Efficient Updates for Worst-Case Optimal Join Triple Stores

—

Technical Report

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

A Bulk-Loading Performance and Storage Efficiency

Table 1. Loading metrics for loading the datasets DBpedia and Wikidata into the different triple stores. Bytes/Triple is the storage requirements in Bytes per triple on disc and Triples/Second is the average bulk loading speed.

Dataset	triple store	Bytes/Triple	Triples/Second
DBpedia	Fuseki	134	169409
	GraphDB	76	185897
	Oxigraph	265	241640
	TENTRIS	160	112321
	TENTRIS-ID (ours)	161	109671
	Virtuoso	53	54700
Wikidata	Fuseki	141	224526
	GraphDB	58	247712
	Oxigraph	150	314915
	TENTRIS	119	156686
	TENTRIS-ID (ours)	119	157671
	Virtuoso	37	295950

B Hypertrie

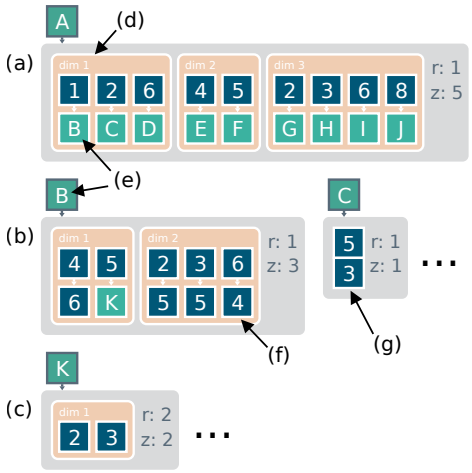
The hypertrie [2,3] generalizes tries [4] to provide access in any collation order. An example of a hypertrie is given in Figure 1. It organizes data into a leveled, depth-d directed acyclic graph, where paths encode d-tuples. In a trie, each node has a child for every unique element e found in the first position of the tuples it encodes. Each child node encodes the remaining portion of tuples starting

```
:X foaf:knows :Y .
:X foaf:knows :Z .
:Y foaf:knows :Z .
:X a :Unicorn .
:Unicorn a rdf:Class .
```

(i) Example RDF graph in Turtle format (prefixes and base IRI omitted for brevity).

Subject	Predicate	Object
1	5	2
1	5	3
2	5	3
1	4	6
6	4	8

(iii) ID representation of listing (i).



(ii) Nodes from a hypertrie encoding (iii).

Fig. 1. The RDF graph from listing (i) is encoded into integer triples, as shown in table (iii), which are represented by the hypertrie partially depicted in figure (ii). Nodes (a), (b), and (c) are hypertrie full nodes at depths 3, 2, and 1, respectively. The child mapping $c_1^{(h)}$ of the hypertrie node h in (a) is highlighted in (d). (e) illustrates how node identifiers are used for referencing. (f) shows an example of a depth-1 single-entry node stored directly in its parent's child mapping, while (g) depicts a depth-2 single-entry node. `r` and `z` indicate the nodes' reference count and number of encoded tuples, respectively.

with e, and the edge connecting the parent node to the child is labeled with e. The hypertrie extends this concept. A depth d full node (FN) h represents a set of d-tuples $T(h)$ and has child mappings $c_p^{(h)}$ for every tuple position p (dimension), not just the first one. The hypertrie leverages a homomorphic hash [1, 5–7] function $i(\cdot)$ to assign each node h an unique identifier $i(T(h))$. Child node mappings refer not to child nodes directly but to these identifiers. We use $\oplus$ and $\ominus$ for the homomorphic equivalents of set[1] union and set minus, e.g., $i(\{1\}) \oplus i(\{2\}) = i(\{1\} \cup \{2\})$. For singleton sets, the hypertrie uses an optimized node type called *single entry node* (SEN), which does not have child mappings and fully encodes its tuple. Depth-2 FNs store SEN child nodes directly *in-place* of their identifiers within the child mapping. Formally, we define the hypertrie[2]:

Definition 1 (Hypertrie). *Let A be a fixed alphabet and let $H(d)$ with $d \geq 0$ be the set of all hypertries with depth d. We define $H(0) = \{\texttt{true}\}$. For a hypertrie*

[1] On sets, homomorphic hashing requires union operands to be distinct and the subtrahend in set difference to be a subset of the minuend.

