Efficient Updates for Worst-Case Optimal Join Triple Stores
—
Complexity Analysis

Alexander Bigerl[1][0000−0002−9617−1466] (✉), Nikolaos Karalis[1][0000−0002−0710−7180], Liss Heidrich[1][0009−0006−2031−2548], and Axel-Cyrille Ngonga Ngomo[1][0000−0001−7112−3516] (✉)

Data Science Group (DICE), Heinz Nixdorf Institute, Paderborn University
`{alexander.bigerl, nikolaos.karalis, liss.heidrich, axel.ngonga}@uni-paderborn.de`

In the following we provide asymptotic runtime bounds for applying a change set of size z to a depth-d hypertrie as described in [2]. We refer to this hypertrie as *optimized hypertrie* in contrast to *baseline hypertrie* from [1].

Therefore, we first prove that the optimized hypertrie does not increase the space complexity compared to the baseline hypertrie. We then show that the space complexity is a surrogate for the update time complexity. We want to note that the provided space complexity for the *optimized hypertrie* is not proven to be sharp. Rather, since the constructed entry set from the proof for the *baseline hypertrie* requires less memory in the *optimized hypertrie* we assume that the sharp-bound is tighter. If the bound is tighter this also translates to the update time complexity.

A Baseline and Optimized Hypertrie Space Complexity

Lemma 1 (Baseline hypertrie space). *A depth-d* baseline hypertrie *has a space complexity of $\mathcal{O}\big(z \cdot d \cdot 2^{d-1}\big)$.*

Proof. The proof is provided in the supplementary material[1] of [1].

Lemma 2 (Optimized hypertrie space). *The space requirement of a* optimized hypertrie *is in the same complexity class as the* baseline hypertrie*.*

Proof. We show that each optimization of the *optimized hypertrie* adds memory bound $\mathcal{O}\big(z \cdot d \cdot 2^{d-1}\big)$ to the hypertrie. Note that reducing the memory used also fulfills this condition.

1. Storing each node of the *optimized Hypertrie* in the *node context* together with its reference count and its identifier is done in constant space per node when using a hash map. Since node count induced space complexity $\mathcal{O}\big(z \cdot (2^d - 1)\big)$ is equivalent to $(d = 1)$ dominated by $(d > 1)$ edge count $\mathcal{O}\big(z \cdot$

[1] Supplementary PDF

$2^{d-1} \cdot d)$ induced space complexity, adding constant space to every node does not increase the overall space complexity.

2. Nodes in the *baseline hypertrie* are de-duplicated if their access path structurally ensures that they encode equal tuple sets. The *de-duplication of equivalent nodes* utilized in the *optimized hypertrie* is a generalization of this method. It de-duplicates at least the nodes that are deduplicated by the *baseline hypertrie*. Thus, the *optimized hypertrie* eliminates at least the same amount of duplicate nodes as the *baseline hypertrie*.
3. A single entry node (SEN) in an *optimized hypertrie* at depth d requires at most d space for storing a d-tuple. This space requirement is equivalent to the d child node references of a full node (FN) encoding the same singleton tuple set that is used in a *baseline hypertrie* instead. The SEN has no child nodes whereas the FN has child nodes. So, the SEN requires no more memory than an FN.
4. In a *optimized hypertrie*, all depth-1 singleton node payloads are stored in the memory slot of their respective parents' child mappings. This requires no additional memory compared to the *baseline hypertrie*. This means, a *optimized hypertrie* never has stand-alone depth-1 nodes with a single entry. If a *baseline Hypertrie* has such nodes, the *optimized hypertrie* for the same entry set requires strictly less nodes.

Because the optimisations operate on disjoint structural aspects (deduplication, nodes with depth > 1 that encode a single entry, edge mappings of nodes with depth 2), their individual upper bounds add rather than multiply.

Since no optimizations require asymptotically more resources, we conclude that the space complexity of the *baseline hypertrie* holds for the *optimized Hypertrie*.

B Optimized Hypertrie Update Time Complexity

Lemma 3 (Adding or removing a tuple).
Adding a single tuple to a baseline or optimized hypertrie node adds at most $(2^d - 1)$ *nodes and* $(2^{d-1} \cdot d)$ *edges. Removing a single tuple from a baseline or optimized hypertrie with* z *entries leaves the hypertrie with at most* $(z-1) \cdot (2^d - 1)$ *nodes and* $(z-1) \cdot (2^{d-1} \cdot d)$ *edges.*

Proof. This can be directly concluded from Lemma 1 in the supplementary material[1] of [1].

Lemma 4 (Constant-time edge operations). *Lookup, insertion, and deletion in a child mapping take amortized constant time.*

Proof. Single entry lookup, insertion, and deletion take amortized constant time in a hashmap. Since hashmaps are used for child mappings, these operations take constant time there.

