

OntoDB: A Fast and Persistent Query Engine for Ontologies

Lam Nguyen

Case Western Reserve University
Cleveland, Ohio, USA
ltn18@case.edu

ABSTRACT

Ontologies are queried in three ways: for the triples that match a pattern, for the instances at or below a class, and for the terms closest in meaning to a given term. Triple stores answer the first well, handle the second by rewriting queries or by materializing entailed triples, and leave the third to a separate vector database; none is built to serve many threads from one embedded engine, or to run in a web page. We present ONTODB, an ontology database that answers all three kinds of query inside one disk-based, multi-threaded engine. Through ONTODB we make the following contributions. (1) A B+ tree triple store of three permutations behind a buffer pool, with algorithms for search, insert, delete, range scan, and bulk load, a bound on the pages a range scan reads, and write-ahead logging with ARIES-style recovery for persistence. (2) A concurrency design that scales: an optimistic leaf-only descent for inserts and deletes, a sharded buffer pool whose hits take no latch, and a one-word page latch. (3) A labelling of the class hierarchy with postorder intervals kept in the same B+ tree, so that “everything under a class” is a few range scans and no entailed triple is stored, and a plan rewrite that answers RDFS subclass entailment of `rdf:type` patterns with these scans, with the answers of materialization, prepared 16 times faster and in 8 times less space. (4) Exact, IVF, product-quantized, and HNSW vector indexes in the engine, a proof that no linear order of points keeps neighbors close, and the finding that in an ontology a class filter is correlated with the vectors, so the usual independence assumption fails. (5) A 0.85 MB WebAssembly build that runs the engine in a browser. On LUBM and the Gene Ontology, against Oxigraph, Apache Jena TDB2, and SQLite, ONTODB answers selective triple patterns 3 to 37× faster and the instances of a class 2 to 48× faster, and serves 3 to 7× more lookups per second with eight threads (1.46 million per second). Its interval labels store 8.1× fewer triples than materializing the hierarchy and read 48× fewer pages than rewriting the query, and exact pre-filtering through them is 10^2 to 10^4 × faster than any index strategy for classes below about 1,000 members.

KEYWORDS

ontology, triple store, B+ tree, RDFS, vector search, WebAssembly

PVLDB Reference Format:

Lam Nguyen. OntoDB: A Fast and Persistent Query Engine for Ontologies. PVLDB, 20(1): XXX-XXX, 2027.
doi:XX.XX/XXX.XX

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 20, No. 1 ISSN 2150-8097.

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/lamng3/OntoDB>.

1 INTRODUCTION

Ontologies such as the Gene Ontology [4] and the schemas of enterprise knowledge graphs are stored as sets of triples and queried in three ways. A *pattern* query asks for the triples that match a template and is the base case of every RDF store [26, 33]. A *structural* query asks for the instances at or below a class, and needs the class hierarchy. A *semantic* query asks for the terms closest in meaning to a given term, which needs a similarity measure over embeddings. The three meet in “the terms most similar to *apoptotic process* that lie under *cell death*”.

Existing systems leave four problems open. **C1. Hierarchy costs space or time.** RDFS entailment is either *materialized*, which stores every inferred triple [24], or *rewritten* into the query, which scans once per subclass. On the Gene Ontology with 200,000 typed entities, materializing multiplies the triples by 8.1, and rewriting reads 48 times more pages than a store that knows the hierarchy. **C2. A sorted index does not hold meaning.** It is tempting to believe that neighbors in meaning are neighbors in the order; we prove they cannot all be (Section 7), so similarity is left to a vector database beside the triple store, with its own storage and no view of the hierarchy. **C3. A class filter is correlated with the vectors.** Filtered vector search is costed by assuming the filter is independent of the similarity; in an ontology the neighbors of a term lie under its own ancestors, and a post-filtered index can find 7% of the right answers. **C4. A paged store must scale across threads.** A classic buffer pool serializes every page access on one latch, and a classic B+ tree write latches the root. In our first version, lookups per second fell from 330 thousand with one thread to 180 thousand with eight.

Example 1.1. Let q be *apoptotic process* in the Gene Ontology and c the class *cell death*. The query asks for the five terms nearest to q among the 97 classes at or below c in an 834-class slice. A triple store cannot rank by meaning; a vector index returns neighbors from anywhere, and a post-filter keeps those under c only if enough are. ONTODB finds the 97 classes with one range scan of the B+ tree that holds the triples, scores them exactly, and returns *apoptotic process involved in development* (cosine 0.865) first, in under a millisecond in a web page.

doi:XX.XX/XXX.XX

Contributions. This paper introduces ONTODB, an open-source ontology database in C++20 (Figure 1), with five technical contributions.

- (1) **A concurrent B+ tree triple store** (Sections 3 and 4): the algorithms for search, insert, and delete, a bound on the pages a range scan reads, write-ahead logging with page-level recovery, and a latch-free hit path and an optimistic writer descent that scale to eight threads.
- (2) **Hierarchies as intervals** (Section 5): a postorder labelling of the class DAG kept in the same B+ tree, so that “everything under c ” is a few range scans and no entailed triple is stored.
- (3) **An entailment-aware plan rewrite** (Section 6): RDFS subclass entailment of `rdf:type` patterns as interval scans the optimizer costs like any other, with the answers of materialization, prepared 16 times faster and in 8 times less space.
- (4) **Vector indexes and filtered search** (Section 7): a proof that no sorted order keeps neighbors close, IVF posting lists stored as key ranges of the triple tree, and the finding that a class filter is correlated with similarity, with the rule that chooses a strategy.
- (5) **A build for the browser** (Section 8): the whole engine as a 0.85 MB WebAssembly module.

We evaluate ONTODB against Oxigraph [29], Apache Jena TDB2 [3], and SQLite on LUBM [14] and the Gene Ontology (Section 9, Table 3). The code, data generators, and benchmarks are at the artifact URL above.

2 PRELIMINARIES

Triples and patterns. An RDF graph is a set $\mathcal{T}$ of triples (s, p, o) of subject, predicate, and object, with $n = |\mathcal{T}|$. A *triple pattern* (s, p, o) replaces some positions by variables, written $\cdot$ here when the variable is not used elsewhere; its answers are the triples of $\mathcal{T}$ that agree with it on every bound position. A *basic graph pattern* is a set of triple patterns joined on shared variables. Following common practice [26, 33], ONTODB replaces each term a by a dense integer $id(a)$ and stores triples as three-integer keys.

Class hierarchy. The `rdfs:subClassOf` triples define a directed graph $H = (\mathcal{C}, E)$ over classes, with an edge (c, c') when c is a subclass of c' . We write $desc(c)$ for the classes at or below c , including c itself, and call a class *below* c when it is in $desc(c)$. Cycles are collapsed into one class, since the classes on a cycle are equivalent. The *instances* of c are the subjects e with a triple $(e, \text{rdf:type}, c')$ for some $c' \in desc(c)$.

Similarity. Each term t among N terms has a unit vector $\mathbf{v}(t) \in \mathbb{R}^d$ produced by an embedding model, and the similarity of two terms is the cosine $\text{sim}(t, u) = \mathbf{v}(t) \cdot \mathbf{v}(u)$.

Definition 2.1 (Queries). Given the data above, ONTODB answers (i) a *pattern query*, the answers of a basic graph pattern; (ii) an *instance query* $\text{inst}(c)$, the instances of a class c ; and (iii) a *filtered similarity query* $\text{near}(q, c, k)$, the k terms $t \in desc(c)$, $t \neq q$, that maximize $\text{sim}(q, t)$.

Table 1 lists the notation used throughout.

Table 1: Notation.

