Partially Materializable Delta Trees for Efficient Data Wrangling of Semi-Structured Contents

Nico Schäfer
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
nschaefer@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
michel@cs.uni-kl.de

ABSTRACT

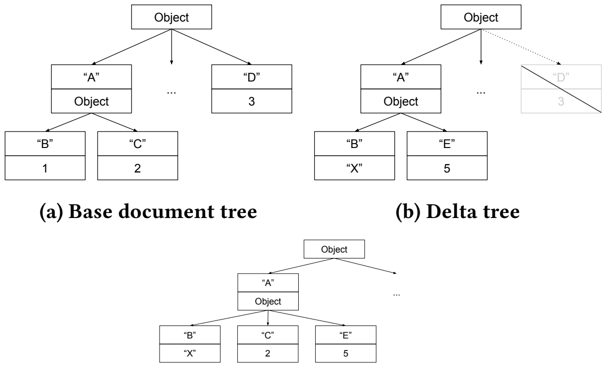
In this paper, we propose delta trees to boost efficiency and reduce storage requirements of iterative data exploration and data wrangling tasks over massive, semi-structured datasets. During such tasks, data is filtered, projected, joined, and converted in multiple successive or independent steps, driven by data scientists or higher-level applications. While the original datasets can often not be disposed, delta trees are necessary to represent only the changes to the original data, instead of creating largely redundant copies. With delta trees, we are able to reduce storage requirements and query execution time for various data manipulation operations, while maintaining acceptable query times for others. We report on a first experimental study over a dataset of Twitter tweets, showing that the expected vast savings of storage consumption can be enjoyed with negligible computational overhead compared to a full data duplication.

1 INTRODUCTION

In recent years, the interest in semi-structured file formats steadily increased. Arguably, one of the most visible data formats is JSON, which eliminates the need to force data into relations and is designed to be human-readable. It has been adopted by various platforms and systems, for instance, for data exchange through APIs, to store system access logs, or configuration files. Common operations in semi-structured document processing are adding, removing, and moving values. Consider for instance the case of data scientists working on a large sample of Twitter tweets. While some 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 a novel way to store changes made to original documents, leading to a vastly reduced memory footprint and an even improved querying performance for some query types, while having a modest performance overhead for others.

1.1 Sketch of the Approach and Related Work

The concept of late materialization [1] is used to push back materializations of (intermediate) results until they are needed. For this to work, the system has to store the queries, or similar information, and the base data. The result is then calculated on demand as soon as it is required.



(a) Base document tree

(b) Delta tree

(c) Combined document

Figure 1: Example of base document with a delta tree

Our approach also aims to reduce the amount of unnecessary transformations of the base data. But instead of storing the operations that lead to the result, we calculate the difference (or delta) of the transformations, compared to the base data and only store this new information, with a reference to the base document. Almeida *et al.* already proposed a map as Conflict-free Replicated Data Type (CRDT) [2], 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. Generally, the concept of extracting or storing deltas from semi-structured documents has been investigated before [3, 6, 7]. It has also been applied in the context of relational database systems to compact historical data [4, 5]. However, in contrast, we store the changes made by incremental data modification operations, in order to improve the memory consumption.

2 DELTA TREES

Figure 1a visualizes the tree representation of one semi-structured document. It consists of (nested) objects containing attributes, with atomic data. Now, a query may transform this base document by deleting the "D" attribute of the root object, changing the atomic data within the "B" attribute, and adding an additional member to the "A" object. Instead of storing the complete transformed document, we only store the changed parts of the tree, as shown in Figure 1b. In this example, the changed "B" member and additional "E" attribute is stored, together with the required parent structure (in this case the root node and "A" object).

Additionally, we keep track of all paths within the tree that have been changed. A path is an ordered list of tokens, identifying all nodes that have to be traversed to arrive at a desired destination. In this example, the changed paths would be “/A/B”, “/A/E”, and “/D”. This information is sufficient to reconstruct the complete transformation result as shown in Figure 1c.

The benefits of this approach may not be immediately evident from the given example. But in real world data sets, documents having hundreds of attributes, distributed over many nested objects, are very common. If a transformation query only modifies a couple of nodes, storing the complete transformation result clearly is unnecessary—and modifying the original data is often prohibited.

