

OntoDB: A Fast and Persistent Query Engine for Ontologies (Extended Version)

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 (Extended Version). 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 [5] 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, such as the resources of one type, and is the base case of every RDF store [28, 36]. 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 has no answer in the data and needs a similarity measure over embeddings. The three meet in queries such as “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* at write time, which stores every inferred triple [26], or *rewritten* into the query, which scans once per subclass. On the Gene Ontology with 200,000 typed entities, materializing the closure 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.** Triple stores answer patterns with range scans over sorted permutations. It is tempting to believe that neighbors in meaning are neighbors in the order; we prove they cannot all be (Section 5), and measure how few are. Similarity is therefore left to a vector database next to the triple store, with its own storage, its own paging, and no view of the hierarchy. **C3. A class filter is correlated with the vectors.** Filtered vector search is usually costed by assuming that the filter is independent of the similarity. In an ontology it is not: the neighbors of a term lie under the term’s own ancestors. A post-filtered index can then find 7% of the right answers, and the independence estimate of the work it must do is wrong in both directions (Section 8). **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, and inserts from 184 thousand to 23 thousand: a store embedded in a service or a notebook that answers many queries at once spent its time waiting for itself.

Example 1.1. Let q be the term *apoptotic process* in the Gene Ontology and c the class *cell death*. The query asks for the five terms nearest to q in embedding space among the 97 classes at or below c in an 834-class slice of the ontology. A triple store alone cannot

doi:XX.XX/XXX.XX

rank by meaning. A vector index alone returns neighbors from anywhere in the ontology, and a post-filter keeps those under c only if enough of them are. ONTODB finds the 97 classes with one range scan of the same 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.

Database systems have handled a similar split for decades: an access method is added to the engine that owns the storage, paging, and logging, and not beside it. This paper takes that view of ontologies.

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). Three permutations behind a buffer pool, with the algorithms for search, insert, and delete, a bound on the pages a range scan reads, and a latch-free hit path and an optimistic writer descent that scale to eight threads.
- (2) **Hierarchies as intervals** (Section 6). 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 7). RDFS subclass entailment of `rdf:type` patterns as interval scans the optimizer costs and joins like any other, with the answers of materialization, prepared 16 times faster and in 8 times less space.
- (4) **Vector indexes and filtered search** (Sections 5 and 8). 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 between pre-filtering and masked search.
- (5) **A build for the browser** (Section 9). The whole engine, with its paged trees and semantic search, as a 0.85 MB WebAssembly module.

We evaluate ONTODB against Oxigraph [32], Apache Jena TDB2 [3], and SQLite on LUBM [15] and the Gene Ontology (Section 10, summarized in Table 4). The code, data generators, and benchmark programs are available 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 [28, 36], 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 $c' \in desc(c)$.

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
$inst(c), \text{near}(q, c, k)$	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 5.1)

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* $inst(c)$, 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.

3 THE ONTODB STORE

ONTODB is a single-process engine with the layers of a conventional disk-based DBMS (Figure 1). A SPARQL query is parsed, bound against the dictionary, planned as a left-deep tree, re-ordered by the optimizer, and run by volcano-style executors that pull rows of term ids. The executors read triples through a store interface. The *indexed store*, which this paper is about, keeps each triple in three B+ trees in one page file, behind a buffer pool with LRU-K eviction [30]; terms are encoded by a dictionary built from the same parts, a heap of spellings and two B+ trees. Updates are logged page by page for ARIES-style recovery [25]. The semantic

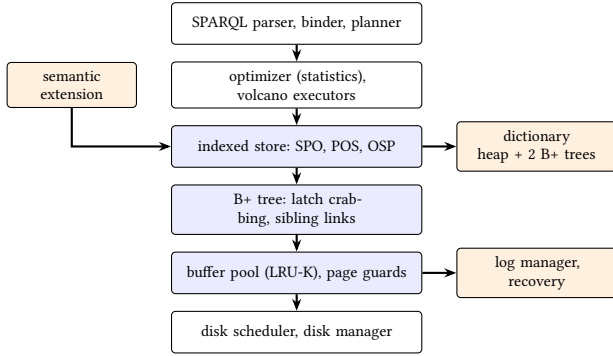


Figure 1: The layers of ONTODB. Blue boxes are the storage structures of Section 3.

Table 2: The permutation π that answers each pattern, and the length $\ell(\pi)$ of the bound prefix.

Pattern	$(s, \cdot, \cdot)$	$(\cdot, p, \cdot)$	$(\cdot, \cdot, o)$	$(s, p, \cdot)$	$(\cdot, p, o)$	$(s, \cdot, o)$	(s, p, o)
$\pi, \ell(\pi)$	SPO, 1	POS, 1	OSP, 1	SPO, 2	POS, 2	OSP, 2	SPO, 3

extension of Sections 6 and 8 sits beside the store and reads it through the same interface.

3.1 Keys, Pages, and 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 with $F+1$ child ids. Leaves are linked to their *right* sibling only. A header page names the root, so a root split changes one word. Every page but the root keeps at least half its maximum: a leaf at least $\lfloor B/2 \rfloor = 84$ keys, an internal page at least $\lfloor F/2 \rfloor + 1 = 73$ children.