[2] We use a simplified hypertrie definition limited to sets, compared to the original, which could also represent maps.

full node (FN) $h \in H(d), d > 0$ *and a dimension* $p \in \{1 \dots d\}$*, the outgoing edges are declared by the partial mapping* $c_p^{(h)} : A \nrightarrow \mathbb{I}$ *where the domain is the edge label and the range is the child node's identifier. d-tuples that are elements of* $H(d)$ *are called single entry nodes (SEN) and have no associated mappings* $c_p^{(h)}$*. We refer to* $|T(h)|$ *as size of* h*.*

Hypertrie nodes are organized in a *hypertrie context*. It deduplicates them using their identifiers and counts how often each node is referenced from parent nodes. Only referenced nodes are stored. Declaring a hypertrie as *primary* increments its reference count by 1, preventing it from being removed. In this work a hypertrie context has exactly one primary hypertrie that spans all other nodes.

Definition 2 (Hypertrie Context). *Let* Λ_0 *be the set of all hypertries including empty hypertries. A hypertrie context is a tuple* (P, m, r) *composed of a set of hypertrie nodes* $P \subseteq \Lambda_0$*, a partial mapping from identifiers to these nodes* $m : \mathbb{Z} \nrightarrow P$ *and another mapping that assigns a reference count to each node* $r : P \to \mathbb{N} \cup 0$*. The notation* $C.m(id)$ *denotes the node identified by id, and* $C.r(n)$ *represents the reference count of node* n*.*

C Hypertrie Bulk Updates

We propose an algorithm for bulk updates to the hypertrie. Each update exclusively consists of either insertions or removals.[3] Updates that involve both insertions and removals must be split into two separate operations and executed consecutively. Formally, this algorithm transitions one hypertrie context C to another by modifying the tuple set represented by its primary hypertrie.

A hypertrie h can be in one of the following states prior to the insertion of a non-empty set of tuples t^+: (i) empty ($|h| = 0$), (ii) a SEN ($|h| = 1$), or (iii) a FN ($|h| > 1$). The change depends on the combined size $|h| + |t^+|$: (i) an empty hypertrie transitions to a new SEN if $|t^+| = 1$, or a new FN if $|t^+| > 1$, (ii) a SEN becomes a FN, or (iii) an existing FN is modified to include the new tuples. When removing a non-empty set t^- of tuples, the hypertrie h cannot be empty. If it is a SEN then h is the tuple[3] that is being removed resulting in an empty hypertrie. When removing tuples from a FN, the result depends on the number of tuples remaining ($|h| - |t^-|$). If exactly one tuple remains, the FN transitions to a SEN. If no tuples remain, h becomes empty. Otherwise, the FN is modified to exclude the removed tuples. Both insertions and removals on a FN trigger additional operations on child nodes (e.g., reference count adjustments, insertions or removals). In the following, when changing a hypertrie node h results in a new node h', we refer to h as the source and h' as the target.

Hypertrie nodes may be referenced by multiple parent nodes or even multiple times by the same parent. As a result, operations triggered on child nodes may require multiple, independent changes starting from the same source hypertrie h, with both identical and different sets of tuples. To eliminate redundancy,

[3] For simplicity, we assume only new tuples are inserted and existing ones removed.

changes that would create an already existing node (e.g., from prior operations or the original hypertrie context) are replaced with reference count increments. Changes involving different sets of tuples are applied to a copy of h to ensure h remains unmodified for subsequent operations. Note that h may or may not be referenced after all operations are completed. If only a single change depending on h remains and the reference count $r(h)$ will be 0 after all operations are applied, this change modifies h directly without creating a copy.

Our proposed algorithm operates in a level-wise, top-down manner, progressing through three phases at each level: *requesting updates*, *planning changes*, and *applying changes*. In the *first phase*, updates are requested for hypertrie nodes, including tuple insertions or deletions, node creations and reference count adjustments. In the *second phase*, these updates are translated into an efficient plan of changes. The *third phase* applies the planned changes, triggering further updates to child nodes as needed. This process repeats recursively, with phase 3 at one level overlapping with phase 1 at the next, except for the final step, which takes place at depth-1 hypertrie nodes. The algorithm begins with a single request to update the primary hypertrie.