2.1 Reconstructing Combined Documents

Given a base tree B and a delta tree D, with the overwritten paths P, we may need to construct the complete result document tree R to return to the user. This is achieved by simultaneously iterating through both trees in a depth-first manner. Starting at the root of the trees, for every unique path p in both trees, there are now five possibilities:

1. $\exists n_B \in B$ and $\exists n_D \in D$ belonging to p
   (a) $p \in P \Rightarrow$ the path is overwritten in D, hence, we only continue traversing n_D for this sub-tree.
   (b) $p \notin P \Rightarrow$ the path is shared between B and D, thereby, we continue traversing both n_B and n_D.
2. $\exists n_B \in B$ and $\nexists n_D \in D$ belonging to p
   (a) $p \in P \Rightarrow$ the path was removed by the transformation and we do not continue traversing this sub-tree.
   (b) $p \notin P \Rightarrow$ the node is not materialized, but still valid, thus, we continue traversing n_B.
3. $\nexists n_B \in B$ and $\exists n_D \in D$ belonging to p, it must thereby be added by D and we only traverse n_D.

The result is constructed by traversing B and D as above and adding all visited nodes to R.

2.2 Delta Hierarchies

There can be multiple delta trees, which reference the same base tree, in which case the memory savings introduced by delta trees are magnified. Additionally, delta trees may be based on other delta trees. Given $(B, D_1, ..., D_i, ..., D_n)$, where B is the base document, and D_i are delta trees, each based on the previous tree. This is called a *delta hierarchy.* The result document R_1, of B and D_1 may be constructed as shown in Section 2.1. To construct the result document R_i, the same algorithm would then be executed with R_{i-1} as base tree and D_i as delta tree. In case of larger hierarchies, this naïve execution is suboptimal. Instead, we perform the reconstruction algorithm for all trees at the same time, by traversing the whole delta hierarchy simultaneously. For each unique path p in the delta hierarchy, starting at the root: If p exists only in one tree, we follow only this subtree. If p exists in multiple trees, we use the value of the uppermost delta tree, in case of atomic values. In case of objects, whose child attributes are distributed over multiple tree, we continue traversing all affected trees.

3 ARCHITECTURE AND ALGORITHMS

We implemented our approach in our in-house JSON exploration system JODA, written in C++. This system uses the RapidJSON (http://rapidjson.org/) parser, which uses a DOM-tree in-memory representation to store the parsed JSON documents—the extension to XML and YAML is straightforward.

Documents provided to the system are organized in so-called collections, for instance, a collection of Twitter tweets and a collection of blog posts. All documents within a collection are organized into a number of containers. Each container is a self-contained unit, which includes all required information to perform a query upon. After creation, these containers are immutable to make the query execution free of any synchronization overhead.

Queries are simple PIG-style sequences of commands. To begin, a collection can be chosen and data can be imported into the system by the `LOAD` step. This data is then passed to the `CHOOSE` command, which may filter the data depending on a given predicate. The filtered documents may then be transformed in the `AS` step, by using an arbitrary amount of transformation instructions, in the form of `(<destination>:<source>)` tuples, where the destination is a path in the new document, and the source can be any supported function or a path in the base document. The transformed documents are then passed to the `AGG` command for aggregation and may finally stored in a collection or exported into a file with the `STORE` expression.

3.1 Construction

The `AS` instruction also supports the special `*` operator, which copies the whole source document. This operator may be combined with additional transformations to change the source document. If the system detects this combination, delta trees may be used to perform this transformation.

In this case, the system will first create the support structure, by instantiating an empty delta tree with a pointer to the base document, which may also be a delta tree. Each additional transformation is then executed and the result materialized in the newly created delta tree. Additionally, the `<destination>` pointers form the explicitly overwritten paths and are stored with the tree, as described in Section 2. These paths will be the same for all delta trees that are created by a given query. We can therefore store them once in the—previosuly introduced—container class, as it includes all data that may be shared by its documents. Now, a delta hierarchy is given by following the base document pointers from any given delta tree to the bottom.

3.2 Reconstruction

To reconstruct the result document we created a visitor class, which implements the idea described in Section 2.1–2.2. This visitor class simultaneously traverses the delta hierarchy as described. This basic visitor class is then extended to fulfill different needs, like materializing the final result documents or accessing specific subtrees for query evaluation.