Three permutations are enough. A pattern binds some of s, p, o , and with the permutations SPO, POS, and OSP every pattern has one in which its bound positions form a prefix (Table 2); it is then one range scan that returns only matching triples. Algorithm 6 (Appendix A) picks the permutation with the longest bound prefix. The other three permutations would make no pattern cheaper (Section 10).

3.2 Search

A point lookup and the start of every scan use one descent. CHILD-For picks the child whose range holds x : the last separator that is $\leq x$, or child 0 if none is. The descent *crabs*: it latches the child before it releases the parent, so a concurrent split cannot move x away between two steps [13, 22].

A lookup reads h pages. Every non-root leaf holds at least 84 keys and every non-root internal page has at least 73 children, so a tree of n keys has at most $n/84$ leaves and height $h \leq 2 + \log_{73}(n/84)$: at most four levels for the 1.3 million triples of LUBM(10), and three when the tree is bulk-loaded (about 7,800 full leaves under 54 internal pages).

3.3 Insert

Insert is *pessimistic* crabbing. It latches the header and then each page on the way down in write mode, and releases every latched

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  $P$  (binary
  search);

```

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;
    // split the leaf: the left half stays, the right half
    moves, its first key is copied up
11   $R \leftarrow$  new leaf with the upper half of  $P$ ;  $R.next \leftarrow P.next$ ;
     $P.next \leftarrow R$ ;  $\sigma \leftarrow R.keys[0]$ ;
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;
    // split the internal page: the middle separator
    moves up and leaves both halves
17    $P \leftarrow Q$ ;  $R \leftarrow$  new page with the upper half of  $Q$ ;  $\sigma \leftarrow$  the
    middle separator of  $Q$ ;

```

ancestor, the header included, 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, so a split never needs a latch it does not hold, and latches are taken in one order everywhere: the header, then 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, so the half-full invariant holds. Splits propagate only while the parent is full, so with random keys a split costs $O(1/B)$ page writes per insert on average. A split or a merge is the only update that spans pages, and Section 3.8 makes it atomic for recovery.

3.4 Delete

Delete crabs the same way with a different notion of safe: a page above its minimum (for the root, more than one key), because only then can it not underflow. After removing x , a leaf below $\lfloor B/2 \rfloor$