C.1 Phase 1: Request Updates

In the first phase, update requests are broken down into two categories: (i) those targeting FNs and (ii) those targeting SENs. The following mappings are populated to plan and execute changes in subsequent phases:

`create_fns` tracks candidates for creating new FNs by mapping their identifiers $i(s)$ to their corresponding set of tuples s.

`change_fns` tracks changes where source and target nodes are FNs. Each change includes the source and target node identifiers, a set of tuples, and a tag indicating whether the set is inserted or removed. The mapping associates each source node identifier with a nested mapping of target node identifiers to ordered pairs of changed tuples and an operation tag $w \in \{\texttt{ins}, \texttt{rem}\}$.

`fns_delta` tracks changes to reference counts of FNs involved in a change. It maps each FN identifier to its reference count delta. A positive delta indicates an increase, while a negative delta indicates a decrease. For newly created nodes, the delta corresponds to their initial reference count.

`change_sens` tracks changes to SENs, such as reference count adjustments and tuple updates. It maps each SEN identifier $i(\{t\})$ to an ordered pair. The pair consists of the reference count delta and the corresponding tuple t.

The update requests are issued via four functions, detailed in Algorithm 1. Each function returns the identifier of the target node:

CREATE (ll.2) plans to create a new SEN or FN from a set of tuples or increments the reference count if such a node is already planned or exists already.

INSERT (ll.12) plans the insertion of a set of tuples into an existing node. The reference count of the source node is decreased because the insertion transforms the source into the target, and the source will no longer be referenced

Algorithm 1 Functions to request updates in phase 1

1: *Abbreviations: source (src), target (tgt), tuple (tpl), reference (ref), domain (dom)*
2: **function** CREATE(tpl_set) ▷ tpl_set: set of hypertrie tuples
3: node_id $\leftarrow i$(tpl_set)
4: **if** |tpl_set| = 1 **then**
5: **if** node_id $\notin$ dom(`change_sens`) **then** `change_sens`(node_id) $\leftarrow (0, \texttt{null})$
6: `change_sens`(node_id)[1] $\leftarrow \bigcup$ tpl_set ▷ Extract the single tuple in tpl_set
7: **else**
8: **if** node_id $\notin$ dom($C.m$) $\wedge$ node_id $\notin$ `create_fns` **then**
9: `create_fns`(node_id) $\leftarrow$ tpl_set
10: ADJUST_REF_COUNT(node_id, 1)
11: **return** node_id
12: **function** INSERT(src_id, tpl_set)
13: ADJUST_REF_COUNT(src_id, -1)
14: tgt_id $\leftarrow$ src_id $\oplus\, i$(tpl_set)
15: **if** src_id identifies SEN **then**
16: **return** CREATE(tpl_set $\cup$ {$C.m$(src_id)})
17: **if** node_id $\notin$ dom($C.m$) **then**
18: **if** src_id $\notin$ dom(`change_fns`) **then**
19: `change_fns`(src_id) $\leftarrow (\emptyset \rightarrow \emptyset)$ ▷ Assign an empty map to src_id
20: **if** tgt_id $\notin$ dom(`change_fns`(src_id)) **then**
21: `change_fns`(src_id)(tgt_id) $\leftarrow$ (tpl_set, `ins`) ▷ Plan insertion
22: ADJUST_REF_COUNT(tgt_id, 1)
23: **return** tgt_id
24: **function** REMOVE(src_id, tpl_set)
25: ADJUST_REF_COUNT(src_id, -1)
26: tgt_id $\leftarrow$ src_id $\ominus\, i$(tpl_set)
27: tgt_size $\leftarrow$ |$C.m$(src_id)| $-$ |tpl_set| ▷ tgt_size = 0: Target is empty
28: **if** tgt_size = 1 **then** ▷ Singleton set remains, target is SEN
29: **return** tgt_id $\leftarrow$ CREATE($T(C.m(src\_id)) \setminus$ tpl_set)
30: **if** tgt_size > 1 **then** ▷ Target is FN
31: **if** src_id $\notin$ dom(`change_fns`) **then** `change_fns`(src_id) $\leftarrow (\emptyset \rightarrow \emptyset)$
32: **if** tgt_id $\notin$ dom(`change_fns`(src_id)) **then**
33: `change_fns`(src_id)(tgt_id) $\leftarrow$ (tpl_set, `rem`)
34: ADJUST_REF_COUNT(tgt_id, 1)
35: **return** tgt_id
36: **function** ADJUST_REF_COUNT(node_id, ref_change)
37: **if** node_id identifies SEN **then**
38: **if** node_id $\notin$ dom(`change_sens`) **then** `change_sens`(node_id) $\leftarrow (0, \texttt{null})$
39: `change_sens`(node_id)[0]+ $\leftarrow$ ref_change
40: **else** ▷ node_id identifies FN
41: **if** node_id $\notin$ dom(`fns_delta`) **then** `fns_delta`(node_id) $\leftarrow 0$
42: `fns_delta`(node_id)+ $\leftarrow$ ref_change

