Utilizing Delta Trees for Efficient, Iterative Exploration and Transformation of Semi-Structured Contents

Nico Schäfer
Department of Computer Science
TU Kaiserslautern
Kaiserslautern, Germany
nschaefer@cs.uni-kl.de

Sebastian Michel
Department of Computer Science
TU Kaiserslautern
Kaiserslautern, Germany
michel@cs.uni-kl.de

***Abstract*—The keywords data exploration or data wrangling summarize various different query workload scenarios in which users aim to explore or tailor data to their needs. For semi-structured data, next to commonly used SQL-style select-from-where and aggregation queries, also the structure of the possibly-nested schema-free data can be altered, schema attributes renamed, and so on. This typically involves various rounds of refining or discarding queries—imposing that intermediate results as well as the original sources cannot be eliminated. In this work, we extend our prior work on JODA, a vertically scalable, versatile JSON data processor, to make use of so-called delta trees for the succinct representation of incrementally created query results.**

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
doi:10.1109/ICDE51399.2021.00173

I. Introduction

Data scientists and practitioners are commonly faced with large amount of structured or semi-structured data, often in form of CSV or JSON [1] data, stemming from access logs, scientific experiments, Twitter streams, etc. In order to explore or transform data for gained insights or later-on use in higher-level applications, tools need to be able to process potentially large amounts of data through several iterative explorative or transformative stages. Such data wrangling operations require highly-responsive tools that do not require heavy initial indexing, or other forms of data preprocessing. As shown by the seminal work on NoDB [2], it is indeed possible to beat full-fledged database systems for ad-hoc query workloads that do not exhibit potential for data indexing. In a similar vein, we previously demonstrated JODA [3][1], a lightweight data processor for exploring and transforming large amounts of JSON data in a vertically scalable manner. Common operations in semi-structured document processing are filtering documents and adding, removing, moving, and aggregating values. Consider for instance the case of data scientists working on a large sample of Twitter tweets. While some scientists first extract textual content and geo-coordinates of Canadian tweets written in French they later observe that also the author of the tweet is required, others need to convert the original schema (attribute names) to match their existing data visualization libraries. To execute these operations, systems have to either edit the original document, or perform the operation on a copy. The first option may not always be possible, if the base documents are to be used in future processing steps or by multiple data scientists working in parallel. The second option is undesirable, because it introduces a huge overhead regarding performance and memory usage, for copying the documents. In this paper, we propose the usage of so-called delta trees to succinctly represent data modifications while keeping original datasets intact, with the option to perform materialization of increased runtime performance.

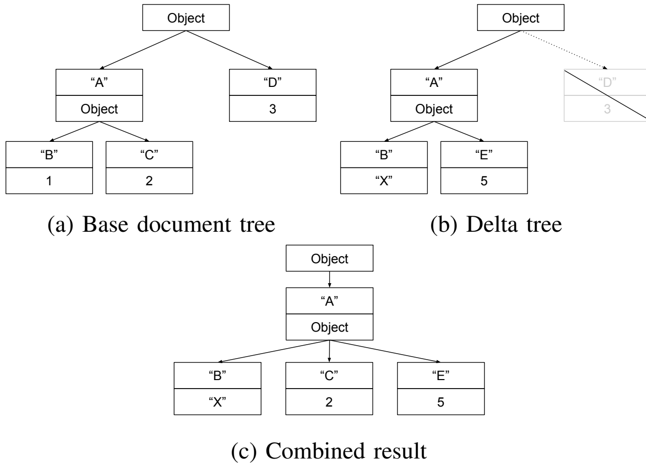
[1]https://youtu.be/HSThB8mxTTA

A. Problem Statement

We consider a collection of semi-structured documents over which users execute queries. Queries can select certain parts of a document, rename attributes, or perform aggregations. We say a collection is derived from another one by means of queries. For an individual document, given its tree representation, queries may either update, create, or remove a node in the tree.

Consider the example document tree in Fig. 1a. Assume a data scientist wants to change the value of “B”, create a child “E” of “A”, and remove “D”. The resulting document is shown in Fig. 1c. The base document might be required for later use as-is and can thereby not be transformed in-place. A naïve implementation would now copy the document and edit this copy. This will lead to redundantly stored data, as the “C” attribute is now duplicated in the base and result documents.