Symbol	Meaning
$(s, p, o), \mathcal{T}, n$	a triple; the set of triples; $n = \mathcal{T} $
$id(a)$	dictionary id of term a
x, σ	a B+ tree key, three ids in 24 bytes; a separator key
e	an instance (an entity with a type)
$\pi, \ell(\pi)$	a permutation (SPO, POS, OSP); the length of the prefix of π a pattern binds
B, F	capacity of a leaf (169 keys) and of an internal page (144 separators)
h	height of a B+ tree
P, Q, R	a page; its parent; its right sibling
$[a, b), r$	a half-open key range; the number of keys a scan returns
$H = (\mathcal{C}, E)$	the class hierarchy, a DAG of classes $\mathcal{C}$ and subclass edges E
$desc(c)$	the classes at or below c , including c
$po(c), lo(c)$	postorder number of c ; smallest number in its subtree of the spanning forest
$I(c)$	the set of intervals that covers the numbers of $desc(c)$
$E'(c)$	non-tree edges whose tail is in $desc(c)$
$f(c)$	fraction of terms in $desc(c)$: $ desc(c) /N$
N, d	number of terms with vectors; vector dimension
$\mathbf{v}(t), \text{sim}$	unit vector of t ; cosine similarity
q, k	the query term; the number of neighbors asked for
$\text{inst}(c), \text{near}(q, c, k)$	the instance query and the filtered similarity query
v, ρ	IVF: number of lists; lists probed per query
S, τ	buffer-pool shards; threads
m	product quantization: pieces per vector
M, ef	HNSW: links per node; search beam width
G, D	a graph and its diameter (Lemma 7.1)

3 THE ONTODB STORE

ONTODB is a single-process engine with the layers of a disk-based DBMS (Figure 1). A SPARQL query is parsed, bound against the dictionary, planned as a left-deep tree, reordered by the optimizer, and run by volcano-style executors that pull rows of term ids. The *indexed store* keeps each triple in three B+ trees in one page file, behind a buffer pool with LRU-K eviction [27]; terms are encoded by a dictionary built from the same parts; updates are logged page by page for ARIES-style recovery [23]. The semantic extension (Sections 5–7) reads the store through the same interface.

Keys, pages, permutations. A triple is a 24-byte key x of three dictionary ids, stored big-endian so that `memcmp` is the order. A page is 4 KiB; a leaf holds up to $B=169$ keys and an internal page up to $F=144$ separators; leaves are linked to their right sibling only; every page but the root keeps at least half its maximum. With the permutations SPO, POS, and OSP every triple pattern has one in which its bound positions form a prefix, so a pattern is one range scan that returns only matching triples (Appendix A); a fourth to sixth permutation would make no pattern cheaper (Section 9).

Search. CHILDFor picks the child whose range holds x : the last separator that is $\leq x$. The descent *crabs*: it latches the child before

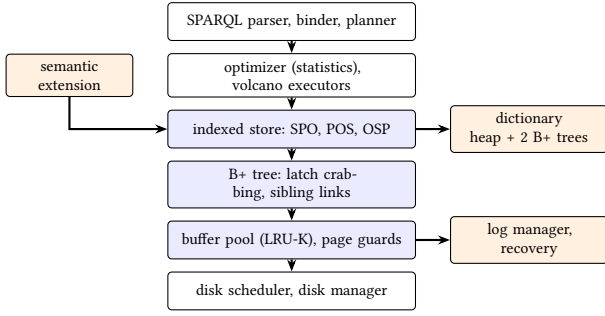


Figure 1: The layers of ONTODB. Blue boxes are the storage structures of Section 3.

Algorithm 1: LEAFFOR(x) and CONTAINS(x)

```

1 latch the header page (shared);  $P \leftarrow$  the root; if  $P$  is nil then
  return nil;
2 latch  $P$  (shared); release the header;
3 while  $P$  is an internal page do
4    $P' \leftarrow \text{CHILDFOR}(P, x)$ ; // binary search over the
    separators
5   latch  $P'$  (shared); release  $P$ ;  $P \leftarrow P'$ ;
6 return  $P$ ;
7 CONTAINS( $x$ ):  $P \leftarrow \text{LEAFFOR}(x)$ ; return whether  $x$  is in  $P$ ;
```

it releases the parent, so a concurrent split cannot move x away [12, 20]. A lookup reads $h \leq 2 + \log_{73}(n/84)$ pages (at most four for the 1.3 million triples of LUBM(10)).

Insert. Insert is *pessimistic* crabbing: it write-latches each page on the way down and releases every latched ancestor as soon as the page just latched is *safe*, meaning it cannot split. What is still latched at the leaf is exactly what a split can reach, and latches are taken in one order everywhere (header, top to bottom, siblings left to right). A leaf split leaves two leaves of $\lfloor (B+1)/2 \rfloor$ and $\lceil (B+1)/2 \rceil$ keys and copies the right leaf’s first key up; an internal split moves the middle separator up.

Delete. Delete crabs with a different notion of safe: a page above its minimum, because only then can it not underflow. A leaf below $\lfloor B/2 \rfloor$ keys merges with or borrows from a sibling under the same parent, preferring the left one; a leaf emptied by a merge is kept, empty, with its sibling link, because an iterator may already hold its id.

Range scan and bulk load. The half-open range $[a, b)$ is what the semantic extension needs. The iterator *copies* the keys of a leaf below b , remembers the right sibling’s id, and releases the leaf, so it holds no latch and no pin between rows and a stalled consumer blocks no writer (Appendix A).

LEMMA 3.1 (COST OF A RANGE SCAN). *A scan returning r keys from an insert-only or bulk-loaded tree of height h reads at most $h + \lceil r/\lfloor B/2 \rfloor \rceil$ pages, and at most $h + \lceil r/B \rceil$ if the tree was bulk-loaded.*

Loading sorts the triples once per permutation and builds each tree bottom-up, writing every page once and full: LUBM(1) takes 10.8 MiB by bulk load against 16.2 MiB by insertion.

Algorithm 2: INSERT(x)

```

1 latch the header (exclusive); if the tree is empty then make a leaf
  holding  $x$  the root; return true;
2  $path \leftarrow []$ ;  $P \leftarrow$  the root;
3 repeat
4   latch  $P$  (exclusive);
5   if  $P$  is not full then
6     release every latch in  $path$  and the header; //  $P$  is safe
7     push  $P$  on  $path$ ;  $P \leftarrow \text{CHILDFOR}(P, x)$ ;
8   until  $P$  is a leaf;
9    $P \leftarrow path.pop()$ ; if  $x \in P$  then return false;
10  insert  $x$  into  $P$ ; if  $|P| \leq B$  then return true;
11   $R \leftarrow$  new leaf with the upper half of  $P$ ;  $R.next \leftarrow P.next$ ;
     $P.next \leftarrow R$ ;  $\sigma \leftarrow R.keys[0]$ ; // copied up
12 while true do
13   if  $path$  is empty then
14     make a new root over  $(P, R)$  with separator  $\sigma$ ; set it in the
      header; return true
15    $Q \leftarrow path.pop()$ ; insert  $(\sigma, R)$  into  $Q$  after  $P$ ;
16   if  $Q$  has at most  $F$  separators then return true;
17    $P \leftarrow Q$ ;  $R \leftarrow$  new page with the upper half of  $Q$ ;  $\sigma \leftarrow$  the
      middle separator of  $Q$ ; // moved up
```

Algorithm 3: REMOVE(x)

```

1 descend as in INSERT, releasing ancestors when the page just
  latched is safe (above its minimum);
2  $P \leftarrow$  the leaf; if  $x \notin P$  then return false;
3 remove  $x$  from  $P$ ; if  $P$  is the root or  $|P| \geq \lfloor B/2 \rfloor$  then return true;
4 while the page  $P$  on top of  $path$  is under its minimum and has a
  parent  $Q$  do
5    $R \leftarrow$  the left sibling of  $P$  under  $Q$  (the right one if  $P$  is first);
    latch siblings left to right;
6   if the entries of  $P$  and  $R$  fit in one page then
7     merge into the left page and delete the right entry from
       $Q$ ;
8   else
9     redistribute: move one entry from  $R$  to  $P$  and fix the
      separator in  $Q$ ;
10   $P \leftarrow Q$ ;
11 if the root has one child left then make the child the root and free
    the old root;
```