Algorithm 2 Plan hypertrie context changes for full nodes

```
 1: function PLAN(create_fns, change_fns, fns_delta)
 2:     new_fns ← (dom(create_fns) ∪ ⋃{dom(g) | g ∈ cod(change_fns)}) \ dom(C.m)
 3:     delete_fns ← {i | i ∈ dom(C.m) ∧ C.r(C.m(i)) + fns_delta(i) = 0}
 4:     fns_delta ← fns_delta |_{dom(fns_delta)\delete_fns}  ▷ Restrict domain of fns_delta
 5:     mv_change_fns ← ∅, cp_change_fns ← ∅
 6:     for all src_id ∈ dom(change_fns), targets ← change_fns(src_id) do
 7:         for all tgt_id ∈ dom(targets) do
 8:             (tuples, mode) ← targets(tgt_id)
 9:             if tgt_id ∉ new_fns then continue
10:             new_fns ← new_fns \{tgt_id}
11:             if src_id ∈ delete_fns then
12:                 delete_fns ← delete_fns \{tgt_id}
13:                 mv_change_fns ← mv_change_fns ∪{(src_id, tgt_id, tuples, mode)}
14:             else
15:                 cp_change_fns ← cp_change_fns ∪{(src_id, tgt_id, tuples, mode)}
16:     new_fns ← {(i, create_fns(i)) | i ∈ new_fns}
17:     return (new_fns, cp_change_fns, mv_change_fns, delete_fns, fns_delta,
    change_sens)
```

by the function's caller. The identifier of the target node (tgt_id) is calculated by combining the identifier of the source node (src_id) with the identifier of the tuple set (tpl_set, l.14). If the source node is a SEN, the tuple it represents is added to tpl_set, and the combined set is passed to CREATE (ll. 15-16). Otherwise, the algorithm plans a FN to FN change if the target node is not already in the hypertrie context, and an increment of the target node's reference count.

REMOVE (ll.24) plans the removal of tuples from a source node, decrementing the source node's reference count. Depending on the size of the resulting node no action is required (tgt_size = 0), a SEN is created using CREATE (tgt_size = 1) or a change from the source FN to the target FN is planned, similar to INSERT (tgt_size > 1).

ADJUST_REF_COUNT (ll.36): Adjusts the reference count of a node.

A single call to one of these functions, using the identifier of a primary hypertrie, initiates the algorithm. For insertions, CREATE is called for empty nodes, while INSERT handles additions to SENs or FNs. For tuple removals, REMOVE is used, and to delete all tuples from a primary node, its reference count is reduced using ADJUST_REF_COUNT. During phase 3, additional function calls are issued to propagate changes to child nodes.