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: move the right page's entries into the left (a leaf
       keeps its sibling link; an internal page pulls the
       separator down); delete the right entry from  $Q$ ;
8   else
9     redistribute: move one entry from  $R$  to  $P$  (a leaf moves a
       key; an internal page rotates through  $Q$ ) 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;
```

keys repairs itself against a sibling under the same parent, preferring the left one.

A leaf emptied by a merge is *kept*, empty, with its sibling link still pointing on, rather than freed. An iterator may already hold its id (next subsection), and an empty page it can step through is cheaper than a protocol that tells every iterator the page is gone. Space reclaimed this way is not reused until the tree is rebuilt; freed internal pages are unreachable once their parent is updated and are released.

3.5 Range Scan

The half-open range $[a, b)$ is the primitive the semantic extension needs (Section 6); a prefix scan is a range between the smallest and the largest key with that prefix. BEGINRANGE descends to the leaf for a , finds the first key $\geq a$ by binary search, and hands the leaf to an iterator.

The iterator *copies* the keys of a leaf that lie below b , remembers the right sibling's id, and releases the leaf. It holds no latch and no pin between rows, so a consumer that stalls, such as a join waiting on its other input or a client reading slowly, blocks no writer and pins no frame. Reading the next leaf is one page read and no descent. The cost is the copy and a weaker guarantee: between two leaves a writer may change the tree, so a scan sees each leaf as it was when it was read, and the transaction layer's locks, not the latches, give a query its isolation. The iterator checks that keys increase along the chain and that a run of empty leaves does not cycle, and stops with an error otherwise.

LEMMA 3.1 (COST OF A RANGE SCAN). *A scan that returns 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 $h + \lceil r/B \rceil$ in a bulk-loaded tree.*

PROOF. 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 . $\square$

3.6 Bulk Load

Loading a file inserts hundreds of thousands of triples into an empty store, and inserting one at a time leaves leaves about half full. ONTODB instead sorts the triples once for each permutation and builds the tree bottom-up.

Each page is written once, in order, and no page is split. A bulk-loaded tree is full, which Section 10 measures.

3.7 Buffer Pool, Dictionary, and Query Processing

Buffer pool. Pages live in a fixed set of frames. A *page guard* pins a frame and takes its latch for the guard's lifetime, and its destructor releases both, so no path leaves a page pinned. A frame is evicted by LRU-K with $K=2$ [30], which ranks a frame by the time of its second most recent use and so is not flooded by one long scan. Reads and writes go through a disk scheduler.

Dictionary. Terms (IRIs, literals with a language or datatype, blank nodes) have one canonical spelling and get a dense 64-bit id in the order they first appear. The mapping is stored in pages: spellings in a heap, $id \rightarrow record$ in a B+ tree keyed (id, page, slot), and $spelling \rightarrow id$ in a B+ tree keyed (hash, id) with a 64-bit FNV-1a hash; a lookup scans the hash prefix and compares the candidates' spellings, so a collision costs a comparison and not a wrong answer. Decoding an id costs a tree descent and a heap read, about $0.7 \mu s$, and a query that returns thousands of rows decodes thousands of ids. A direct-mapped cache of the 65,536 most recently decoded spellings (slot = id mod size) removes most of that on a warm store; ids are dense and a spelling never changes, so an entry is never stale. Section 10 says where the cache helps and where it does not.

Planning and joins. The planner builds a left-deep tree of triple scans in textual order and the optimizer reorders it greedily: it appends, at each step, a pattern that shares a variable with what is already joined (any pattern, if none does), the one whose estimated join output is smallest. The output of a join of inputs of a_1 and a_2 rows is estimated as $a_1 a_2 / \max(V_1, V_2)$, where V_1 and V_2 are the numbers of distinct values of the join variable, from exact counts that the store collects at load for every position, every pair of positions, and every triple. The join operator is a nested loop, an index nested loop (the inner scan is reopened with the outer row's bindings as constants, so it is again a prefix scan of one tree), or a hash join, chosen per node. The SPARQL subset is basic graph patterns with FILTER, DISTINCT, ORDER BY, LIMIT, OFFSET, and INSERT DATA and DELETE DATA; OPTIONAL, UNION, property paths, and aggregates are not implemented.

3.8 Transactions and Recovery

Updates run under strict two-phase locking with a deadlock detector. Persistence has two paths: 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, and the page store has the page-level log described here, exercised by a crash harness; wiring the page-level log into the facade is future work. Recovery of the page store is

page-oriented: when a dirty page is released, the pool appends a record holding the page’s image and stamps the page with its log sequence number, and before a page is written the pool forces the log to that stamp. Redo therefore copies page images whose stamp is older than the record and needs no knowledge of the B+ tree layout. Undo of a triple is *logical* and writes a compensation record, because other transactions may have changed the page since. A split or merge is the one update that spans pages: insert and remove open a *page group* that holds the log back from the group’s first record until it ends, and the pool does not choose a page of the group as a victim, so a crash never leaves half a split on disk. Commit forces the log, and checkpoints are sharp (flush every page, then write the record). A crash harness kills the process between the pages of every multi-page operation and checks the tree afterwards: sorted keys, separators consistent with leaves, a leaf chain that visits every key once, and the half-full bound.

4 CONCURRENCY

An ontology store embedded in a service answers many queries at once while facts arrive. Strict two-phase locking keeps transactions isolated (Section 3.8); this section is about the *latches* that keep the pages and the trees sound, which decide how many threads the engine can use. Our first implementation followed the textbook: one latch over the buffer pool’s tables, a mutex over the dictionary, and a root-to-leaf chain of exclusive latches for every insert. Sampling a run of eight reader threads showed that four of five samples were threads waiting for a lock: 80% of the waits were on the pool’s latch and 20% on the dictionary’s mutex. Once those were gone the next waits were on the shard latch of the pages every query touches and on the page latches themselves (a `shared_mutex` keeps its reader count behind an internal mutex, so readers of one root page queue on it). We replaced each in turn.

4.1 An Optimistic Descent for Writers

A pessimistic insert latches every page on the way down in write mode until it reaches a page that cannot split, so every insert latches the root exclusively. Most inserts split nothing: a leaf holds 169 keys. Algorithm 9 (Appendix A) descends with read latches instead, takes the write latch of the leaf alone, and finishes if the leaf has room (for a delete, if it would stay at least half full). Otherwise nothing has been written, and the pessimistic algorithm of Section 3 runs from the start. The type of a child is read from its pinned page, which cannot change while it is pinned, so the leaf can be write-latched while its parent is still read-latched and no thread ever holds a write latch above the leaf unless a split or merge needs it.