Buffer pool, dictionary, planning, recovery. A *page guard* pins a frame and takes its latch for its lifetime, so no path leaves a page pinned. Terms get dense 64-bit ids in first-seen order, stored in a heap, an $id \rightarrow record$ tree, and a $spelling \rightarrow id$ tree keyed by a 64-bit hash; decoding an id costs about $0.7 \mu s$, so a direct-mapped cache of 65,536 decoded spellings serves a warm store. The optimizer reorders a left-deep plan greedily by estimated join output $a_1 a_2 / \max(V_1, V_2)$ from exact counts collected at load, and picks a nested loop, an index nested loop, or a hash join per node; the SPARQL subset has no OPTIONAL, UNION, property paths, or aggregates. Updates run under strict two-phase locking. Persistence has two paths. The page store logs a page image when a dirty page is released, forces the log before the page is written, redoes by copying images, undoes a triple logically, and logs a

Algorithm 4: TRYLEAFONLY(x, op) for an insert or a delete

```
1 read-latch the header;  $P \leftarrow$  the root; if the tree is empty or  $P$  is a
  leaf then return fail;
2 read-latch  $P$ ; release the header;
3 while true do
4    $P' \leftarrow$  CHILDFOR( $P, x$ ); pin  $P'$  and read its type;
5   if  $P'$  is internal then read-latch  $P'$ ; release  $P$ ;  $P \leftarrow P'$ ;
     continue;
6   write-latch  $P'$  (the leaf); release  $P$ ;
7   if  $op$  is insert and  $x \in P'$  or  $op$  is delete and  $x \notin P'$  then
8     return done (no change)
9   if  $op$  is insert and  $P'$  is full, or  $op$  is delete and  $|P'| \leq \lfloor B/2 \rfloor$ 
     then return fail;
10  insert or remove  $x$  in  $P'$ ; return done;
```

split or merge as one atomic *page group*; a crash harness kills the process between the pages of every multi-page operation and checks the tree afterwards. The database facade’s durable mode (Database::Open) keeps a checksummed snapshot and a logical log over the in-memory backend and replays committed transactions on opening; wiring the page-level log into the facade is future work.

4 CONCURRENCY

Strict two-phase locking isolates transactions; this section is about the *latches* that keep pages and trees sound, which decide how many threads the engine can use. Our first implementation followed the textbook (one latch over the pool’s tables, a mutex over the dictionary, exclusive latches from the root for every insert), and sampling eight reader threads showed that four of five samples were threads waiting for a lock. We replaced each in turn.

An optimistic descent for writers. Most inserts split nothing, so TRYLEAFONLY descends with read latches, write-latches only the leaf, and finishes if the leaf has room (for a delete, if it stays at least half full); otherwise nothing has been written and the pessimistic algorithm runs. The type of a child is read from its pinned page, which cannot change, so the leaf is write-latched while its parent is still read-latched (Algorithm 4); the latch order stays top to bottom.

A sharded pool with a latch-free hit. The pool has S shards (16 in our runs); a page belongs to shard $id \bmod S$, with its own frames, page table, and LRU-K replacer. The pages every query touches, the roots, hash to a few shards, so a hit on a resident page takes no latch at all: the shard keeps its page table also as an open-addressing array of atomic words, a reader finds the frame and adds a pin with a compare-and-swap that fails if the frame has been *claimed* (pin count -1), and checks that the frame still holds the page. A frame is evicted or deleted only after its owner swaps a pin count of zero to -1 , so a frame is never evicted under a reader, and a pin that raced with an eviction sees another page and lets go. A hit sets a *referenced* bit instead of touching the replacer, and a shard folds the bits into LRU-K when it must evict; a pool with one shard, the default, keeps the exact textbook behavior (Algorithm 5).

The latch and the dictionary. The page latch is one atomic word (30 bits of readers, a writer-waiting bit, a writer bit), because `shared_mutex` keeps its reader count behind a mutex that

Algorithm 5: FETCH(id) in a shard, and the claim that guards eviction

```
1  $\varphi \leftarrow$  INDEXFIND( $id$ ); // atomic array, no latch
2 if  $\varphi \neq \perp$  and TRYPIN(frame  $\varphi$ ) then
3   if frame  $\varphi$  holds page  $id$  then mark frame  $\varphi$  referenced;
     return frame  $\varphi$ ;
4   unpin frame  $\varphi$ ; // the frame was refilled
5 latch the shard;
6 if page  $id$  is in the page table then pin and return its frame;
7  $\varphi \leftarrow$  a free frame, or else a victim;
8 fold the referenced marks into the replacer; mark frames with
  pin count 0 evictable;
9 repeat
10  ask the replacer for the victim  $v$ ; TRYCLAIM( $v$ ): swap pin
    count 0  $\rightarrow -1$ 
11 until a victim is claimed;
12 ;
13 write  $v$  back if dirty; INDEXERASE( $v$ ); drop  $v$  from the page table;
14 read page  $id$  into  $\varphi$ ; set its pin count to 1; INDEXINSERT( $id, \varphi$ );
   release the shard;
15 return frame  $\varphi$ ;
```

readers of one root page queue on. The dictionary serializes only inserts, so a spelling is never interned twice; decoding takes no dictionary-wide lock, and its cache is guarded by 64 stripe locks.

Correctness. A frame’s pin count is -1 only while a thread holding the shard latch owns the frame, a page is in the atomic index iff it is in the page table, a leaf changes only under its write latch, and the latch order is the same on every path. Two stress tests (eight readers against a pool of 64 frames and 400 pages whose bytes name their page; inserters, deleters, and scanners against a tree of tiny nodes) and the whole suite pass in release mode and under ThreadSanitizer and AddressSanitizer.

5 HIERARCHIES AS INTERVALS

An instance query $\text{inst}(c)$ (Definition 2.1) must return the instances of every class in $\text{desc}(c)$. A store can *materialize* the entailed type triples, or *rewrite* the query into one scan per class in $\text{desc}(c)$. ON-TO-DB instead numbers the classes so that $\text{desc}(c)$ is a few integer intervals, and keeps the numbers in the B+ tree it already has.

Labelling. Collapse each cycle of H into one class. Choose a spanning forest by taking, for each class with several parents, the parent with the longest path from a root. Number the classes $1, \dots, |\mathcal{C}|$ in postorder of the forest, and let $lo(c)$ be the smallest number in the forest subtree of c . In a forest, $u \in \text{desc}(c)$ if and only if $lo(c) \leq po(u) \leq po(c)$, a single interval. In a DAG each class carries a *set* of intervals $I(c)$, built bottom-up (Algorithm 6).

LEMMA 5.1 (CORRECTNESS). $u \in \text{desc}(c)$ if and only if $po(u)$ lies in an interval of $I(c)$.

LEMMA 5.2 (SIZE). For every class c , $|I(c)| \leq 1 + |E'(c)|$, where $E'(c)$ is the set of non-tree edges whose tail is in $\text{desc}(c)$.

The labelling takes $O(|\mathcal{C}| + |E|)$ plus the merges, and a tree needs one interval per class.

Algorithm 6: Interval sets for a class DAG

Input: subclass edges E of H

Output: $po(c)$ and an interval set $I(c)$ for each class c

```
1 collapse the strongly connected components of  $H$  (Tarjan), and
  number them parents first;
2 for each component  $c$ , parents first do
3    $depth(c) \leftarrow 1 + \max_{c'} depth(c')$  over its parents  $c'$ ; // 0 for
    a root
4    $tp(c) \leftarrow$  the parent with the largest depth;
5   number the forest  $\{(c, tp(c))\}$  in postorder; record  $lo(c)$ ;
6 for each component  $c$ , children first do
7    $I(c) \leftarrow \text{MERGE}(\{[lo(c), po(c)]\} \cup \bigcup_{c' \text{ a child of } c} I(c'))$ ; // sort;
    join overlapping or adjacent
```

Index. The labels go into B+ trees of ONTODB: key $(po(c), id(c), 0)$ for each class, and $(po(c), id(e), 0)$ for each triple $(e, rdf: type, c)$. The classes below c , or the instances of c and its subclasses, are then one range scan $[(a, 0, 0), (b+1, 0, 0)]$ per interval $[a, b]$ of $I(c)$, and the answers of different intervals are disjoint. Inserting an instance adds one key; deleting a class edge rebuilds the labels, which takes tens of milliseconds for an ontology of tens of thousands of classes. Incremental maintenance with gaps in the numbering [1] is not implemented. Dictionary ids are untouched: ids appear in three permutations, the log, and checkpoints, and renumbering them to follow the hierarchy would rewrite all of it. The label tree is separate and rebuildable, which keeps ids stable.