C.2 Phase 2: Plan Changes

In the second phase, FN changes are planned based on the mappings produced during the first phase. The purpose of this planning is to identify a set of changes that produces the same result with minimal computational cost while ensuring

that each node is created only once and is otherwise referenced. There are three ways to create or modify a FN: (i) *creating a node from scratch*, (ii) *copying and updating an existing node* and (iii) *reusing an existing node*. The *first option* is the most computationally expensive, as it requires constructing a new node, generating its child mappings, and recursively requesting to create all child nodes. The *second option*, copying and updating, is more efficient as it modifies only the necessary child mappings on a copy while leaving the rest unchanged. For unchanged child mappings, only a reference count increment is needed, avoiding unnecessary child node creations. Recursive calls to CREATE, INSERT or REMOVE, and ADJUST_REF_COUNT are issued only for added, changed, or deleted child mappings respectively. The *third option* is the least costly because it eliminates the need for copying by directly modifying a node that is no longer referenced elsewhere in the hypertrie. Similar to the second approach, added, changed, or removed child mappings require recursive calls, but no calls are needed for unchanged child mappings. Since reusing a node is a destructive operation, it must be ensured that no pending changes depend on it and it is no longer referenced after all operations are completed.

Algorithm 2 determines which FNs must be created or deleted, identifies updates that require copying a source node versus reusing it, and computes the necessary deltas for FNs. The algorithm selects the least costly option for each decision and ensures that each node is created only once. It begins by computing the set of node identifiers that need to be created (l. 2). This set includes nodes created from scratch (`create_fns`) as well as target nodes resulting from changes (`change_fns`), excluding those already present in the hypertrie context. Next, `delete_fns` (l. 3) determines identifiers of nodes, which will have a reference count of 0 after the update. In l. 4, these soon-to-be-deleted nodes are excluded from `fns_delta` because their reference counts, along with the nodes themselves, will be removed in phase 3. Subsequently, the nested `for` loops process all requested changes in `change_fns`, distributing them into one of two sets: `mv_change_fns`, if the source node will be deleted and can thus be directly reused, or `cp_change_fns`, if the source node remains needed elsewhere and must be copied. If the target node does not need to be created (l. 9), i.e., it was already processed in a previous iteration, the loop proceeds to the next iteration. Otherwise, if the source node is marked for deletion, its identifier is removed from `delete_fns`, and the change is placed to `mv_change_fns`; if it is not marked for deletion, the change is added to `cp_change_fns`. After completing these loops, `new_fns` is updated (l. 16) to pair each new node identifier with its corresponding set of tuples. Finally, the algorithm returns the following sets and mappings that will be used in phase 3:

`new_fns` contains FNs paired with their encoded tuples.
`cp_change_fns` contains the changes that require copying an existing node.
`mv_change_fns` contains the changes that reuse an existing node.
`delete_fns` contains the FNs to be deleted.
`fns_delta` maps FNs to reference count deltas.
`change_sens` contains the changes and reference count adjustments for SENs.

Algorithm 3 Apply hypertrie context changes

```
1: function APPLY(new_fns, cp_change_fns, mv_change_fns, delete_fns, fns_delta, change_sens)
2:     done_fns ← ∅
3:     for all (id, tuples) ∈ new_fns do ▷ Create FNs and sets their reference counts
4:         node ← CREATE_FN(tuples)
5:         C.r(node) ← fns_delta (id), C.m(id) ← node
6:         done_fns ← done_fns ∪{id}
7:     for all (src_id, tgt_id, tuples, mode) ∈ cp_change_fns do  ▷ FNs copy-changes
8:         tgt_node ← CP_CHANGE_FN(src_id, tuples, mode)
9:         C.r(tgt_node) ← fns_delta (tgt_id), C.m(tgt_id) ← tgt_node
10:        done_fns ← done_fns ∪{tgt_id}
11:    for all (src_id, tgt_id, tuples, mode) ∈ mv_change_fns do  ▷ FN move-changes
12:        tgt_node ← MV_CHANGE_FN(src_id, tuples, mode)
13:        C.r ← C.r|_{dom(C.r)\{C.m(src_id)}}, C.m ← C.m|_{dom(C.m)\{src_id}}  ▷ Remove src
14:        C.r(tgt_node) ← fns_delta (tgt_id), C.m(tgt_id) ← tgt_node  ▷ Add tgt
15:        done_fns ← done_fns ∪{src_id, tgt_id}
16:    for all id ∈ delete_fns do  ▷ FNs deletes
17:        DELETE_FN(id)
18:        C.r ← C.r|_{dom(C.r)\{C.m(id)}}, C.m ← C.m|_{dom(C.m)\{id}}  ▷ Remove id
19:    for all id ∈ (dom(fns_delta) \ done_fns) do  ▷ Reference count update
20:        C.r(C.m(id))+ ← fns_delta(id)
21:    if depth > 1 then
22:        for all id ∈ change_sens do  ▷ SEN changes
23:            (node, ref_count_delta) ← change_sens(id)
24:            if ref_count_delta = 0 then continue
25:            if id ∉ dom(C.m) then C.r(node) ← 0, C.m(id) ← node  ▷ Add id
26:            C.r(node)+ ← ref_count_delta  ▷ Update or assign ref count
27:            if C.r(node) = 0 then  ▷ Remove unreferenced SENs
28:                C.r ← C.r|_{dom(C.r)\{C.m(id)}}, C.m ← C.m|_{dom(C.m)\{id}}
```