Delta trees, instead, only store the changes made by the query in a new tree as shown in Fig. 1b. Then, by algorithmically combining the base and delta tree, we can create the result document on demand, without storing redundant data. We show that this may lead to vastly reduced memory requirements, while only incurring an acceptable runtime overhead, if not even improving the query time.



(a) Base document tree (b) Delta tree

(c) Combined result

Fig. 1: Example of base document with a delta tree

B. Contribution and Outline

With this paper, we make the following contributions:

- We propose the usage of delta trees as a highly space-efficient schema for exploratory and transformative queries over semi-structured documents.
- We give an in-depth description of implementation aspects of integrating delta-trees in our JSON data processor JODA.
- We discuss further optimizations to the approach for additional performance improvements.

The remainder of this paper is organized as follows. In Section II, related work is discussed. Then the core concepts and theoretical models of the approach are given in Section III. Realization and optimizations of this model inside JODA are described in Section IV. This implementation is then evaluated in Section V, followed by the conclusion in Section VI.

II. Related Work

Change detection in hierarchical data, which is often based on tree edit distances [5], [6], is a well studied topic [7]. XML trees are a special case of hierarchical data, for which many approaches have been created [8], [9]. One of the most cited approaches is X-Diff by Wang *et al.* [10]. They propose an algorithm to find changes in XML trees by only using ancestor relationships. Most algorithms created for XML can be adapted to work with JSON trees, as they have a similar hierarchical structure and path expressions. While these approaches aim at identifying differences between two given trees, our approach wants to store changes without duplicating any data.

Almeida *et al.* proposed a map as a Conflict-free Replicated Data Type (CRDT) [4], which can be synchronized by sending delta changes of the modification action to replicated instances. This data type can also be nested and JSON documents can be translated to and from nested maps. But the computational overhead required for conflict-free synchronization is too large for our use case.

Some database systems create query plans that decide between late and early materialization of intermediate query results [11]. Early materialization means that result tuples are constructed immediately and then processed further by remaining operations or queries. On the other hand, late materialization tries to push expensive operations as far back as possible, to potentially reduce the required work of result tuple construction. In our case, early materialization is similar to the default approach of immediately constructing and storing the result documents. The approach we are suggesting is a compromise between early and late materialization, by immediately materializing the changes of a document, while deferring the potential construction of a complete result document to a later point of time.

III. Delta Trees

Most semi-structured documents can be represented as a directed tree. A directed tree is an acyclic connected graph $T' = (V, E)$, with V being the vertices in the graph and E the edges. This representation is also often used as the in-memory storage model of the documents to allow easy traversal and modification. More specifically, the Document Object Model (DOM) [12] is often used to represent XML and HTML documents. Every document has one root node, which can contain children nodes that are themselves a root nodes of subtrees. The leaves of the tree represent the stored data. Additionally, every node is labeled with meta data, like the attribute name and/or type.

We augment the tree definition to obtain a document $T = (V, E, I, A, L, D)$. V and E are still denoting vertices and edges, respectively. The vertices are distributed into the distinct subsets $I \cup A = V$, with I being the inner structural nodes and A the atomic leaf nodes. An edge $(x, y) \in E$ represents a parent/child relationship with y being the child node of x. L is a set of labels, a mapping of $I \to \Sigma^*$, where Σ is a domain specific alphabet. For each parent node, the labels of all its children are unique. D represents the actual stored data and is a mapping $A \to \Sigma^*$ from the leaf nodes to the data. This is a very generic abstraction, to which JSON, XML, and YAML documents comply.

A. Path

A path p is an ordered list of labels $p = (l_1, \ldots, l_n)$ with $l_i \in \Sigma^*$. As the child labels are unique in the scope of the parent node, it is possible to uniquely identify each node in the tree by a path. For each inner node $i \in I$ there exists a mapping $vtp(v) = v \to p$. There is also the reverse mapping, path-to-vertice. Given a tree T and a path p:

$$ptv(T, p) = \begin{cases} v & \exists v \in V, vtp(v) = p \\ \varnothing & \text{else} \end{cases}$$

XPath [13] as well as JSON pointer [14] adhere to this concept of a path.

B. Delta Tree

Using the previous definitions, we can now formally introduce *delta trees*. A delta tree $D = (b, t, P_D)$ is a tuple consisting of a base tree $b = (V_b, E_b, I_b, A_b, L_b, D_b)$, a tree $t = (V_D, E_D, I_D, A_D, L_D, D_D)$ containing the changes relative to b, and a set of paths P. The base tree b and the change tree t are both valid documents.