Membership. A class u is below c if $po(u)$ lies in an interval of $I(c)$, a binary search over $|I(c)|$ intervals with no scan. This makes the class filter of Section 7 a constant-space predicate on a term's number.

6 ENTAILMENT-AWARE QUERY PLANNING

The interval index answers the instance query $inst(c)$ of Section 2, but a SPARQL user writes $?x \text{ rdf: type } c$ and receives the stored triples only: ten of the fourteen LUBM queries return no rows (Section 9). ONTODB therefore offers RDFS subclass entailment as a *plan rewrite*. With `\set entail on`, a type pattern with a constant class is answered by the interval index, and the optimizer, the joins, and every other operator are unchanged.

Semantics. Let $\mathcal{T}^+ \subseteq \mathcal{T}$ together with $(e, rdf: type, c')$ for every $(e, rdf: type, c) \in \mathcal{T}$ and every c' with $c \in \text{desc}(c')$. A triple pattern $(s, rdf: type, c)$ with a constant class has as *entailed answers* its answers over $\mathcal{T}^+$; every other pattern is evaluated over $\mathcal{T}$. A pattern whose class is a variable, a class the hierarchy does not contain, and every other predicate are left as they were. Only `rdfs: subclassOf` is read: no `rdfs: subPropertyOf`, `domain`, `range`, or `OWL axiom`, and no inference over the properties of a query.

The rewrite. Algorithm 7 runs on the planner's tree before the optimizer. It replaces a qualifying scan by an *interval type scan* and records how many rows it expects, which is the number of type triples whose class label lies in an interval of $I(c)$, read from prefix sums over the labels in $O(|I(c)|)$ time. The optimizer then orders the joins with that estimate, as it does for any scan. This matters: from the stored triples alone, $?x \text{ rdf: type } \text{Student}$ looks empty, and

Algorithm 7: REWRITE TYPE SCANS($plan$)

```
1 foreach triple-scan leaf  $(s, p, o)$  of plan do
2   if  $p$  is the constant rdf: type,  $o$  is a constant class  $c$  of the
    hierarchy  $H$  then
3     mark the scan as an interval type scan of  $c$ ;
4      $rows \leftarrow \sum_{[a,b] \in I(c)} (cnt(b) - cnt(a-1))$ ; // prefix sums
    of type triples by label
```

a plan that starts from it starts from a pattern that in fact returns 75,547 rows.

Example 6.1. Over `Dog`, `Cat` $\sqsubseteq$ `Mammal` $\sqsubseteq$ `Animal` and `Bird` $\sqsubseteq$ `Animal`, the query `SELECT ?x ?n WHERE { ?x a ex:Animal . ?x ex: name ?n }` returns one row without entailment (the entity typed `Animal` itself) and four with it. `\explain` shows the rewritten plan, in which the optimizer has joined the interval scan to the name scan by index nested loops:

```
Projection [?x, ?n]
  IndexNestedLoopJoin
    IntervalTypeScan {?x rdf: type ex:Animal}
    TripleScan {?x ex: name ?n}
```

Execution. An interval type scan is still a scan: it yields triples $(e, rdf: type, c)$ for the instances of c , and the operators above it read them as they read stored ones. With a free subject it takes one range scan of the label tree per interval of $I(c)$ and returns the distinct instances in ascending order. With a bound subject, as when an index nested-loop join probes it with each outer row, it reads the subject's stored types from the SPO tree and tests each label against $I(c)$ by binary search. An entailed triple is not stored, so a scan under a transaction locks the triple set and not the triple.

LEMMA 6.2 (SOUNDNESS AND COMPLETENESS). *For every class c of the hierarchy, the interval type scan of c returns $(e, rdf: type, c)$ if and only if this triple belongs to $\mathcal{T}^+$.*

Freshness. The hierarchy and the label tree are a snapshot, kept on a private file of the same buffer-pool engine. Every update, load, and backend switch advances a version number, and the first entailed query after a change rebuilds the snapshot from the store: 0.06 s for LUBM(10) and its 43 classes, a few tens of milliseconds for the Gene Ontology. Incremental maintenance of the labels with gaps [1] would avoid the rebuild and is not implemented. The snapshot is not part of a transaction, so an entailed query sees the committed state at the time it built.

7 SEMANTIC SEARCH IN THE ENGINE

Order is not similarity. A sorted index answers a bound prefix with one range scan, and it is tempting to hope that terms close in meaning are close in the order.

LEMMA 7.1 (BANDWIDTH). *Let G be a graph with N vertices and diameter D . In every linear order of the vertices some edge joins vertices at least $(N-1)/D$ positions apart.*

For the neighbor graph of a d -dimensional grid of $N = g^d$ points this is at least g^{d-1}/d positions, so a scan of width w cannot contain all neighbors unless w is that large. On the Gene Ontology

Table 2: Share of the 10 nearest neighbors (by cosine) found within $\pm W$ positions of a term, and the median position gap to the nearest neighbor. GO, 38,092 terms, 2,000 queries.

Order	gap	± 50	± 100	± 500	± 1000	± 5000
dictionary id (first seen)	1739	.195	.211	.258	.284	.438
hierarchy, deepest parent	45	.328	.377	.498	.561	.735
hierarchy, first parent	37	.319	.366	.481	.546	.726
label, alphabetical	1749	.310	.334	.379	.408	.558
random	10823	.003	.005	.024	.049	.242

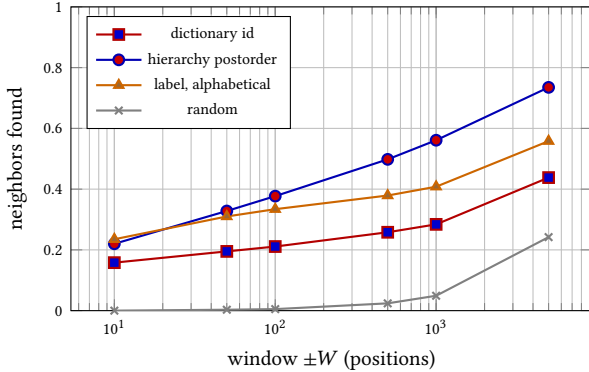


Figure 2: The data of Table 2: the share of neighbors found within $\pm W$ positions, by order.

($N = 38,092$ terms, 384-dimensional embeddings of the names by all-MiniLM-L6-v2 [32]) the best order we tried, the postorder of the class hierarchy, finds only 38% of a term’s ten nearest neighbors within 100 positions, and ONTODB’s first-seen ids 21%. The order serves structure, not similarity, so ONTODB adds vector indexes.

Vector indexes in the engine. One unit vector $\mathbf{v}(t)$ per term is held in memory, and four index families are available: exact (flat) search, IVF [18], product quantization (PQ), and HNSW [22]. IVF clusters the vectors into v lists and reads the ρ nearest; its lists are stored as *key ranges of the B+ tree that holds the triples*, the key (list, row, 0), so a list is a prefix scan and lists page, cache, latch, and update like every other index of the engine.

Filtering by a class. A filtered query $\text{near}(q, c, k)$ asks for the k terms of $\text{desc}(c)$ nearest to q . *Pre-filter* resolves c by interval scans and scores its members exactly; *post-filter* asks an index for more than k and keeps those below c (a binary search over $I(c)$); *masked search* gives the index the predicate “the row’s label lies in an interval of $I(c)$ ”, so nothing is resolved in advance. *Auto pre-filters* when $|\text{desc}(c)| \leq 4,096$ and otherwise masks HNSW. The filter is *correlated* with the vectors: the names of a class’s descendants are close to one another, so of the top 100 neighbors of a query from *inside* c , 4% to 72% lie under c , and from *outside* only 0% to 12%. The independence estimate $k/f(c)$ of the candidates needed, with $f(c) = |\text{desc}(c)|/N$, is wrong in both directions: post-filtering is cheap for an inside query and fails for an outside one. Testing membership with $I(c)$ per row touched, with no scan, makes masked search 6 times faster than resolving the filter first.