C.3 Phase 3: Apply Changes

After the planning phase is complete, the specified changes to the hypertrie context are carried out in phase 3, as shown in Algorithm 3. In this phase, the algorithm updates the hypertrie context C to reflect the changes on nodes at the current depth d.

The process begins by initializing a set `done_fns` (l.2) to track which FNs have been successfully updated. First, new FNs are created (ll.3), and their reference counts are set according to `fns_delta`. Each newly created FN is recorded in `done_fns`. Next, updates that require copying a source FN are applied (ll.7), followed by updates that reuse an existing source FN (ll.11). As discussed in Appendix C.2, applying move updates only after copy updates ensures the source FN remains available when needed. Once these creation and update steps are complete, nodes scheduled for deletion are removed from the hypertrie context (ll.16). Any remaining reference count adjustments for existing nodes not yet processed are then applied (ll.19). Domain restrictions on $C.m$ and $C.r$ ensure

that no obsolete nodes remain. Finally, changes to SENs are applied (ll.22). SEN adjustments may involve incrementing or decrementing reference counts, adding newly referenced tuples, or removing SENs from the context if their reference count drops to 0. For depth-1 hypertries, this step is unnecessary because SENs at the lowest level are stored in-line rather than as separate nodes. In this algorithm, changes to FNs are executed through four core functions that return the updated node:

CREATE_FN creates a new FN for the given tuples.
CP_CHANGE_FN copies an existing FN and applies changes to the copy.
MV_CHANGE_FN applies changes directly to an existing FN.
DELETE_FN deletes an FN by decrementing child reference counts.

Each of these functions computes the new child mappings and propagates changes recursively using the phase 1 functions. In the following we first explain how child mappings are derived, and then describe how each core function uses these mappings to apply changes and trigger the necessary recursive changes.

C.4 Change Child Mappings

Changes are applied on each dimension's child mapping individually. Therefore, we define $\delta_p^{(t)}$ to capture the changes for dimension p. $\delta_p^{(t)}$ groups a set of d-tuples $t = \{t^1 \dots t^k\}$ by their p-th element, discarding that coordinate for the grouped result. t^j_ℓ represents the ℓ-th element of t^j, while $t^j_{\ell,\ell+1}$ denotes the tuple consisting of the ℓ-th and $(\ell{+}1)$-th elements.

$$\delta_p^{(t)} : t^j_p \mapsto \{t^\ell_{1,\dots,p-1,p+1,\dots,d} \mid \ell \in \{1 \dots k\} \wedge t^\ell_p = t^j_p\}, \quad j \in \{1 \dots k\}. \tag{1}$$

Tuple Insertion & Node Creation Insertion is the process of adding a new set of tuples t to the set of tuples represented by an empty or FN $h \in H(d)$ with $|h| + |t| \neq 1$ and $T(h) \cap t = \emptyset$. *Inserting* t into source hypertrie h results in a target FN $h' \in H(d)$ that represents the tuples $T(h) \cup t$. We also refer to insertions into an empty node as *creating a node*. The update to h is reflected in the child mappings $c_p^{(h')}$ of h' for each dimension $p \in \{1 \dots d\}$. For $d > 2$, we consider three primary cases:

$$c_p^{(h')} : k \mapsto \begin{cases} c_p^{(h)}(k), & \text{if } k \in \mathrm{dom}(c_p^{(h)}) \setminus \mathrm{dom}(\delta_p^{(t)}); \quad (2) \\ i(\delta_p^{(t)}(k)), & \text{if } k \in \mathrm{dom}(\delta_p^{(t)}) \setminus \mathrm{dom}(c_p^{(h)}); \quad (3) \\ c_p^{(h)}(k) \oplus i(\delta_p^{(t)}(k)), & \text{if } k \in \mathrm{dom}(c_p^{(h)}) \cap \mathrm{dom}(\delta_p^{(t)}). \quad (4) \end{cases}$$