Our system preserves the order of members within a JSON object, which is not required by the standard itself. Thereby we have to adapt the traversing algorithm described previously as follows:

Algorithm 1 reconstructs a result document, given a delta hierarchy. Our implementation handles the base document as just another delta tree D_0 which overwrites the whole tree (root path ' '). The algorithm visits a node n, which at the beginning is the root node of D_n. If the given node is shared, i.e., it is an object or array which is distributed over multiple delta trees, then we first search the base document. The base document is the document which has overwritten this node last. Starting from the base, we collect all members of this shared node in all delta trees. The algorithm is then repeated for each of these members.

If the node is not shared, then we retrieve the upper most instance of the node in the delta hierarchy and visit this node.

```
Data: n = root($D_n$); p = ' '; $D_0, ..., D_n$; $P_0, ..., P_n$
if IsShared(n,$P_1, ..., P_n$) then
    Base = GetBaseDeltaTree(n);
    for i = Base to $D_n$ do
        members += GetMembers($D_i$,$P_i$);
    end
    for member in members do
        Recurse(member,p+'/'+member.id,$D_0$,...);
    end
else
    $n_D$ = GetBaseNode(n);
    Visit $n_D$;
end
```

Algorithm 1: Reconstruction algorithm

The visit function belongs to the given visitor, which may, as explained previously, perform different actions for the given node.

3.3 Estimating Memory Usage

The main advantage of delta trees is the reduced memory requirement. We define the memory cost of a tree, as the sum of all costs of its nodes. The cost of a node is given by the byte size of its in-memory representation. Evidently, each delta tree is smaller or equal in size as the result tree, given by combining the base with the delta tree, 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.

In reality, this is often not the case. For instance, the RapidJSON library, that we use 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 proved to be a disadvantage. Each placeholder child will be added to the cost, which for some queries and documents may be more than materializing the result.

We thereby 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 is calculated and the method with the lowest requirement is chosen. This decision is performed on per-container basis in our system, which mostly contain similar documents.

3.4 Accessing values

Queries in our system may use a wide range of functions to evaluate values. The parameters of functions may be nested functions or retrieved directly from the document. Hence, functions have to be able to access all values contained in delta trees. For example, it is not trivial to evaluate the member count of an object, that is distributed over multiple trees. If a function accesses a value, specified by a given path, the system will check top-down for the given delta tree hierarchy if this specific path is shared. If not, the value can be directly passed to the function, without even traversing the whole delta hierarchy. If it is shared, we have to evaluate this function by traversing the delta tree hierarchy as explained in Section 3.2.

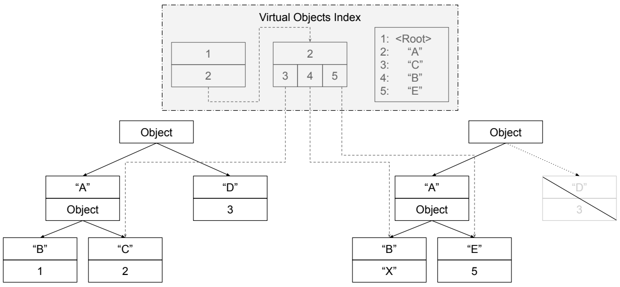


Figure 2: Virtual Object Index

The main cost of accessing values by path is traversing the tree. For each token in the path, the child represented by this token has to be found. For arrays, this is simple, as the token is the index of the child. Objects on the other hand, store their children in order of occurrence in the source document. Here a linear search with string comparisons has to be performed to find the specified child. This operation is the dominating cost factor for finding a value belonging to a path. For delta hierarchies this cost may be amplified, if many trees have to be searched. But if the value is overwritten in one of the higher trees in a delta hierarchy, this cost can be less than for one materialized result document.

4 PARTIAL MATERIALIZATION AND OBJECT INDICES

It may happen, that multiple queries repeatedly require the same value, which may be an object or array distributed over multiple delta trees. In these cases it can be beneficial to materialize the given object or array at the cost of increased memory usage. This is achieved by traversing only this sub-tree in the delta hierarchy, as described previously, and copying the whole sub-tree into the highest delta tree. We call this *partial materialization* of the result. By strategically materializing parts of the result we can still massively reduce the memory footprint while preventing the performance drawbacks of simultaneously traversing multiple trees.

To prevent the materialization of objects, we created an *virtual object index*. This index tries to combine the powers of delta trees with the 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 2. 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 is stored in the container and shared by many documents, thereby 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.