The set of paths P describe all the paths that have been changed in this delta tree. For example, would the paths `{(<Root>, "A","B"), (<Root>, "A","E"), (<Root>, "D")}` denote all changes of the delta tree in Figure 1b. Given this set and the base tree b, we can define the set of overwritten vertices with $OV(b_D, P_D)$. For all $v \in V_b, v \in OV(b_D, P_D)$ iff:

1) $\exists p \in P_D, vtp(v) = p$
2) $\exists v' \in OV(b_D, P_D), (v', v) \in E_b$

This means, a vertex of b is overwritten in D if it is either directly changed in the delta tree, or if it is a child of a changed vertex.

The document resulting from combining the base tree b with the delta tree t is again a tree $R = (V_R, E_R, I_R, A_R, L_R, D_R)$ with:

$$V_R = \{v | v \in V_b \land v \notin OV(b_D, P_D)\} \cup V_D$$

$$E_R = \{(x, y) | (x, y) \in E_b \land x, y \notin OV(b_D, P_D)\} \cup E_D$$

I_R, A_R, L_R, and D_R are defined analogously to V_R.

C. Delta Hierarchy

As mentioned previously, a delta tree describes changes to a base tree. Thus, it is possible to have a base tree b and a delta tree $D = (b, t_D, P_D)$ derived from it. Let the result of merging these trees be R_D. We can now use this result tree as a base tree for an additional delta tree $D' = (R_D, t_{D'}, P_{D'})$. Analogously, we obtain the result tree $R_{D'}$.

This means, we can build delta trees on the results of previous trees. A set of delta trees $H = (D_0, D_1, \ldots, D_n)$, where each delta tree references the result of the previous one is called a *delta hierarchy*. D_0 represents the base tree b, which can be seen as a delta tree $D_0 = (\varnothing, b, \{< \text{Root} >\})$, with no base document and where the root node is overwritten, thereby using the whole change tree. The result of the delta hierarchy equals the result tree of the upper most delta tree $R_H = R_{D_n}$.

IV. Realization & Optimizations

We now describe how delta trees are implemented inside JODA. JODA uses the RapidJSON parser, which creates an in-memory DOM tree representation. Given a delta hierarchy $H = (D_0, \ldots, D_i, \ldots, D_n)$, as defined above. The document trees are represented internally as DOM trees, which can directly be mapped to our theoretical model of document trees presented before.

H supports the following actions:

- Traverse a result document R_i using the visitor pattern.
- Materialize changes of a sub-tree at path expression p.
- Get the sub tree of a path expression p.

A. Traversal with Visitor Pattern

A well known method of traversing structured data without changing it is the visitor pattern [15]. With it we can traverse our internal tree structure to perform many different tasks. For example, is it used for stringification and duplication of whole documents or parts of them within our system. An efficient

Data: p = ' '; $D_0, \ldots, D_m$
1 **if** *IsShared(p,$D_0, \ldots, D_m$)* **then**
2 O = GetLastOverwrite(p,$D_0, \ldots, D_m$));
3 **for** *i = 0 to D_m* **do**
4 members += GetMembers(D_i);
5 **end**
6 **for** *member in members* **do**
7 Visit member;
8 Recurse(p+'/'+member.id,$D_0, \ldots, D_m$);
9 **end**
10 **else**
11 n_D = GetBaseNode(n);
12 Visit n_D;
13 **end**

Algorithm 1: Simultaneous traversal algorithm

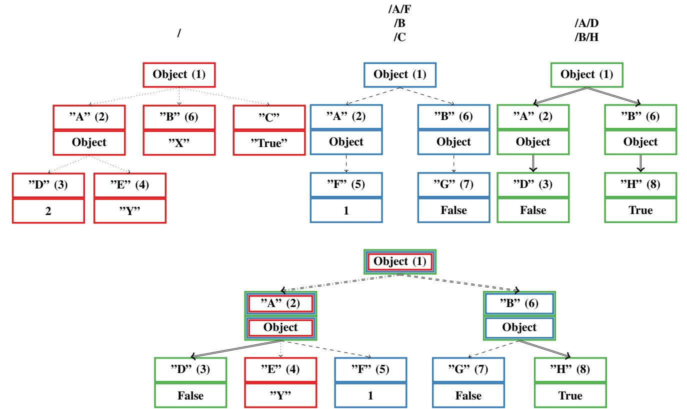


Fig. 2: Simultaneous traversal

implementation of this pattern is crucial for the efficiency of our delta tree implementation.