Table 3: Selected results: how many times faster ONTODB is than each other engine (the time of the other engine divided by ONTODB’s; for throughput, ONTODB’s divided by the other’s). All tables of this section give every measurement.

	Ox. RocksDB	Ox. memory	Jena TDB2	SQLite
pattern ($s, \cdot, \cdot$)	8.4×	3.4×	36.7×	4.2×
pattern ($s, p, \cdot$)	7.2×	4.6×	27.3×	5.5×
LUBM query 1	3.7×	1.4×	58.4×	2.0×
instances of a class, 10–99 below	39.1×	7.7×	47.9×	14.9×
instances of a class, 1000+ below	5.8×	2.2×	5.3×	3.5×
lookups per second, 8 threads	7.0×	3.2×	6.1×	3.9×
lookups per second, 8 readers + 1 writer	10.2×	4.8×	18.8×	4.3×

8 A BUILD FOR THE BROWSER

The engine compiles with Emscripten to a 0.85 MB WebAssembly module [16] that runs the same indexed store, optimizer, and semantic extension in a web page, after replacing two parts: the raptor C library by a Turtle and N-Triples parser of about 500 lines (RDF/XML needs the native build), and the threads (a disk scheduler, a deadlock detector, a worker per session) by inline execution on one session. The buffer pool, the page file in the browser’s memory file system, the trees, and the plans are unchanged. In a desktop browser, for an 834-class slice of the Gene Ontology with its embeddings, building the interval labels takes 3 ms, the instance query for *cell death* (97 classes, one range scan) 0.3 ms, a filtered similarity query 0.8 ms, and the IVF and HNSW indexes 58 and 98 ms. Updates are journalled in local storage and replayed on load, which is not durability. We did not use WebGPU: a query scores a few hundred vectors, under a millisecond with 128-bit SIMD, less than a round trip to a GPU.

9 EXPERIMENTAL STUDY

We evaluate ONTODB along four research questions.

- **[RQ1] The store:** how does a B+ tree store of three permutations compare with an LSM, a Java B+ tree, and a relational store on triple patterns, queries, and loading?
- **[RQ2] Hierarchies and the plan rewrite:** what do the interval labels and the entailment rewrite cost and buy against materializing, rewriting, and other engines’ subclass-path queries?
- **[RQ3] Concurrency:** how do lookups and inserts scale with threads, and what do readers see while a writer runs?
- **[RQ4] Semantic search:** what do the vector indexes and the filtering strategies give on real embeddings?

Table 3 summarizes the answers.

Setup. We compare ONTODB (release build, B+ tree store with SPO, POS, OSP, a 256 MiB pool, automatic joins) with Oxigraph 0.5.13 [29] (Rust API; RocksDB on disk, and its in-memory store), Apache Jena TDB2 5.2.0 [3] (Java 17, 4 GiB heap), and SQLite 3.51 as a triple table of three string columns with three indexes, a basic graph pattern translated into self-joins and ANALYZE after loading, all in-process on an Apple M2 Pro with 16 GiB. The data are LUBM [14] at one and ten universities (1.3 million statements), with its ontology where entailment is needed, and the Gene Ontology [4] (38,092 classes, 57,456 is_a edges, 384-d embeddings [32]) with 200,000 synthetic entities of one to three types. Queries are parsed once and run once to warm the caches (three times for the

JVM and SQLite), then five to nine times; we report the median, with every answer converted to its string and row counts agreeing across engines; times are warm, including ONTODB’s decode cache. Concurrency uses threads (processes with one connection each for SQLite).

What the numbers do not show. Every time is warm, with ONTODB’s decode cache on; cold, the string-level class-instance query is 2.0 times slower (Table 7) and still 1.1 times faster than the best other engine. SQLite runs through Python’s driver, whose call overhead is a few microseconds; Jena’s JVM is warmed; ONTODB’s inserts in RQ3 bypass the write-ahead log while the other engines commit with their defaults; and the instances of the Gene Ontology are synthetic. The machine has 8 performance cores, so eight threads fill it. ONTODB implements a SPARQL subset and RDFS subclass entailment only, where Oxigraph implements SPARQL 1.1, so the comparison is on basic graph patterns, the part the systems share.

9.1 RQ1: The Store

Selective patterns. For patterns that bind the subject ONTODB takes 3 and 2 μ s and the others 9 to 117 μ s: 3.4 to 37 $\times$ faster on $(s, \cdot, \cdot)$ and 4.6 to 27 $\times$ on $(s, p, \cdot)$, and 1.4 to 58 $\times$ faster on the selective LUBM queries. **Large results.** For patterns returning 16,000 to

71,000 rows ONTODB takes 10 to 71 ms: 1.9 to 3.2 $\times$ faster than Oxigraph on RocksDB, 1.5 to 2.4 $\times$ faster than Jena, and within a factor of two of SQLite, whose string index returns spellings without a dictionary, and of Oxigraph’s in-memory store. On the LUBM triangle query SQLite takes 4.0 ms, ONTODB 8.2 ms, and Oxigraph on RocksDB 275 ms (33 $\times$ slower than ONTODB). **Loading.** ONTODB is

the slowest to load LUBM(10), 11.4 s against 2.0 s for Oxigraph on RocksDB, 6.1 s for Jena, and 8.1 s for SQLite, with a peak of 584 MiB against 78 MiB for Oxigraph, because it parses RDF/XML through a C library, interns each term through two B+ trees, and builds exact statistics; its file is the smallest, 134 MiB against 176 for Oxigraph on RocksDB, 363 for Jena, and 706 for SQLite.

Design choices. Three permutations cut the page reads of $(\cdot, p, \cdot)$ by 7.9 $\times$ and of $(\cdot, \cdot, o)$ and $(\cdot, p, o)$ by 40 $\times$ against SPO alone, and the other three trees cost 66% more space and change no pattern (Table 6). The decode cache makes the subject-bound pattern 4.3 $\times$ faster, patterns returning thousands of rows 1.5 to 2.3 $\times$, and the largest instance queries 2.0 $\times$ (Table 7); a cheaper id-to-spelling path would help every cold query.

9.2 RQ2: Hierarchies and the Plan Rewrite

Interval labels. The Gene Ontology needs 76,570 intervals for 38,092 classes, 2.01 per class (at most 741, within Lemma 5.2), built in 15 ms, against 393,448 ancestor pairs for the closure. On 200,000 typed entities, materializing the closure turns 495,552 triples into 4,017,973 (8.1 $\times$), the file from 63.6 to 309.5 MiB, and the load from 5.1 to 180 s; the interval index adds 9.3 MiB (15%) and 0.17 s. For classes with 1,000 or more descendants it reads 265 pages against 182 (materialized) and 12,803 (one scan per class).

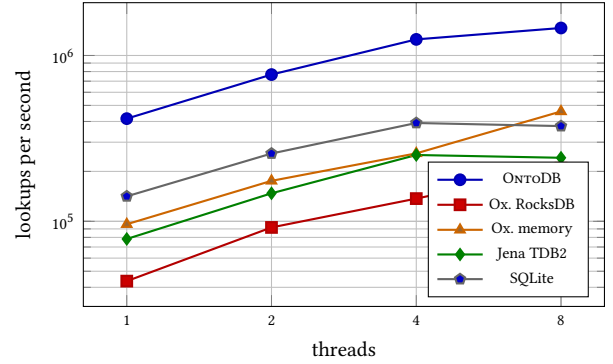


Figure 3: Lookups per second on LUBM(10) against threads (log-log).

Against other engines. Table 9 gives the instances of a class and its subclasses. The interval scan alone returns ids, in 1.9 ms for the 29,904 answers of the largest classes; returning strings takes 11.5 ms because each id is decoded. At the string level ONTODB is 2.2 $\times$ faster than the best other engine on the largest classes and 5 to 9 $\times$ faster than Oxigraph in memory on classes with fewer than 1,000 instances, 3.5 $\times$ faster than SQLite’s recursive query, and 5.3 to 5.8 $\times$ faster than Jena and Oxigraph on RocksDB.