Definition 1 (Surrogate hypertrie). *A surrogate hypertrie for an insert or deletion update operation is the hypertrie s that is populated from the changeset that is to be added to or removed from a hypertrie h.*

Corollary 1 (Optimized hypertrie update time complexity). *The time complexity of applying a change set of z tuples to a* optimized hypertrie *is bound by the space complexity of the surrogate hypertrie, i.e.* $\mathcal{O}(z \cdot 2^{d-1} \cdot d)$.

For the proof we first construct the surrogate hypertrie and then apply the changes recursively. In the algorithm proposed in the paper, this process is separated into level-wise phases to save space and optimize for cache locality in practice. This adjustment is irrelevant for asymptotic complexity.

Proof. In the following we show that the space required by the nodes of the surrogate hypertrie is an upper bound for the time required to apply an update. By lemma 2, the surrogate the space complexity of a surrogate hypertrie for z tuples is $\mathcal{O}\big(z \cdot 2^{d-1} \cdot d\big)$. We apply updates recursively starting at the root node of the hypertrie being updated and the surrogate hypertrie. The changeset is the set of tuples encoded by the surrogate hypertrie. Recursive calls are issued for child nodes of the surrogate hypertrie thus it ends at the leaf nodes that do not have child nodes. Since we have shown in lemma 2 that SEN and in-place stored nodes do not increase the space complexity, we can assume that all nodes are FNs. SEN and in-place will at most require the same runtime. Note that this proof leverages the optimization that generalizes access-path based de-duplication to full de-duplication of equivalent nodes.

We first consider the cost within a recursive step. This includes issuing the recursive steps but not the execution of the recursive steps. For adding and removing tuples the following cases need to be considered:

1. Adding z_h tuples to a depth-d_h hypertrie node h requires adding or updating a child mapping, and recursively inserting into a child node or creating a new child node for each child mapping in the surrogate hypertrie. The surrogate hypertrie child mappings are exactly for the respective tuple positions and key parts that must be either inserted or updated at h to apply the insertion. Changing the child mappings of h and issuing the recursive calls requires no more constant-time steps than outgoing edges at the corresponding surrogate hypertrie node exist (at most $e = d_h \cdot z_h$). The changesets for updating child nodes in the recursive steps can be computed in a linear scan for each child node map. This requires at most e accesses and write operations. Thus, adding to a hypertrie node requires time equivalent to the space requirement of the corresponding surrogate hypertrie node by a constant factor.
2. Creating a node is done by inserting into a empty hypertrie node. item 1 applies.
3. Removing z_h tuples from a depth-d_h hypertrie node h is widely analogous to inserting tuples. It differs in requiring to update or remove a child mapping, and to recursively remove tuples from a child node or to delete a child node for each child mapping in the surrogate hypertrie. Otherwise, the same

argument as for item 1 applies to the number of required calls to execute these steps.

4. If a depth-d_h hypertrie node h with z_h tuples is deleted, it is by construction equivalent to its surrogate node, i.e., both nodes share the same child node mappings. Thus, issuing a recursive call per child child mapping and deallocating takes time equivalent by a constant factor to the number of stored edge mappings in the surrogate node.

Note that the surrogate hypertrie applies de-duplication of equivalent nodes. As shown above, issuing recursive calls for a surrogate hypertrie each time it is referenced is covered. Since the each unique node is present only once in the surrogate hypertrie we must guarantee that the node resulting from the update is only created once, or, that it is deleted only once. This is done by inserting a de-duplication operation before that conditions the execution of the recursive step. Creating a node, adding tuples and removing tuples are executed only if no node for the corresponding identifier exists yet. Otherwise, the step is completed with incrementing the reference count for the node. For node deletions, the reference count is decremented. Only if it reaches zero, the actual step is executed. Otherwise, the step is completed. These precondition tests run in constant time and guarantee that an update operation is run only once per surrogate hypertrie node. Because each check is executed exactly once per child-mapping in the surrogate hypertrie (and there are $\mathcal{O}\big(z \cdot 2^{d-1} \cdot d\big)$ of those), the total overhead of the checks is absorbed in the claimed bound.

Thus, an update takes no more time than the nodes in the surrogate hypertrie have child mappings and runs in $\mathcal{O}\big(z \cdot 2^{d-1} \cdot d\big)$.

References

1. A. Bigerl, L. Conrads, C. Behning, M. Sherif, M. Saleem, and A.-C. Ngonga Ngomo. *Tentris – A Tensor-Based Triple Store.* Proc. ISWC 2020.
2. A. Bigerl, L. Conrads, C. Behning, M. Saleem, and A.-C. Ngonga Ngomo. *Hashing the Hypertrie: Space- and Time-Efficient Indexing for SPARQL in Tensors.* Proc. ISWC 2022.