The latch order stays top to bottom, so it cannot deadlock with the pessimistic path: a thread that holds a read latch on a parent and waits for the leaf is waiting for a thread that either finishes the leaf or, if it needs a split, already holds the leaf and does not need the parent again. Writers to the same leaf still serialize, as they must.

4.2 A Sharded Pool with a Latch-Free Hit

The buffer pool is split into S shards (16 in our measurements). A page belongs to shard $id \bmod S$, which owns an equal share of the

frames, its own page table, free list, and LRU-K replacer, and one latch. That alone removes most waiting, but the pages every query touches, the roots of the trees, all hash to a few shards, so a latch per shard is still a latch every thread takes at every query. A hit on a resident page therefore takes no latch. The shard keeps its page table also as an open-addressing array of atomic words (page id and frame), written under the shard latch and read without it (Algorithm 10 (Appendix A)). A reader finds the frame, 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 ; from then on no one can pin it, so a pinned frame is never evicted under a reader, and a pin that raced with an eviction lands on a frame that now holds another page, sees the mismatch, and lets go.

Recording every hit in the LRU-K history would put the shard latch back. A hit instead sets a *referenced* bit on the frame (a plain load when already set); when a shard must evict, it first folds the marked frames into the replacer. Eviction order is then the LRU-K order of the accesses the replacer has seen (every miss, and every marked frame once per eviction), a coarser history than the textbook’s. A pool with one shard, the default, keeps the exact textbook behavior and takes none of these paths.

4.3 The Latch and the Dictionary

The page latch is one atomic word: 30 bits count readers, one bit says a writer is waiting (and keeps new readers out, so writers are not starved), one says a writer holds it. It spins briefly and then yields; a page latch is held for one page operation and never across I/O. Decoding a term id reads the dictionary’s trees, which latch their own pages, and a cache of decoded spellings. A dictionary-wide mutex serialized all of it; now only inserts are serialized, so that a spelling is never interned twice, and the cache is guarded by 64 stripe locks chosen by cache slot, so threads decoding different ids do not meet.

4.4 Correctness

The invariants are small. A frame’s pin count is -1 only while a thread that holds the shard latch owns the frame for eviction or deletion, and is restored on every exit; a page is in the atomic index if and only if it is in the page table; a leaf is changed only under its write latch; and the order of latches (header, then top to bottom, siblings left to right) is the same on every path. Two stress tests run eight reader threads against a pool of 64 frames and 400 pages whose bytes name their page, with hits and evictions mixed, and four inserters, two deleters, and two scanners against a tree of tiny nodes, checking order, invariants, and contents. The whole suite of 185 tests passes in release mode, under ThreadSanitizer, and under AddressSanitizer and UndefinedBehaviorSanitizer.

5 WHY 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. This section shows that no order can make them so, measures how far real orders get, and so explains why ONTODB adds vector indexes instead of reordering its trees.

Table 3: 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

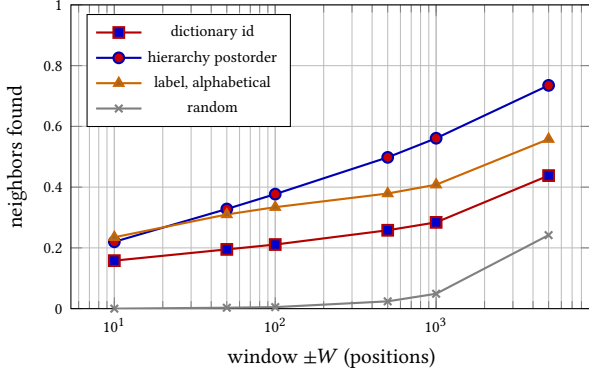


Figure 2: The data of Table 3: the share of neighbors found within $\pm W$ positions, by order.

The order of the trees. A triple is a key of three dictionary ids compared with memcmp. The dictionary assigns ids densely in first-seen order, and the loader interns the terms of each statement in file order, so the order of every tree is the order in which terms first appeared in the input. Two classes are adjacent in the index only if they were adjacent in the file.

LEMMA 5.1 (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.*

PROOF. 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. $\square$

For the neighbor graph of a d -dimensional grid with $N = g^d$ points (diameter $d(g - 1)$), Lemma 5.1 gives $(g^d - 1)/(d(g - 1)) \geq g^{d-1}/d$ positions in any order, so a scan of width w cannot contain all neighbors unless w is that large. Space-filling curves keep neighbors close on average and need several scans to cover a query ball; they do not escape the lemma.

How much locality do real orders give? We took the Gene Ontology (release 2026-07-26 without obsolete terms: 38,092 classes and 57,456 is_a edges), embedded each term’s name with all-MiniLM-L6-v2 [35] ($d = 384$), and for 2,000 random terms took the ten nearest by cosine. For each order of the terms, Table 3 gives the share of those neighbors within $\pm W$ positions.

Three things follow. First, the postorder of the class hierarchy is the best order, and much better than first-seen ids: the nearest neighbor is 45 positions away instead of 1,739. Second, even so, a

Algorithm 4: 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

scan of 100 positions finds 38% of the ten neighbors, and a window of $\pm 1,000$ positions (2.6% of the ontology) still misses 44%; Lemma 5.1 is not an artifact of the grid. Third, first-seen ids are not random: the ontology’s ids were minted in related batches, so ids carry some locality (0.21 at ± 100) that depends on the file and not on the engine. Of the ten neighbors, 12% are a parent or child and 32% are within two is_a steps, while a two-step neighborhood holds only 45 terms (0.12% of the ontology), so hierarchy proximity is highly informative about meaning without being sufficient. The order is therefore useful for structure and not a substitute for a vector index.

6 HIERARCHIES AS INTERVALS

An instance query $inst(c)$ (Definition 2.1) must return the instances of every class in $desc(c)$. A store can *materialize* the entailed type triples, or *rewrite* the query into one scan per class in $desc(c)$. ON-TO-DB instead numbers the classes so that $desc(c)$ 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 $1, \dots, |C|$ in postorder of the forest, and let $lo(c)$ be the smallest number in the forest subtree of c . In a forest, $u \in desc(c)$ 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 4).

LEMMA 6.1 (CORRECTNESS). *$u \in desc(c)$ if and only if $po(u)$ lies in an interval of $I(c)$.*

PROOF. By induction over the order in which Algorithm 4 builds I : the numbers covered by $I(c)$ are exactly those of the classes in $desc(c)$. $\square$

LEMMA 6.2 (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 $desc(c)$.*

PROOF. A class $u \in desc(c)$ 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 e be the last non-tree edge on it; u is then in the forest subtree of the head of e . 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. $\square$

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

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 8 a constant-space predicate on a term's number.

7 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 10). 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 $\mathcal{T}^+$ be $\mathcal{T}$ together with $(e, rdf:type, c')$ for every $(e, rdf:type, c) \in \mathcal{T}$ and every c' with $c \in desc(c')$. 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 5 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, $?x \text{ rdf:type } Student$ looks empty, and a plan that starts from it starts from a pattern that in fact returns 75,547 rows.

Example 7.1. 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}
```