As mentioned in Section III-C, the result of a delta hierarchy can be computed by iteratively creating an intermediate result for every level of the delta hierarchy. Using this approach on a delta tree D_1 which is based on the document D_0, we would combine both trees, to build the result document R_1. However, this approach does not work well with delta hierarchies of many levels, as too many intermediate results have to be computed. Instead, Algorithm 1 traverses a delta hierarchy as if it was one document (illustrated in Fig. 2), for an arbitrary number of delta DOM trees. We start at the root document of the upper most delta tree. First, we check using a function `IsShared`, if the children of the given node are distributed over multiple trees. By storing all overwritten paths in a suitable structure, like a map, we can check this by performing one hash lookup per delta tree.

If this node is shared—like the root, "A", and "B" nodes in the example—we get the upper most overwrite for the given node, with the `GetLastOverwrite` function, which returns the document highest up in the delta hierarchy which completely replaced this attribute. In case of node "A", the base document was the last one to overwrite the value, as the base document overwrites everything. For node "B", the second layer overwrote this sub-tree last. For this, we also only have to do a prefix check on the overwritten paths. In

our implementation, we include this check in the `IsShared` function call, to reduce the runtime. We then collect all unique paths of children of this node and nodes with the same path in the trees above. Each member is then visited once, and the function is called with each of their paths again.

B. Retrieval of Atomic Values

Most query functions only have to read atomic values. As atomic values are never shared and can be read from a single delta tree directly, we introduce a `getAtomic` function which optimizes these accesses. By using the previously explained `GetLastOverwrite` function, we get the delta tree which overwrote the given path last. The atomic value we try to find is either in this tree, or does not exist at all. Hence, we can simply extract the value from this one delta tree.

C. Partial Materialization

Simultaneous traversal of delta hierarchies is still more expensive than directly accessing a normal (materialized) tree. To mitigate this issue, we introduce a method that allows *partial materialization* of a given delta tree. Given a delta hierarchy $H = (D_0, \ldots, D_n)$ and a path p, we can materialize p into D_n.

Algorithm 2 shows the materialization procedure. Lines 1–6 traverse the delta hierarchy to the required path p and prune delta trees, which are not required for further computation, from the search space. The resulting delta hierarchy is then used to take a special `CopyVisitor` object, which creates a deep-copy of the sub tree, in Line 7—we use Algorithm 1 to traverse this sub tree in the delta hierarchy. This copy is then assigned to path p within the topmost delta tree and the materialized path is added to the set of overwritten paths.

```
  Data: p; D_0, ..., D_n;
1 for D_i ∈ D_0 to D_{n-1} do
2 |  Node = D_i.find(p);
3 |  if Node is null then
4 |  |  remove D_i from list;
5 |  end
6 end
7 D_n.set(p,Accept(CopyVisitor,p, D_0, ..., D_n))
8 D_n.P+ = p;
```

Algorithm 2: Materialize_P()

If the path to be materialized points to the root, the entire delta hierarchy is materialized. In this case, the uppermost delta tree is converted into a normal tree, and removed from the delta hierarchy. This is called a *complete materialization*.

Materializing (parts of) the delta hierarchy increases memory consumption, as we store duplicated data. Partial materialization could be used by the system, if specific paths in the delta hierarchy are accessed frequently by queries and the increased memory footprint is tolerably small. Hence, we can choose, depending on the available memory, if we want to materialize a path to decrease query runtime.

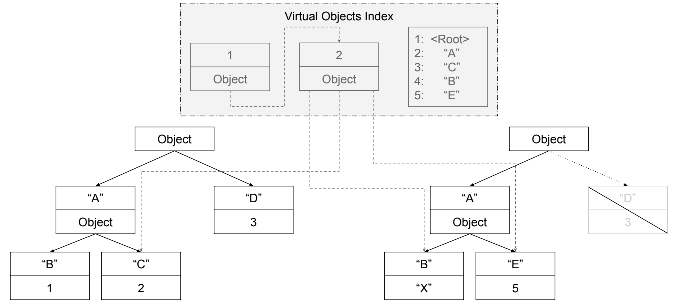


Fig. 3: Object index

D. Object Indexing

To prevent partial materialization of objects, a *virtual object index* is created. This index combines the key benefits of delta trees with the advantage in read performance of materialization. A virtual object is a list of tuples containing an attribute id and a pointer to a value or nested virtual object, as shown in Figure 3. The attribute id is a numerical value, retrieved by mapping a string attribute name to a numerical value using a hash map. This has two advantages. (1) Having a numerical value reduces the cost of comparisons needed for the linear search of children. (2) The string dictionary can be shared by many documents, thus, reducing the required memory of this index.