The existing child identifier is retained if no new tuples map to k (Equation (2)). If k is new, the identifier of $\delta_p^{(t)}(k)$ is used (Equation (3)). If k was already present and now also appears in $\delta_p^{(t)}(k)$, the old identifier is combined with the identifier of $\delta_p^{(t)}(k)$ (Equation (4)). For depth $d = 2$, the insertion logic closely resembles the $d > 2$ case, with minor adjustments to support reading and writing *in-place*

nodes. In Equation (3), a depth-1 SEN is stored in place if $|\delta_p^{(t)}(k)| = 1$. In Equation (4), the function $i(\cdot)$ is applied to the left side if $c_p^{(h)}(k)$ is a SEN.

The mappings in leaf nodes of depth $d = 1$ simplifies to a union of the mappings:

$$c_1^{(h')} : k \mapsto \mathit{true}, \text{ if } k \in \mathrm{dom}(c_1^{(h)}) \cup \mathrm{dom}(\delta_1^{(t)}). \tag{5}$$

Tuple Removal & Node Deletion Analogous to insertion, removal is the process of removing a set of tuples t from the set of tuples a hypertrie node h represents. Consider a FN $h \in H(d)$ with $|h| > 2$ and a set of tuples t with $t \subseteq T(h)$. As in the case of insertions, the change is reflected in the resulting child mappings $c_p^{(h')}$. For depth $d > 2$, we distinguish 3 cases.

1. If no tuples are removed from a mapped child, i.e., $k \in \mathrm{dom}(c_p^{(h)}) \setminus \mathrm{dom}(\delta_p^{(t)})$, the child's original mapping is kept:

$$c_p^{(h')} : k \mapsto c_p^{(h)}(k). \tag{6}$$

2. If k is mapped by both $c_p^{(h)}$ and $\delta_p^{(t)}(k)$, and *not* all tuples are removed from the mapped child node, i.e., $k \in \mathrm{dom}(c_p^{(h)}) \cap \mathrm{dom}(\delta_p^{(t)}) \wedge c_p^{(h)}(k) \neq i(\delta_p^{(t)}(k))$, the mapping is defined:

$$c_p^{(h')} : k \mapsto \begin{cases} c_p^{(h)}(k) \ominus i(\delta_p^{(t)}(k)), & \text{if } |C.m(c_p^{(h)}(k))| - |\delta_p^{(t)}(k)| > 1; \quad (7) \\ s(c_p^{(h)}(k) \ominus i(\delta_p^{(t)}(k))), & \text{if } |C.m(c_p^{(h)}(k))| - |\delta_p^{(t)}(k)| = 1. \quad (8) \end{cases}$$

   In the first subcase the resulting child is a FN and in the second subcase it is a SEN and its identifier must be tagged as such.
3. Finally, if k is mapped by both $c_p^{(h)}$ and $\delta_p^{(t)}(k)$, and all tuples are removed from the mapped child node, i.e., $k \in \mathrm{dom}(c_p^{(h)}) \cap \mathrm{dom}(\delta_p^{(t)}) \wedge c_p^{(h)}(k) = i(\delta_p^{(t)}(k))$, the child mapping is fully removed.

For depth 2, the mapping is the same except for Equation (8). Here, the remaining SEN is calculated and stored in-place. For depth 1, the tuples of $\delta_1^{(t)}$ can simply be removed from $c_1^{(h)}$:

$$c_1^{(h')} : k \mapsto \mathit{true}, \text{if } k \in \mathrm{dom}(c_1^{(h)}) \setminus \mathrm{dom}(\delta_1^{(t)}). \tag{9}$$

Removing the entire encoded set $t := T(h)$ from a node h is referred to as *deleting* h, resulting in empty child mappings.