Algorithm 5: 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
```

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 7.2 (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 $\mathcal{T}^+$.*

PROOF. The triple is in $\mathcal{T}^+$ iff e has a stored type c'' with $c'' \in desc(c)$. The label tree holds $(po(c''), e)$ for every stored type of every entity, so by Lemma 6.1 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 $c'' \in desc(c)$. A bound subject is tested with the same condition on its stored types. $\square$

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.

8 VECTOR INDEXES AND FILTERED SEARCH

ONTODB keeps one unit-length float vector $\mathbf{v}(t)$ per term in memory, loaded from a checksummed file whose records name terms by spelling, so a file survives a dictionary that numbers terms differently. Embeddings come from outside; ONTODB stores and searches them. Four index families are available, each a different trade between exactness, memory, and build time.

Flat. Score every vector: $O(Nd)$ and exact. With 8-bit scalar quantization (one scale $\delta = \max_i |v_i|/127$ per vector, for coordinates v_i) each coordinate is off by at most $\delta/2$, so for a unit query a score is off by at most $(\delta/2)\sqrt{d}$, in a quarter of the memory.

IVF. Spherical k -means with k -means++ seeding [4] gives v centroids and each vector goes to its nearest. A query scores the centroids and reads the ρ nearest lists. The cost is $vd + \rho(N/v)d$, minimal at $v = \sqrt{\rho N}$. The design choice is where the lists live: *as key ranges of the B+ tree that holds the triples*. The pair (list, row) is a key (list, row, 0), so a list is a prefix scan, an insert is a tree insert, and

the lists page, cache, and latch like every other index of the engine. Under a filter the search keeps probing the nearest remaining lists until it holds k allowed rows, so a filter that excludes everything near the query costs more instead of returning nothing.

Product quantization (PQ). Cut a vector into m pieces and replace each by the index of its nearest of 256 learned centers [20]: m bytes per vector. A query builds m tables of 256 dot products and scores a code as the sum of m lookups, which equals the dot product with the decoded vector, so all the error is on the stored side.

HNSW. The layered proximity graph of Malkov and Yashunin [24] with the paper’s neighbor-selection heuristic: a candidate is kept only if it is nearer the new node than to every neighbor already kept. Layer 0 allows $2M$ links, and the search beam is ef . Search takes an optional predicate and still crosses nodes it may not return.

8.1 Filtering a Similarity Query by a Class

A filtered similarity query $\text{near}(q, c, k)$ asks for the k terms of $\text{desc}(c)$ nearest to q . Let $f(c) = |\text{desc}(c)|/N$. We compare four ways to answer it, all returning terms of $\text{desc}(c)$ only.

Pre-filter. Resolve c to its members first (one range scan per interval of $I(c)$), then score only those, exactly. About $f(c)Nd$ work.

Post-filter. Ask the index for $k' > k$ terms, keep those below c (a membership test is a binary search over $I(c)$), and multiply k' by four until k remain or the index is exhausted. If the filter were independent of the similarity, about $k/f(c)$ candidates would be needed.

Masked search. Give the index a predicate, “the row’s number lies in an interval of $I(c)$ ”, so a list scan or a graph walk skips rows outside c . Nothing is resolved in advance: each row the search touches costs one binary search. HNSW still crosses rows it may not return, to keep the graph connected, in the spirit of ACORN [34]; IVF keeps probing lists until it has k allowed rows.

Auto. Pre-filter when $|\text{desc}(c)|$ is at most 4,096, otherwise masked HNSW.

The filter is correlated with the vectors. In an ontology the names of a class’s descendants are close to the class and to each other, so the neighbors of a term tend to lie under the term’s own ancestors. On the Gene Ontology, of the unfiltered top 100 neighbors of a query from *inside* c , 4% to 72% lie under c , rising with the size of c ; for a query from *outside* c the share is 0% to 12%. The independence estimate $k/f(c)$ is therefore wrong in both directions: post-filtering is cheap for an inside query and does not terminate usefully for an outside one.

Do not resolve the filter. A masked search that first resolved c by interval scans and built a bitmap per query spent most of its time resolving: for a class of over 5,000 members that was 560 of 669 μ s. Testing the number of each row the search touches, one binary search over a handful of intervals, brought the same search to 105 μ s. This is where interval labels earn their keep a second time: they make membership an $O(\log |I(c)|)$ test with no scan.

9 A BUILD FOR THE BROWSER

The engine compiles with Emscripten to a 0.85 MB WebAssembly module [17] that runs the same indexed store, planner, optimizer, and semantic extension in a web page. Two parts of the native build do not carry over, and replacing them is the whole port.

No raptor. The native loader parses RDF/XML, Turtle, and N-Triples with the raptor C library. The web build uses a Turtle and N-Triples parser of about 500 lines of our own (prefixes, a, ; and , , blank-node property lists, collections, long strings, escapes, language tags, datatypes, and numeric and boolean literals), and a test checks that it yields the same triples as raptor on the shipped data. RDF/XML needs the native build.