We create these virtual objects as soon as an object, that is distributed over multiple trees in the delta hierarchy, is traversed for the first time. During traversal, we map the attribute names of the children to the attribute id and add it to the virtual object, together with the pointer of the actually traversed value. Each value may reside in a different tree within the delta hierarchy. The traversed object is then replaced by the virtual object in the highest delta tree of the hierarchy. Future accesses of the object can then use the created index without traversing multiple trees.

E. Adaptive Algorithm

The main advantage of delta trees is the reduced memory requirement. Evidently, each delta tree is smaller or equal in size as the result tree that is given by combining the whole delta hierarchy—as all of its nodes are contained in the result, plus potential additional nodes from the base document. Thus, the memory cost of delta trees should always be smaller or equal to materializing the whole result. However, this may not always be the case in practice, as the RapidJSON library, that is used to create JSON documents, creates each new object and array with 16 placeholder children. In many cases, this is a sensible decision, as reallocating memory for more children is an expensive operation and objects and arrays often have more than one child. For delta trees that mostly consist of a few nodes, this decision can often be a disadvantage. We created a cost model and extended it by these implementation-dependent factors. Additionally, we added a sample step to our system before deciding which execution method, delta trees or complete materialization, to choose. The transformation is performed for $\leq 1\%$ of documents with both execution methods. Then the memory requirement of these documents

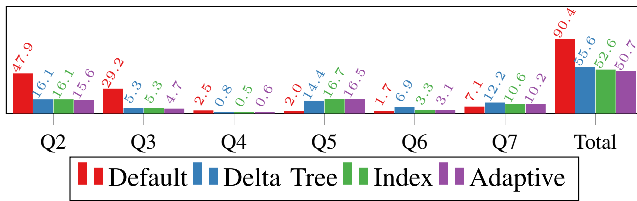


Fig. 4: Runtime of different execution methods (in s)

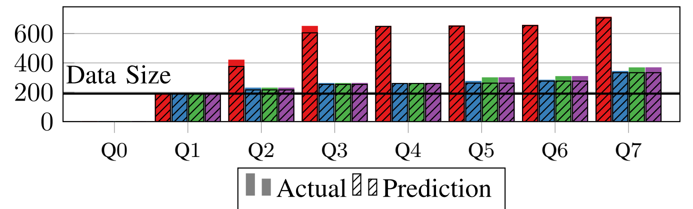


Fig. 5: Memory consumption (GB) of different execution methods

```
Q1: LOAD t1 FROM FILES "/data/twitter";
Q2: LOAD t1 CHOOSE EXISTS('/user')
    AS *,('/user/v1':1) STORE t2;
Q3: LOAD t2 AS *,('/user/v2':2) STORE t3;
Q4: LOAD t3 AGG ('':SUM('/user/v1')) STORE a;
Q5: LOAD t3 AS ('':MEMCOUNT('/user')) STORE c1;
Q6: LOAD t3 AS ('':MEMCOUNT('/user')+1) STORE c2;
Q7: LOAD t3 AS ('':'/user') STORE user;
```

Listing 1: Queries iteratively changing an object and reading it

is calculated and the method with the lowest requirement is chosen. This decision is performed for chunks of similarly shaped documents in our system.

V. Experimental Evaluation

A. Settings and Data and Workloads

The experiments are executed on a machine with 4 Xeon E7-4830 CPUs, each having 12 cores—and 24 threads—with 2.1 GHz, and 1 TB of RAM. The data is stored on one HGST Ultrastar 7K4000 HDD. Ubuntu 16.04.3 LTS is used as the underlying operation system. The described delta tree approach and optimizations are implemented as extensions to JODA [3].

The **dataset** used is a 109 GB file containing a sample of the raw Twitter JSON stream[2]. It consists of 29,634,708 JSON documents, where each document has between 7 and 348 attributes, containing every JSON type. The documents are split into two major groups. Around 23.5 million (79.33%) documents are normal tweets, while around 6.1 (20.67%) million documents are deletion instructions. The tweets have a varying number of attributes, depending on their status, e.g., retweets and favorites, while the deletion documents consist of seven attributes.