The rewrite. Table 10 runs the seven LUBM queries that need only subclass reasoning on LUBM(10) with its ontology. Without the rewrite ONTODB returns no rows for six of them; with it, every query returns exactly the rows of the materialized store and of the other engines’ subclass-path queries, in times within 20% of the materialized store (Table 11), with no stored entailed triple, 16 times faster to prepare (0.06 s against 0.97 s) and 8 times smaller (5.1 against 40.7 MiB). Oxigraph on RocksDB is 1.9 to 6.9 times slower than the rewrite and Jena 2 to 9,300 times slower, with LUBM 9 timing out; Oxigraph’s in-memory store is faster on five of the seven queries, by 1.1 to 2.7 times.

9.3 RQ3: Concurrency

We run one to eight threads of lookups (the $(s, \cdot, \cdot)$ and $(s, p, \cdot)$ queries), of inserts of new triples with scattered subjects, and of readers beside one writer, for three seconds each on LUBM(10).

Readers. ONTODB serves 416 thousand lookups per second with one thread and 1.46 million with eight, 3.5 times more, and its one thread already beats every other engine at eight except Oxigraph in memory (460 thousand); at eight threads it serves 7.0, 3.2, 6.1, and 3.9 times the lookups of Oxigraph on RocksDB, Oxigraph in memory, Jena, and SQLite. **Beside a writer.** Its eight readers still serve 1.39 million lookups per second while the writer inserts 111 thousand triples per second, 4.3 to 18.8 times the readers of the other engines. **Writers.** ONTODB’s insert rate grows 3.6 times, from 264 to 950 thousand per second; its inserts bypass the write-ahead log here, while the others commit with their default durability, and Oxigraph’s in-memory store inserts faster with one thread (679 thousand). Heavy queries scale less: a 16,000-row query goes from 148 to 330 per second with eight threads, bound by memory traffic.

Table 4: Query times on LUBM(10): the mean over 100 sampled constants per pattern shape (all 18 predicates for $(\cdot, p, \cdot)$), and the median of the four LUBM queries that return rows without inference. Row counts agree across engines. Bold marks the best; a blue cell marks ONTODB as the best.

Query	rows	ONTODB	Ox. RocksDB	Ox. memory	Jena TDB2	SQLite
$(s, \cdot, \cdot)$	7	3 μs	27 μ s	11 μ s	117 μ s	13 μ s
$(s, p, \cdot)$	2	2 μs	14 μ s	9 μ s	52 μ s	11 μ s
$(\cdot, \cdot, o)$	16,233	11.0 ms	35.7 ms	10.4 ms	23.7 ms	8.4 ms
$(\cdot, p, o)$	16,225	10.2 ms	19.4 ms	7.8 ms	24.8 ms	6.2 ms
$(\cdot, p, \cdot)$	70,720	70.8 ms	161.2 ms	41.2 ms	107.5 ms	35.5 ms
LUBM 1	4	9 μs	32 μ s	12 μ s	508 μ s	18 μ s
LUBM 3	6	11 μs	37 μ s	13 μ s	234 μ s	18 μ s
LUBM 2 (triangle)	28	8.2 ms	275.1 ms	34.1 ms	120.1 ms	4.0 ms
LUBM 14 (scan)	75,547	46.0 ms	88.1 ms	40.8 ms	138.7 ms	29.3 ms

Table 5: Bulk loading LUBM(10) (1.3 million triples). ONTODB counts statements read; the others count distinct triples. Bold marks the best; a blue cell marks ONTODB as the best.

Engine	triples	load (s)	disk (MiB)	RSS (MiB)
ONTODB	1,316,700	11.41	133.7	584
Ox. RocksDB	1,272,953	2.04	176.3	78
Ox. memory	1,272,953	2.44	—	563
Jena TDB2	1,272,953	6.13	363.0	1447
SQLite	1,272,953	8.08	706.1	32

Table 6: Page reads per pattern on LUBM(1) (cold 64-frame pool, bulk-loaded) and file size, by set of permutations.

Permutations	$(\cdot, p, \cdot)$	$(\cdot, \cdot, o)$	$(\cdot, p, o)$	file (MiB)
SPO	599	599	599	6.1
SPO, POS, OSP	76	15	15	10.8
all six	76	15	15	17.9

Table 7: The effect of ONTODB’s decode cache (65 536 entries) on the time to return strings: mean time per query on LUBM(10), and the instances of a class on the Gene Ontology (30 classes per row).

Workload	answers	cache off	cache on
$(s, \cdot, \cdot)$	7	13 μ s	3 μ s
$(s, p, \cdot)$	2	3 μ s	2 μ s
$(\cdot, \cdot, o)$	16,233	24.9 ms	11.0 ms
$(\cdot, p, o)$	16,225	14.9 ms	10.2 ms
$(\cdot, p, \cdot)$	70,720	113.1 ms	70.8 ms
instances, 100-999 below	2,954	2.4 ms	258 μ s
instances, 1000+ below	29,904	22.6 ms	11.5 ms

Table 8: Interval labels for the Gene Ontology.

Spanning-forest rule	non-tree edges	intervals	mean / p99 / max	build
deepest parent	19,367	76,570	2.01 / 17 / 741	14.9 ms
first parent	19,367	87,769	2.30 / 22 / 740	14.4 ms

9.4 RQ4: Semantic Search

On 38,092 Gene Ontology terms and 1,000 queries (Table 13, Figure 4), HNSW ($M=16$) reaches recall 0.993 at 44 μ s, 51 times faster

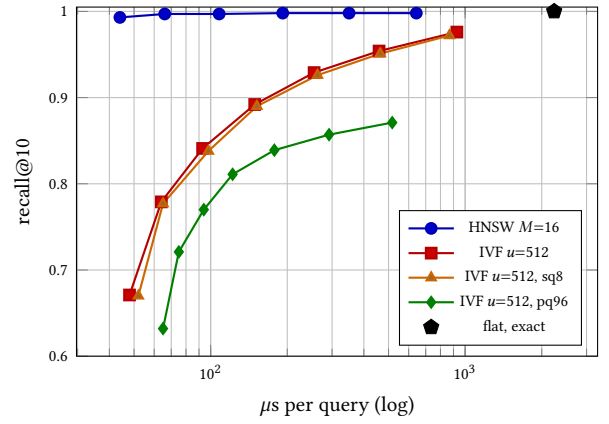


Figure 4: Recall@10 against time per query on GO (38,092 vectors, 384 dimensions).

than the 2.2 ms exact scan, for 3.9 MiB of links; IVF over the B+ tree ($v=512$) needs 255 μ s for 0.929 and 928 μ s for 0.976, about 2.7 $\times$ the cost per scored row of a contiguous scan, the price of lists that page, cache, and update with the triples. Our first exact baseline used a scalar loop the compiler would not vectorize and took 8.3 ms, which would have doubled every speed-up; we report against the vectorized 2.2 ms.

Exact pre-filtering for small classes, masked search for large. Up to about 1,000 members, resolving the members through the intervals and scoring them exactly takes 3 to 62 μ s and is the exact answer, 10^2 to 10^4 times faster than any index strategy. Post-filtering fails for outside queries (IVF recall 0.07 on the smallest classes, 0.64 on the largest); above 5,000 members masked HNSW takes 105 to 900 μ s, exact, against 1.4 ms for resolving and scoring. With a threshold of 4,096 members, auto matches the fastest strategy in every cell but one and is never slower than 0.9 ms; the cell it loses (269 μ s against 70 μ s) avoids a factor of 79 in its neighbor.

10 RELATED WORK

RDF stores. Hexastore indexes all six permutations [33]; RDF-3X keeps six compressed B+ trees [26]; QLever builds read-mostly permutation indexes [5]; HDT is a compressed read-only format [10]; Virtuoso stores quads in a relational engine [9]. Apache

Table 9: Instances of a class and its subclasses on the Gene Ontology with 200 000 entities: mean time over 30 classes per row. ONTODB answers by interval scans: “ids” stops before the dictionary, “strings” decodes every answer as the other engines do. The others evaluate the subclass path `rdfs:subClassOf*` (SQLite: a recursive query). Answer counts agree on every class. Bold marks the best; a blue cell marks ONTODB as the best.