No threads. The native engine has a disk-scheduler thread, a deadlock detector, and a worker per session. In the module the scheduler executes each request inline, the detector is not started, and a statement runs on the calling thread; there is one session, so there is nothing to deadlock. The buffer pool, the page file (held in the browser’s memory file system), the B+ trees, and the plans are unchanged, so the page counts of Section 10 and the plans the engine prints are the native ones.

Using it. The module exports three functions: execute a statement, run a query and return JSON, and reset. The playground page built on them offers the memory store and the B+ tree store on a buffer pool, shows the buffer-pool hits and misses of each query, and loads an 834-class slice of the Gene Ontology with its embeddings. In a desktop browser, building the interval labels for the slice 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 building the IVF and HNSW indexes 58 and 98 ms. Updates are saved as a journal in the page’s local storage and replayed when the page loads, which keeps a session across reloads but is not durability: there is no write-ahead log or recovery in the web build.

No GPU. We considered WebGPU for the dot products and did not use it. A similarity query scores a few hundred vectors with HNSW, or at most a few thousand exactly; in the module that is under a millisecond with 128-bit SIMD, which the build enables, and moving the query to a GPU and back costs more than that.

10 EXPERIMENTAL STUDY

We evaluate ONTODB along five research questions.

- **[RQ1] The store:** how does a B+ tree store with three permutations compare with an LSM-based, a Java B+ tree, and a relational store on triple patterns, queries, and loading?
- **[RQ2] Hierarchies:** what do interval labels cost, and how do they compare with materializing and rewriting, and with other engines’ subclass-path queries?
- **[RQ3] Entailment-aware planning:** does the plan rewrite return the answers of materialization, at what cost, and how does it compare with other engines’ subclass-path queries?
- **[RQ4] Vector indexes:** what recall, time, and memory do the four indexes give on real embeddings?
- **[RQ5] Filtered search:** which strategy answers $\text{near}(q, c, k)$ fastest, and does a single rule choose well?

Table 4: 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 (<i>s</i> , ·, ·)	8.4×	3.4×	36.7×	4.2×
pattern (<i>s</i> , <i>p</i> , ·)	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×

- [RQ6] **Concurrency**: how do lookups and inserts scale with the number of threads, and what do readers see while a writer runs?
- [RQ7] **Design choices**: what do three permutations, the bulk loader, and the decode cache each buy?

Table 4 summarizes the answers; the subsections give every measurement.

10.1 Experimental Setup

Systems. We compare ONTODB, built in release mode with the B+ tree store (SPO, POS, OSP), a 256 MiB buffer pool, and automatic join selection, with four configurations of three other systems, all run in-process on one thread of the same machine: Oxigraph 0.5.13 [32] through its Rust interface, with RocksDB on disk and with its in-memory store; Apache Jena TDB2 5.2.0 [3] on Java 17 with a 4 GiB heap and its own bulk loader; and SQLite 3.51 with one table of three string columns, a primary key on (*s*, *p*, *o*) and two more indexes on (*p*, *o*, *s*) and (*o*, *s*, *p*), a SPARQL basic graph pattern translated into a join of the table with itself, and ANALYZE after loading. SQLite stores terms as strings and ONTODB as ids; we include it as the relational baseline. RDF4J, QLever, and RDFS are discussed in Section 11 but not run. Concurrency is measured in-process: threads for ONTODB, Oxigraph, and Jena, and processes with one connection each for SQLite, whose Python driver would otherwise share one interpreter lock.

Data. LUBM [15] at one and ten universities (103,104 and 1,316,700 statements) and the Gene Ontology [5] (release 2026-07-26 without obsolete terms: 38,092 classes, 57,456 *is_a* edges) with *d* = 384 embeddings of the term names by all-MiniLM-L6-v2 [35]. Because the Gene Ontology has no instances, we add 200,000 synthetic entities of one to three types drawn uniformly from its classes (495,552 triples in all).

Workloads and method. The pattern workload samples, from each dataset, 100 constants for every bound position of each shape of Table 2, so that every shape is measured; the LUBM workload is the 14 queries of the benchmark. The instance workload asks for *inst(c)* for 30 classes in each of five size buckets, answered by ONTODB through interval scans and by the others through the subclass path *?c rdfs:subClassOf* c . ?x a ?c* (SQLite: a recursive query). Each query is parsed once, run to warm the caches (three times for the JVM and SQLite), and then five to nine times; we report the median, and the mean of medians over a workload. Every

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