B. Multi-layer Delta Hierarchy Creation and Shared Reads

In the first evaluation, we execute a number of queries that illustrate the core features of our approach. The first query in Listing 1 loads the Twitter dataset. Then a collection is created which adds one member to the user object of the previous dataset. Derived from this collection, another attribute is added to the user object. In the following query, only the data added in Q2 is used in an aggregation. Then the member count of the user object is queried in the next two queries. The last query copies the shared user object into a new collection.

We compare our introduced approaches against the default execution method, which copies and modifies the full JSON documents. The delta tree approach is based on our implementation within the system, as explained in Section IV. The index approach uses the same implementation, but with enabled virtual object indexing, as described in Section IV-D. Lastly, the adaptive approach—described in Section IV-E—is compared.

The query time plot in Fig. 4 is omitting the first data import query, as it is unaffected by the execution method and requires the same time for all of them. As we can see in Fig. 4, the implementation without delta trees, which transforms the documents by copying the source data, requires for queries Q2–Q7 about 90.4 seconds. The delta tree implementations on the other hand executed these queries significantly faster, with 55.6, 52.6, and 50.7 seconds in total respectively. Queries which modify the documents, by either adding or removing attributes, or reading single atomic attributes, perform a multitude faster with each of the delta trees approaches, compared to the default execution method. On the other hand, queries reading objects, which are distributed over multiple delta trees are unsurprisingly always faster for the default execution method. We can improve the runtime of such queries by using the introduced index and adaptive execution methods.

Fig. 5 immediately shows the main advantage of delta trees. All implementations start with the same memory usage after the data import query Q1. For Q2 and Q3, the default implementation has to copy the data it wants to modify. This of course has a heavy impact on memory usage. The delta tree implementations on the other hand only have to store the additional data, with a little overhead for the internal tree structure.

When using the object index, memory usage is increased by up to 9% compared to the normal delta tree implementation. But compared to the baseline data ($190GB$ in-memory), delta trees with indexes only required an additional 87%, while copying the data increases the memory usage by 274%.

We can also see, that of our adaptive approach predicts the required memory usage with satisfactory precision to correctly choose the right execution method. It will choose the delta tree approach for queries Q2 and Q3, while using the default execution method for all other queries, hence reducing the execution times.

C. Adaptive Execution Method

In this experiment, we evaluate the capabilities of the adaptive execution method. The queries shown in Listing 2, once again, import our twitter dataset and then replace the user IDs with a hash value. Queries Q2 and Q3 replace the user IDs in the differently structured tweet and delete objects and

[2]https://developer.twitter.com/en/docs/labs/sampled-stream

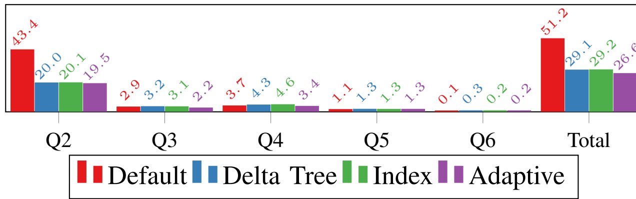


Fig. 6: Runtime of queries for different configurations (in s)

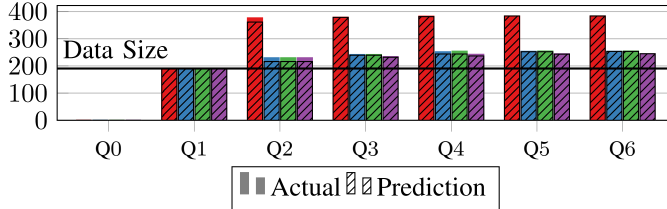


Fig. 7: Memory consumption (GB) of queries for different configurations

store the adapted documents in the same collection. Q4 and Q5 then extract these hashes into their own documents with only the hash value as the root. Lastly, Q6 aggregates all hash values to find the minimum and maximum.

Fig. 6 shows the runtime of each of these queries executed by all introduced approaches. As before we omit the parsing step as it is uninteresting for this evaluation. In Q2, we replace the user id of all tweet documents. These documents are large and hence, require a lot of time alone for copying the source documents before modifying them in the default execution. All delta tree implementation require less than half of the query time to evaluate this query.