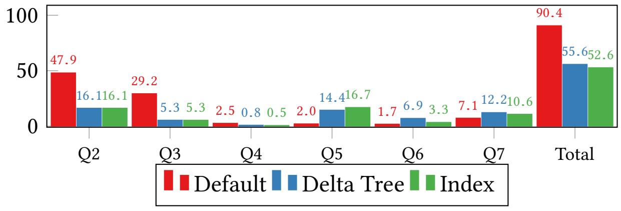


Figure 3: Runtime [s] of different execution methods

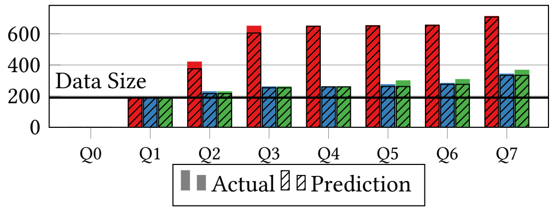


Figure 4: Memory consumption [GB] of different execution methods

5 EVALUATION

In this section, we will evaluate the performance and memory footprint of our delta tree implementation. The data set used is a 109 GB selection of the Twitter JSON stream[1]. 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.

The tests are executed on a machine with 4 Xeon E7-4830 CPUs, each having 12 cores—and 24 threads—with 2.1 GHz. Furthermore, 33 RAM-Kits, each having 32 GB of memory at 2400 MHz, are included, providing the server with around 1 TB of RAM. The data is stored on one HGST Ultrastar 7K4000 HDD, with 7200 RPM. Ubuntu 16.04.3 LTS is used as the underlying operation system.

The first query in Listing 1 loads the Twitter data set. Then a collection is created which adds one member to the user object of the previous data set. 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.

```
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

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 3. The index approach uses the same implementation, but with enabled virtual object indexing, as described in Section 4. The query time plot in Figure 3 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, for queries Q2 and Q3, the delta tree approaches have a strong advantage over the default execution method. While the default execution copies the whole Twitter data set, the delta tree approaches only require one reference to the base document and the `/user/v<x>` values, with the supporting tree structure. This results in an increase of maximum 37.24GB and 31.89GB to the previous query for Q2 and Q3 respectively, as can be seen in Figure 4. The default approach on the other hand increases its memory consumption by 228GB and 230GB. As copy operations are the dominating cost for these queries, the execution times of the delta tree methods is also significantly lower.

In Q4 the value written in Q2 is read. This is fast and memory unintensive for all approaches, but the delta tree approaches are faster, as the value can be read very fast in one of the delta trees without traversal. In Q5 and Q6 the user object, which is distributed between three delta trees is used. For the default execution method this is fast and requires nearly no memory. For the normal delta tree method, this operation is slow, as the whole delta hierarchy has to be traversed. The index introduces additional overhead, as it creates the virtual object indices. This results in vastly improved query times, but increased memory consumption in Q6, which brings it closer to the default implementation, while the delta tree execution is much slower. In Q7 the modified user object is materialized to a new document. This is relatively fast for the default execution method, but once again slow for the delta tree. The indexed method can use the virtual objects to improve its query time. All in all are the delta tree implementations faster and very similar for this query set. But if it would contain additional read queries this situation could quickly change.

6 CONCLUSION

In this paper, we introduced the concept of delta trees, for materializing only the differences of a transformation, to reduce the memory footprint of exploration systems. We explained the basic idea 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. As we have seen, delta trees enable systems to perform the same set of queries, with a fraction of the required memory.

REFERENCES

[1] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. 2007. Materialization strategies in a column-oriented DBMS. ICDE 2007, 466–475.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162–173.

[3] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change detection in hierarchically structured information. *Sigmod Record*, Vol. 25, 493–504.

[4] Sándor Héman, Marcin Zukowski, Niels J Nes, Lefteris Sidirourgos, and Peter Boncz. 2010. Positional update handling in column stores. *SIGMOD 2010*, 543–554.

[5] David B Lomet and Feifei Li. 2009. Improving transaction-time DBMS performance and functionality. *ICDE 2009*, 581–591.

[6] Amélie Marian, Serge Abiteboul, Gregory Cobena, and Laurent Mignet. 2001. Change-centric management of versions in an XML warehouse. *VLDB 2001*, Vol. 1. 581–590.

[7] Hong Su, Diane Kramer, Li Chen, Kajal Claypool, and Elke A Rundensteiner. 2001. XEM: Managing the evolution of XML documents. *11th International Workshop on Research Issues in Data Engineering (RIDE 2001)*, 103–110.

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