engine converts each answer to its string, and the row counts of all engines agree on every query. Queries that run for a minute are reported as timeouts; none did. Times are warm: they include ONTODB’s decode cache and the other engines’ own caches. The machine is an Apple M2 Pro with 16 GiB. Counts of page reads use a cold 64-frame pool. The code, data generators, and every number in this section are available at the artifact URL.

10.2 RQ1: The Store

Table 5 gives the cost of loading LUBM(10), and Table 6 the query times.

Selective patterns. For patterns that bind the subject, ONTODB takes 3 μ s and 2 μ s and the others 11 to 117 μ s and 9 to 52 μ s: ONTODB is 3.4 to 37 $\times$ faster on (*s*, ·, ·) and 4.6 to 27 $\times$ on (*s*, *p*, ·), and 1.4 to 58 $\times$ faster on the selective LUBM queries. Every answer is one descent of a B+ tree that fits the pool and a few keys.

Large results. For patterns that return 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, equal to Oxigraph’s in-memory store on (·, ·, *o*) and 1.3 to 1.7 $\times$ slower on the other two, and 1.3 to 2.0 $\times$ slower than SQLite, whose string index returns spellings without a dictionary.

LUBM queries. Ten of the fourteen LUBM queries return no rows without inference; every engine answers them in microseconds and the ratios of two tiny times mean nothing, so Table 6 lists the four that return rows. On the triangle query SQLite takes 4.0 ms, ONTODB 8.2 ms, Oxigraph in memory 34 ms, Jena 120 ms, and Oxigraph on RocksDB 275 ms: ONTODB is 33 $\times$ faster than the last, and twice as slow as SQLite; we did not inspect the other engines’ plans. On the large scan (LUBM 14) the order is SQLite, Oxigraph in memory, ONTODB, Oxigraph on RocksDB, Jena.

Loading. ONTODB is the slowest to load, 11.4 s against 2.0 s for Oxigraph on RocksDB (5.6 $\times$), 6.1 s for Jena, and 8.1 s for SQLite, and its peak memory is 584 MiB against 78 MiB for Oxigraph on RocksDB, 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. In sum, a B+ tree store of three permutations is competitive with the other designs on basic graph patterns: fastest on selective ones, within a factor of two of the best on large ones, and slowest to load.

Table 6: 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 7: 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

Table 8: Descendants of a class in GO, cold 64-frame pool, mean over up to 40 classes per row.

Below c	method	scans	page reads	μ s
10–99	intervals	5.9	5.8	59
	one scan per class	30.6	16.3	149
100–999	intervals	29.1	13.2	147
	one scan per class	268.2	65.7	803
1000+	intervals	101.3	40.1	528
	one scan per class	3303.6	694.1	8998

10.3 RQ2: Hierarchies

Interval labels. Table 7 shows that the Gene Ontology needs 76,570 intervals for 38,092 classes, 2.01 per class and 598 KiB in all, where the closure it replaces has 393,448 ancestor pairs (9 MiB as 24-byte keys in one permutation). The choice of spanning forest matters little: taking the deepest parent saves 13%. The tail is long, one class needing 741 intervals, within the bound of Lemma 6.2. Labels build in 15 ms, and asking whether a class is below another takes 27 ns against 775 ns for a search up the parents.

Against materializing and rewriting. Table 8 compares interval scans with a search that scans POS once per class. For a class with a thousand or more descendants the intervals need 101 scans instead of 3,304, 40 page reads instead of 694, and 0.5 ms instead of 9 ms. Table 9 compares the three strategies for instance queries on the synthetic instances. 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. On queries the interval index is close to the materialized store and far from rewriting: for a class with a thousand or more descendants it reads 265 pages against 182 (materialized) and 12,803 (rewrite). All three return identical answers on every sampled class. On LUBM(1) the hierarchy is shallow (43 classes) and the three are within a factor of about two on every query, and materialization adds 15% to the triples: the benefit of intervals grows with the depth and breadth of the hierarchy.

Table 9: Instances of c on 200,000 synthetic entities over GO, mean over 30 classes per row (cold 64-frame pool). Top: cost of the store.

Strategy	triples	file (MiB)	load (s)
rewrite	495,552	63.6	5.09
intervals	495,552	72.9	5.26
materialize	4,017,973	309.5	180.31

Below c	strategy	answers	page reads	μ s
10–99	intervals	278	8.4	90
	rewrite	278	87.5	571
	materialized	278	6.6	444
100–999	intervals	2,954	40.9	1,461
	rewrite	2,954	1,153	7,280
	materialized	2,954	22.5	987
1000+	intervals	29,904	264.8	4,555
	rewrite	29,904	12,803	82,441
	materialized	29,904	182.0	3,776

Against other engines. Table 10 compares ONTODB with the other engines on instance queries. The interval scan alone returns term ids, in 1.9 ms for the 29,904 answers of the largest classes (about 60 ns each); returning strings, as the others do, takes 11.5 ms because each id is decoded. At the string level ONTODB is 2.2 $\times$ faster than the best other engine (Oxigraph in memory) on the largest classes, 3.5 $\times$ faster than SQLite’s recursive query, and 5.4 to 5.8 $\times$ faster than Jena and Oxigraph on RocksDB, and 5 to 9 $\times$ faster than Oxigraph in memory on classes with fewer than 1,000 instances. With the decode cache off (Table 17) ONTODB takes 22.6 ms for the largest classes, still 1.1 $\times$ faster than the best other engine. A subclass path is evaluated by the others as a graph traversal followed by a join, and by ONTODB as range scans of one tree that return disjoint answers.