Query Q3 does the same for the delete documents. They are very small and there are only a few of them in the dataset. Hence, this query is evaluated fast. Here the delta tree approach is slightly slower, even for a modifying query. This is the case, because of the nested structure with only a few attributes, for which the implementation allocates placeholder memory. The adaptive algorithm correctly predicts this situation and chooses the default execution method.

Queries Q4–Q5 now read the created datasets and extract the hash values. For Q4, the default and adaptive execution methods are again faster, as the model correctly predicts an advantage for this query. The delta tree approaches are slightly slower.

In total, the delta tree approaches are faster as the default execution method, as the complete document set does not have to be copied. The adaptive approach improves this total

```
Q1: LOAD t1 FROM FILES  "/data/twitter";
Q2: LOAD t1 CHOOSE !EXISTS('/delete')
    AS *, ('/user/id':HASH('/user/id')) STORE hashed;
Q3: LOAD t1 CHOOSE EXISTS('/delete')
    AS *, ('/delete/status/user_id':
        HASH('/delete/status/user_id')) STORE hashed;
Q4: LOAD hashed AS ('':'/user/id') STORE hashes;
Q5: LOAD hashed
    AS ('':'/delete/status/user_id') STORE hashes;
Q6: LOAD hashes AGG ('/min':MIN('')), ('/max':MAX(''));
```

Listing 2: Hashing user IDs in different documents

runtime, as it predicts the advantages of the default execution for some queries.

The memory requirements of all approaches for the given queries is shown in Fig. 7. Once again, we can see that delta trees require less memory for large documents where only parts of it are changed. The adaptive implementation chose to only execute Q2 with the delta tree execution method, while the remaining queries are executed with the default method. This results in the lowest memory consumption of all tested execution methods.

VI. Conclusion

In this paper, we introduced the concept of delta trees, for materializing only the differences of a document transformation, to reduce the memory footprint of exploration systems. We explained the basic idea, theoretical model, and specific implementation details, based on our in-house JSON exploration tool JODA. Additionally, we introduced improvements to the systems to mitigate the performance bottlenecks introduced by the approach. Delta trees enable systems to perform queries, with a fraction of the required memory, leaving original datasets intact. An adaptive algorithm was introduced, which helps to avoid the drawbacks of this approach. This increased transformation performance is bought, by sacrificing performance in some read-heavy operations, especially copying shared objects. We mitigated this performance loss by creating a special index for these shared objects. This increased the read performance, at cost of slightly higher memory usage.

References

[1] T. Bray, "The javascript object notation (json) data interchange format," Internet Requests for Comments, RFC Editor, STD 90, December 2017.
[2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," *SIGMOD 2012*.
[3] N. Schäfer and S. Michel, "JODA: A vertically scalable, lightweight JSON processor for big data transformations," *ICDE 2020*.
[4] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *J. Parallel Distrib. Comput.*, vol. 111, pp. 162–173, 2018.
[5] K. Tai, "The tree-to-tree correction problem," *J. ACM*, vol. 26, no. 3, pp. 422–433, 1979.
[6] P. Bille, "A survey on tree edit distance and related problems," *Theor. Comput. Sci.*, vol. 337, no. 1-3, pp. 217–239, 2005.
[7] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *SIGMOD 1996*.
[8] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet, "Change-centric management of versions in an XML warehouse," *VLDB 2001*.
[9] H. Su, D. Kramer, L. Chen, K. T. Claypool, and E. A. Rundensteiner, "XEM: managing the evolution of XML documents," *Eleventh International Workshop on Research Issues in Data Engineering: Document Management for Data Intensive Business and Scientific Applications, 2001*.
[10] Y. Wang, D. J. DeWitt, and J. Cai, "X-diff: An effective change detection algorithm for XML documents," *ICDE 2003*.
[11] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden, "Materialization strategies in a column-oriented DBMS," *ICDE 2007*.
[12] G. Nicol, M. Champion, J. Robie, A. L. Hors, L. Wood, R. S. Sutor, S. Isaacson, C. Wilson, S. B. Byrne, and I. Jacobs, "Document object model (DOM) level 1," W3C, W3C Recommendation, Oct. 1998, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/.
[13] M. Dyck, J. Spiegel, and J. Robie, "XML path language (XPath) 3.1," W3C, W3C Recommendation, Mar. 2017, https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

[14] P. Bryan, K. Zyp, and M. Nottingham, "Javascript object notation (json) pointer," Internet Requests for Comments, RFC Editor, RFC 6901, April 2013. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6901.txt

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.