Subclasses below <i>c</i>	answers	ONTODB ids	ONTODB strings	Ox. RocksDB	Ox. memory	Jena TDB2	SQLite
1	10	1 μ s	2 μs	38 μ s	15 μ s	281 μ s	27 μ s
2-9	40	3 μ s	3 μs	94 μ s	26 μ s	249 μ s	44 μ s
10-99	278	11 μ s	15 μs	588 μ s	115 μ s	722 μ s	225 μ s
100-999	2,954	131 μ s	258 μs	6.3 ms	1.3 ms	4.3 ms	3.1 ms
1000+	29,904	1.9 ms	11.5 ms	66.8 ms	25.6 ms	61.6 ms	40.7 ms

Table 10: The LUBM queries that need subclass reasoning, on LUBM(10) with the ontology: median time. ONTODB entailing answers the original query with `\set entail on`; ONTODB materialized stores the entailed type triples; the others run the query rewritten with the subclass path. Row counts agree (the column “no entailment” is what ONTODB returns without it). Bold marks the best; a blue cell marks ONTODB as the best.

Query	rows	no entailment	ONTODB entailing	ONTODB materialized	Ox. RocksDB	Ox. memory	Jena TDB2
LUBM 3	6	6	10 μs	12 μ s	53 μ s	16 μ s	933 μ s
LUBM 4	34	0	181 μ s	172 μ s	630 μ s	136 μs	2.0 ms
LUBM 6	75,547	0	42.3 ms	44.6 ms	89.6 ms	38.6 ms	85.4 ms
LUBM 7	59	0	108 μ s	96 μ s	555 μ s	93 μs	997.0 ms
LUBM 8	5,916	0	17.8 ms	14.9 ms	95.4 ms	14.6 ms	423.8 ms
LUBM 9	1,233	0	199.0 ms	178.9 ms	380.5 ms	73.2 ms	timeout
LUBM 10	0	0	8 μs	7 μ s	54 μ s	16 μ s	174 μ s

Table 11: What it costs to answer subclass-entailed type patterns on LUBM(10): time to prepare, type triples added to the store, and bytes added.

Strategy	prepare (s)	triples added	space added (MiB)
materialize the closure	0.97	184,628	40.7
interval index	0.06	0	5.1

Table 12: Concurrency on LUBM(10): lookups per second with 1 and 8 reader threads, readers’ lookups per second with one writer running, and inserts per second with 1 and 8 writer threads (one triple per transaction; ONTODB’s inserts are not logged, and Jena’s commit forces the journal to disk). The machine has 8 performance cores. Bold marks the best; a blue cell marks ONTODB as the best.

Engine	lookups/s		readers + 1 writer		inserts/s	
	1	8	1	8	1	8
ONTODB	416k	1.46M	398k	1.39M	264k	950k
Ox. RocksDB	44k	209k	38k	136k	91k	105k
Ox. memory	96k	460k	69k	288k	679k	186k
Jena TDB2	78k	242k	22k	74k	16	15
SQLite	141k	375k	111k	322k	15k	14k

Jena TDB2 uses B+ trees [3], RDF4J’s native store B-trees [8], and Oxigraph [29] a log-structured merge tree in RocksDB. ONTODB follows the B+ tree line [6, 7] with three permutations; its latching and recovery [12, 20, 23] are textbook, and the contribution is the choice of parts, their measured behavior on ontologies, and what is built on top.

Table 13: Vector indexes on GO (recall@10, mean time per query, memory of the structure, build time).

Index	setting	recall	μ s	MiB	build (s)
flat float32	exact	1.000	2,238	55.8	—
flat 8-bit	exact over codes	0.996	4,554	14.1	—
flat PQ $m=48$	codes only	0.778	819	1.7	10.4
flat PQ $m=48$	re-rank 100	0.998	905	1.7 [†]	10.4
IVF $v=512$	$\rho=8$	0.892	149	56.5	17.5
IVF $v=512$	$\rho=16$	0.929	255	56.5	17.5
IVF $v=512$	$\rho=64$	0.976	928	56.5	17.5
IVF $v=512$, 8-bit	$\rho=16$	0.926	263	14.8	17.5
IVF $v=512$, PQ 96	$\rho=16$	0.839	178	4.2	38.4
HNSW $M=16$	$ef=10$	0.993	44	59.7	15.0
HNSW $M=16$	$ef=40$	0.997	108	59.7	15.0
HNSW $M=32$	$ef=160$	0.999	474	63.4	28.0

[†] re-ranking reads the float vectors as well.

Table 14: Ten nearest terms under a class *c*: recall@10 / μ s per query, for a query inside or outside *c* and classes with at most 10 or more than 5,000 members (Gene Ontology). Bold marks the fastest exact or near-exact strategy.

Strategy	query inside <i>c</i>		query outside <i>c</i>	
	≤ 10	$> 5,000$	≤ 10	$> 5,000$
pre-filter, exact	1.00 / 3	1.00 / 1,404	1.00 / 3	1.00 / 1,394
post-filter, IVF	0.86 / 903	0.99 / 211	0.07 / 966	0.64 / 591
masked HNSW	1.00 / 16,653	1.00 / 105	1.00 / 16,802	1.00 / 900
auto (threshold 4,096)	1.00 / 3	1.00 / 95	1.00 / 3	1.00 / 860

Inference. RDFS entailment is usually *rewritten* into the query or *materialized* with maintenance by counting or delete-and-rederive [15, 24]. Agrawal, Borgida, and Jagadish [1] store a

Table 15: Stores for ontologies, as documented (October 2026). We ran ONTODB, Oxigraph, Jena TDB2, and SQLite.

System	Triples are kept in	Hierarchy by	In a browser
Oxigraph [29]	RocksDB (LSM)	query only	in-memory
Jena TDB2 [3]	B+ trees	reasoner layer	no
RDF4J native [8]	B-trees	materialized	no
RDF-3X [26]	6 compressed B+ trees	query only	no
RDFox [24]	in memory	materialized	no
ONTODB	3 B+ trees	intervals	full engine

compressed transitive closure by numbering a spanning forest in postorder with a set of intervals per node, and Grust [13] numbers XML trees so that axes become regions of a plane; we use the former on class DAGs inside a B+ tree store and make it the target of a plan rewrite.

Vector search. Approximate nearest neighbor search has inverted files and product quantization [18], proximity graphs such as HNSW [22], and disk-oriented graphs such as DiskANN [17]; FAISS packages the first two [19], and Pan, Wang, and Li survey the hybrid queries that are hard for them [30]; ACORN makes HNSW traverse only nodes that pass a predicate [31]. Neo4j, Neptune Analytics, Fluree, and GraphDB add vector or similarity indexes to graph and RDF stores [2, 11, 25, 28]; in the documentation we read, none restricts the search with range-scannable hierarchy intervals in the engine that holds the triples.

Adaptive radix trees and browsers. An ART [21] orders keys by byte, so its range scan is lexical and Lemma 7.1 applies; sibling links are awkward there, because lazy expansion moves a leaf, and ONTODB’s three fixed-width integers leave an ART little to compress. WebAssembly [16] lets a C++ engine run in a page; Oxigraph ships an in-memory JavaScript build, and ONTODB compiles the whole paged engine.

11 CONCLUSION

An ontology store is asked for patterns, hierarchies, and meaning. ONTODB answers all three inside one disk-based, multi-threaded engine: a B+ tree store of three permutations whose latch-free hits and optimistic writers scale to eight threads; interval labels that make “everything under c ” a few range scans, which a plan rewrite uses to answer RDFS entailment with the answers of materialization and none of its storage; vector indexes whose lists live in the same tree; and a 0.85 MB build for the browser. It answers selective patterns and class instances faster than Oxigraph, Jena TDB2, and SQLite and serves 3 to 7 times their lookups per second under eight threads.