C.5 Change Full Nodes & Propagate Changes to Child Nodes

With the calculations of the target child mappings defined, we can now specify the behavior of the functions CREATE_FN, MV_CHANGE_FN, CP_CHANGE_FN used in Algorithm 3. Consider a set of tuples t. CREATE_FN(t) *creates* a node h from t. For each resulting child mapping (Equation (3)), CREATE($\delta_p^{(t)}(k)$)

is called. Analogous, DELETE_FN(t) *deletes* a node h. For each mapped identifier id in the child mappings of h, ADJUST_REF_COUNT($id, -1$) is called. MV_CHANGE_FN($h, t,$ `ins`) *inserts* t into h. For each newly added child mapping (Equation (3)), CREATE($\delta_p^{(t)}(k)$) is invoked. For each altered child mapping (Equation (4)), an insertion is requested into the mapped child node using INSERT($i(c_p^{(h)}), \delta_p^{(t)}(k)$). Similarly, MV_CHANGE_FN($h, t,$ `rem`) *removes* t from h. For each removed child mapping, ADJUST_REF_COUNT($c_p^{(h)}, -1$) is invoked to decrement the reference count. For each altered mapping (Equation (7)), a removal is requested using REMOVE($i(c_p^{(h)}), \delta_p^{(t)}(k)$). CP_CHANGE_FN($h, t, w$), where $w \in \{$`ins`, `rem`$\}$, begins by creating a copy h^c of the hypertrie h, with the result mappings initialized as $c_p^{(h^c)} := c_p^{(h)}$. It increments the reference count for each id in the resulting child mappings using ADJUST_REF_COUNT($id, 1$). Finally, it invokes MV_CHANGE_FN(h^c, t, w) to apply the specified change.

References

1. Bellare, M., Micciancio, D.: A new paradigm for collision-free hashing: Incrementality at reduced cost. In: Fumy, W. (ed.) Advances in Cryptology — EUROCRYPT '97, vol. 1233, pp. 163–192. Springer Berlin Heidelberg, Berlin, Heidelberg (1997). https://doi.org/10.1007/3-540-69053-0_13, series Title: Lecture Notes in Computer Science
2. Bigerl, A., Conrads, L., Behning, C., Saleem, M., Ngonga Ngomo, A.C.: Hashing the hypertrie: Space- and time-efficient indexing for sparql in tensors. In: Sattler, U., Hogan, A., Keet, M., Presutti, V., Almeida, J.P.A., Takeda, H., Monnin, P., Pirrò, G., d'Amato, C. (eds.) The Semantic Web – ISWC 2022. pp. 57–73. Springer International Publishing, Cham (2022)
3. Bigerl, A., Conrads, L., Behning, C., Sherif, M.A., Saleem, M., Ngonga Ngomo, A.C.: Tentris – A Tensor-Based Triple Store. In: Pan, J.Z., Tamma, V., d'Amato, C., Janowicz, K., Fu, B., Polleres, A., Seneviratne, O., Kagal, L. (eds.) The Semantic Web – ISWC 2020. pp. 56–73. Springer International Publishing, Cham (2020)
4. De La Briandais, R.: File searching using variable length keys. In: Papers presented at the the March 3-5, 1959, western joint computer conference. pp. 295–298 (1959)
5. Lewi, K., Kim, W., Maykov, I., Weis, S.A.: Securing update propagation with homomorphic hashing. IACR Cryptol. ePrint Arch. **2019**, 227 (2019), `https://api.semanticscholar.org/CorpusID:75138272`
6. Maitin-Shepard, J., Tibouchi, M., Aranha, D.F.: Elliptic curve multiset hash. The Computer Journal **60**(4), 476–490 (Mar 2017). https://doi.org/10.1093/comjnl/bxw053
7. Mihajloska, H., Gligoroski, D., Samardjiska, S.: Reviving the idea of incremental cryptography for the zettabyte era use case: Incremental hash functions based on SHA-3. In: Camenisch, J., Kesdoğan, D. (eds.) Open Problems in Network Security. pp. 97–111. Lecture Notes in Computer Science, Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-39028-4_8