussed in the paper are significantly more efficient (intuitively, our running time is proportional to the number of objects times the number of changes), but may sometimes find non-minimal, although still correct, deltas.

IV. EXPECTED OUTCOMES AND CONCLUSIONS

We have examined and compared several change detection algorithms, each offering unique approaches and features for identifying and analyzing differences within data structures. The reviewed algorithms, including KF-Diff+, XyDiff, RWS-Diff, and the basic sort merge join algorithm, have demonstrated their effectiveness in various applications and domains.

The KF-Diff+ algorithm showcases its suitability for XML documents with unique node keys, allowing efficient computation of diffs with move operations. However, its applicability is limited to specific types of data structures that adhere to the key uniqueness constraint. XyDiff stands out as an algorithm that operates without strong assumptions and runs in less than quadratic time. By utilizing tree hashes

invariant to sibling order, XyDiff efficiently detects and maps moved subtrees. The algorithm's ability to handle large unchanged subtrees contributes to its overall performance and produces smaller edit scripts. RWS-Diff introduces an elaborate similarity measure and a five-step process for constructing approximate cost-minimal edit mappings. With its focus on finding better mappings than previous approaches, RWS-Diff demonstrates improved accuracy and results. Furthermore, it supports subtree deletion, enabling the removal of surplus data from one of the trees. The basic sort merge join algorithm, although not specifically designed for change detection, serves as a foundational method for comparing and identifying differences between data structures. It excels in scenarios where the data is sorted and requires a straightforward comparison.

Comparing these algorithms, we observe that they vary in their assumptions, computational complexity, runtime performance, and capabilities. Each algorithm caters to specific data structures, optimization goals, and domain requirements. Researchers and practitioners should carefully consider these factors when selecting an algorithm for their applications. While some algorithms achieve better runtime performance or handle specific constraints, such as unique keys or sibling order invariance, others offer more advanced features like semantic change detection or support for subtree deletion. The choice of algorithm should align with the specific needs of the application, considering factors such as dataset size, complexity, the presence of semantic changes, and desired accuracy.

Further research and development in change detection algorithms continue to address the evolving demands of data-intensive applications. By considering the strengths and limitations of existing algorithms, we can pave the way for improved techniques that provide faster, more accurate, and scalable solutions for change detection in various domains and applications.

V. REFERENCES

[1] Y. Wang, D. J. DeWitt and J. . -Y. Cai, "X-Diff: an effective change detection algorithm for XML documents," Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405), Bangalore, India, 2003, pp. 519-530.

[2] Chawathe, S.S. et al. (2005) “Change detection in hierarchically structured information,” ACM SIGMOD Record, 25(2).

[3] Bertino, E., Guerrini, G. and Mesiti, M. (2004) “A matching algorithm for measuring the structural similarity between an XML document and a DTD and its applications,” Information Systems, 29(1), pp. 23–46.

[4] Haw, S.C. and Rao, G.S. (2007) “A comparative study and benchmarking on XML parsers,” The 9th International Conference on Advanced Communication Technology [Preprint].

[5] S. Abiteboul, S. Cluet, and T. Milo. A database interface for file update. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 1995.

[6] H.C. Howard, A.M. Keller, A. Gupta, K. Krishnamurthy, K.H. Law, P.M. Teicholz, S. Tiwari, and J. Ullman. Versions, configurations, and constraints in CEDB. CIFE Working Paper 31, Center for Integrated Facilities Engineering, Stanford University, April 1994.

[7] J. Widom and J. Ullman. C3: Changes, consistency, and configurations in heterogeneous distributed information systems. Unpublished project description, available through the URL http://www-db.stanford.edu/c3/synopsis.html, 1996.

[8] C. M. Hoffmann, M. J. O’Donnell, “Pattern Matching in Trees”, Journal of the ACM, 29: 68-95, 1982.

[9] J. Clark, S. DeRose, et al., “XML Path Language (Xpath) Version 1.0”, November 2008

[10] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In ICDE, 1998.

[11] Labio, Wilburt and Hector Garcia-Molina. “Efficient Snapshot Differential Algorithms for Data Warehousing.” Very Large Data Bases Conference (1996).

[12] Sudarshan S Chawathe and Hector Garcia-Molina 1997, Meaningful change detection in structured data, ACM SIGMOD Record 26, 2 (1997)

[13] H. Xu, Q. Wu, H. Wang, G. Yang, and Y. Jia. KF-Diff+: Highly efficient change detection algorithm for XML documents. In ODBASE, 2002.

[14] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In SIGMOD, 2005.

[15] Finis, Jan & Raiber, Martin & Augsten, Nikolaus & Brunel, Robert & Kemper, Alfons & Färber, Franz. (2013). RWS-Diff: Flexible and efficient change detection in hierarchical data. International Conference on Information and Knowledge Management, Proceedings. 339-348.

[16] N. Augsten, M. B¨ohlen, and J. Gamper. The pq-gram distance between ordered labeled trees. TODS, 35(1), 2005.

[17] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. SIAM J. of Computing, 18(6), 1989

[18] A Marian, S Abiteboul, G Cobéna, L Mignet, Change-Centric Management of Versions in an XML Warehouse, VLDB 2001.

[19] K. Zhang and T. Jiang. Some MAX SNP-hard results concerning unordered labeled trees. IPL, 49(5), 1994.

[20] D. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical report, Queen's University, Kingston, 1995.

[21] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbor search in high-dimensional spaces. In CVPR, 1997.

[22] L. Boyer, A. Habrard, and M. Sebban. Learning metrics between tree structured data: Application to image recognition. In ECML, 2007