Limitations. ONTODB implements a SPARQL subset without OPTIONAL, UNION, property paths, or aggregates, its page-level recovery is exercised by a crash harness while its durable database mode is a snapshot and logical log over the in-memory backend, and it entails only `rdfs:subClassOf` over constant classes; it is the slowest to load and uses much memory while loading; decoding an id costs about $0.7 \mu\text{s}$, which the cache hides on a warm store; labels and the entailment snapshot are rebuilt, not maintained, after updates; and the vectors come from one ontology and one small model, with synthetic instances.

REFERENCES

- [1] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. 1989. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 253–262. <https://doi.org/10.1145/67544.66950>
- [2] Amazon Web Services. 2023. Amazon Neptune Analytics: Vector Similarity Search. AWS documentation and launch announcement. <https://aws.amazon.com/blogs/aws/introducing-amazon-neptune-analytics-a-high-performance-graph-analytics>, accessed October 2026.
- [3] Apache Jena. [n.d.]. TDB2: a native triple store for Apache Jena. <https://jena.apache.org/documentation/tdb2/>. Accessed October 2026.
- [4] Michael Ashburner et al. 2000. Gene Ontology: Tool for the Unification of Biology. *Nature Genetics* 25 (2000), 25–29.
- [5] Hannah Bast and Björn Buchhold. 2017. QLever: A Query Engine for Efficient SPARQL+Text Search. In *Proceedings of the ACM Conference on Information and Knowledge Management*. 647–656.
- [6] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (1972), 173–189.
- [7] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137.
- [8] Eclipse RDF4J. [n.d.]. Eclipse RDF4J: a Java framework for RDF. <https://rdf4j.org/>. Accessed October 2026.
- [9] Orri Erling and Ivan Mikhailov. 2010. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Management*. Springer, 501–519.
- [10] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. 2013. Binary RDF Representation for Publication and Exchange (HDT). *Journal of Web Semantics* 19 (2013), 22–41.
- [11] Fluree PBC. 2026. Fluree Vector Search. Fluree documentation. <https://labs.fluree/docs/db/indexing-and-search/vector-search>, accessed October 2026.
- [12] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.
- [13] Torsten Grust. 2002. Accelerating XPath Location Steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 109–120.
- [14] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics* 3, 2–3 (2005), 158–182.
- [15] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
- [16] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.
- [17] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. 2019. DiskANN: Fast Accurate Billion-Point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [18] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.
- [19] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547.
- [20] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems* 6, 4 (1981), 650–670.
- [21] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 38–49.
- [22] Yu. A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.
- [23] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems* 17, 1 (1992), 94–162.
- [24] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. 2014. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *Proceedings of AAAI*.
- [25] Neo4j, Inc. 2026. Neo4j Vector Index and Search. Neo4j documentation. <https://neo4j.com/developer/genai-ecosystem/vector-search/>, accessed October 2026.
- [26] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal* 19, 1 (2010), 91–113.
- [27] Elizabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 297–306.
- [28] Ontotext. 2019. GraphDB 8.9 with Improved Semantic Similarity Search. Ontotext announcement. <https://sirna.com/news/graphdb-8-9-with-improved-semantic-similarity-search.html>.

- [29] Oxigraph contributors. [n.d.]. Oxigraph: a SPARQL graph database written in Rust. <https://github.com/oxigraph/oxigraph>. Version 0.5; accessed October 2026.
- [30] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of Vector Database Management Systems. *The VLDB Journal* 33 (2024), 1591–1615.
- [31] Liana Patel, Peter Kraft, Carlos Guestrin, and Matei Zaharia. 2024. ACORN: Performant and Predicate-Agnostic Search Over Vector Embeddings and Structured Data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 120.
- [32] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks. In *Proceedings of EMNLP-IJCNLP*.
- [33] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: Sex-tuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1008–1019.

APPENDIX

A ALGORITHMS

The algorithms that the text describes without listing, in the notation of Table 1.

A.1 Choosing a tree for a pattern

Algorithm 8: SCANFOR(s, p, o): choose a tree for a triple pattern

```

1 for each permutation  $\pi \in \{\text{SPO}, \text{POS}, \text{OSP}\}$  do
2    $\ell(\pi) \leftarrow$  the number of leading positions of  $\pi$  that the pattern
   binds;
3  $\pi^* \leftarrow \arg \max_{\pi} \ell(\pi)$  (the first on a tie);
4  $x \leftarrow$  the bound values in the order of  $\pi^*$ , padded with zeros;
5 return BEGIN(tree( $\pi^*$ ),  $x, \ell(\pi^*)$ ), keeping the triples that match
   the pattern;
```

A.2 Range scan

Algorithm 9: BEGINRANGE(a, b), TAKE, and SETTLE

```

1  $P \leftarrow \text{LEAFFOR}(a)$ ;  $i \leftarrow$  the first position in  $P$  with key  $\geq a$ ;
2 TAKE( $P, i$ ); SETTLE();
3 Take( $P, i$ ):  $\text{buf} \leftarrow \langle P.\text{keys}[j] : j \geq i, \text{key} < b \rangle$ ;  $\text{pos} \leftarrow 0$ ;
4  $\text{next} \leftarrow \text{nil}$  if some key  $\geq b$  was met, else  $P.\text{next}$ ; release  $P$  (latch
   and pin);
5 Settle(): while  $\text{pos} \geq |\text{buf}|$  do
6   if  $\text{next}$  is nil then become the end iterator; return;
7   TAKE(fetch( $\text{next}$ ), 0); // one sibling link, no climb to
   the root
8 Advance:  $\text{pos} \leftarrow \text{pos} + 1$ ; SETTLE();
```

A.3 Bulk load

Algorithm 10: BULKLOAD(X) for a sorted set X of unique keys

```

1 cut  $X$  into runs of  $B$  keys; if the last run has fewer than  $\lfloor B/2 \rfloor$ , it
   and the run before it share their keys evenly;
2 write each run as a leaf and link it to the previous one;  $\Lambda \leftarrow$  the
   list of (first key, page id);
3 while  $|\Lambda| > 1$  do
4   cut  $\Lambda$  into runs of  $F+1$  children, again sharing the last two
   runs evenly if the last is too short;
5   write each run as an internal page whose separators are the
   first keys of children 2, 3,  $\dots$ ;  $\Lambda \leftarrow$  the (first key, id) of the
   new pages;
6 set the root in the header to the page of  $\Lambda[0]$ ;
```

B PROOFS

PROOF OF LEMMA 3.1. The descent ends at the first leaf, which is one of the h pages. Every leaf after it holds at least $\lfloor B/2 \rfloor$ keys, all inside the range except possibly the last, so at most $\lceil r/\lfloor B/2 \rfloor \rceil$ more leaves are read. A bulk-loaded leaf is full, which replaces $\lfloor B/2 \rfloor$ by B . $\square$

PROOF OF LEMMA 5.1. By induction over the order in which Algorithm 6 builds I : the numbers covered by $I(c)$ are exactly those of the classes in desc(c). $\square$

PROOF OF LEMMA 5.2. A class $u \in \text{desc}(c)$ is reached from c by a path. If the path uses no non-tree edge, u is in the forest subtree of c . Otherwise let ϵ be the last non-tree edge on it; u is then in the forest subtree of the head of ϵ . So the covered set is the forest subtree of c together with the forest subtrees of the heads of the edges of $E'(c)$, a union of $1 + |E'(c)|$ intervals. $\square$

PROOF OF LEMMA 6.2. The triple is in $\mathcal{T}^+$ iff e has a stored type c'' with $c'' \in \text{desc}(c)$. The label tree holds $(po(c''), e)$ for every stored type of every entity, so by Lemma 5.1 e is returned by the range scans of $I(c)$ iff $po(c'')$ lies in an interval of $I(c)$ for some stored type c'' , that is, iff $c'' \in \text{desc}(c)$. A bound subject is tested with the same condition on its stored types. $\square$

PROOF OF LEMMA 7.1. The first and last vertices of the order are $N - 1$ apart and joined by a path of at most D edges, so some edge of the path covers at least $(N - 1)/D$ positions. $\square$