10.4 RQ3: Entailment-Aware Planning

We add the LUBM ontology to LUBM(10) and run the seven LUBM queries that need only subclass reasoning (3, 4, 6, 7, 8, 9, 10: their classes *Student*, *Professor*, *Faculty*, *Course*, and *Publication* have sub-classes). Table 11 gives the times and Table 12 the cost of preparing.

Same answers. Without the rewrite ONTODB returns no rows for six of the seven queries. With it, every query returns exactly

Table 10: 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 11: 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 12: 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

the rows of the materialized store and of Oxigraph’s and Jena’s subclass-path queries, as the row counts in the table show; a randomized test over ontologies with cycles and over joins of several type patterns, on both backends and across updates, compares the rewrite with a materialized oracle.

Against materialization. Query times are within 20% of those of the materialized store, from 10% faster to 20% slower. Preparing is 16 times faster (0.06 s against 0.97 s), and nothing is added to the store: materialization adds 184,628 triples (14%) and 40.7 MiB (30%) to the file, where the interval index takes 5.1 MiB in a file of its own.

Against other engines. Oxigraph on RocksDB, evaluating `?c rdfs:subClassOf* C . ?x a ?c`, is 1.9 to 6.9 times slower than the rewrite, and Jena TDB2 2 to 9,300 times slower, with LUBM 9 timing out after a minute. Oxigraph’s in-memory store is faster than the rewrite on five of the seven queries, by 1.1 to 2.7 times, and slower on two; the rewrite is the fastest of the disk-based engines on every query.

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.

10.5 RQ4: Vector Indexes

We ran 1,000 query terms (the term itself excluded from its answer) against all 38,092 vectors, with exact search as ground truth. Table 13 gives selected points and Figure 3 the recall-latency curves.

A baseline that is not slow by accident. Our first run scored with a plain dot-product loop. A compiler may not vectorize it without `-ffast-math`, and the exact scan took 8.3 ms, which would have made the speed-ups look about twice as large: HNSW at $ef=10$ is 103 times faster than that baseline and 51 times faster than the vectorized one. With eight independent accumulators the exact scan takes 2.2 ms (13 Gflop/s), and every number here uses that. A speed-up over “exact” depends entirely on how fast exact is.

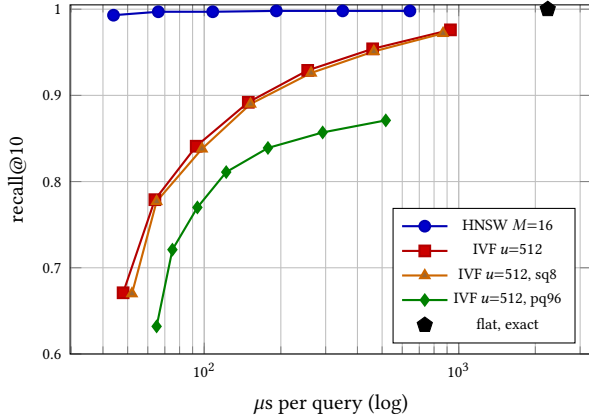


Figure 3: Recall@10 against time per query on GO (38,092 vectors, 384 dimensions).

What the numbers say. HNSW dominates on this data: recall 0.993 at 44 μ s, 51 $\times$ faster than the exact scan, for 3.9 MiB of links on top of the vectors. IVF needs 255 μ s for 0.929 and 928 μ s for 0.976. IVF over the B+ tree is about 2.7 $\times$ more expensive per scored row than a contiguous scan (157 ns against 59 ns): each entry is a decoded 24-byte key and a random read of the vector. That is the price of lists that page, cache, and update with the triples. The cost model holds in shape: more lists make a query cheaper until the centroid scan dominates ($v=2048$ pays 124 μ s at $p=1$). Product quantization is for memory, not speed, at this size: codes alone reach 0.78 or 0.88 recall in 1.7 or 3.5 MiB, and re-ranking the best 100 exactly recovers 0.998 and 1.000 but needs the float vectors. 8-bit codes keep 0.996 recall in a quarter of the memory and are slower than float32, because converting bytes to floats costs more than the cache misses they avoid at 56 MiB. The queries are terms of the ontology, whose neighbors are also terms of it; queries from outside the vocabulary, such as free text, would be harder, and we do not measure them.

10.6 RQ5: Filtered Search

For 20 classes in each of five size buckets we ran the ten nearest terms under c for a query inside c and one outside it, against the exact answer over the members. IVF uses 512 lists and 16 probes, HNSW $M=16$ and $ef=64$. Table 14 gives recall and time.

Exact pre-filtering wins for small classes. Up to about 1,000 members, resolving the members through the interval index and scoring them exactly takes 3 to 62 μ s and is the exact answer. Every index-based strategy is slower, from 1.7 $\times$ (masked IVF at 101 to 1,000 members, at recall 0.96) to over 10,000 $\times$ (post-filtered HNSW on the smallest classes), because few of the vectors an index touches belong to the class. A graph search for a rare filter crosses most of the graph: masked HNSW reads 16 ms for a class of at most 100 members, the failure ACORN is designed to remove.

Post-filtering fails for outside queries. IVF post-filter recall falls to 0.07 on the smallest classes and 0.64 on the largest; HNSW reaches full recall only by searching again and again (37 ms for a class of at most ten members, in six rounds).

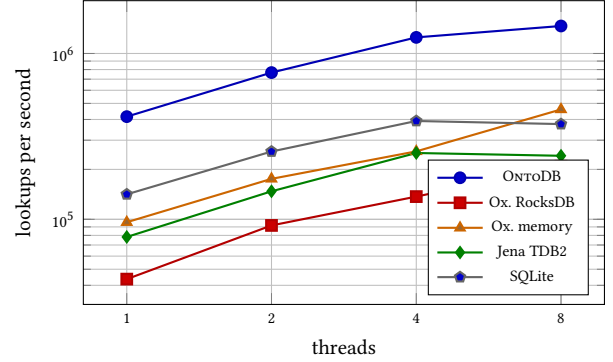


Figure 4: Lookups per second on LUBM(10) against the number of threads (both axes logarithmic).

Masked HNSW wins for large classes. Above 5,000 members it takes 105 μ s inside and 900 μ s outside, exact, against 1.4 ms for resolving and scoring the members.

A cost model for the crossover. Let γ_m be the cost of pre-filtering per member (an interval scan, a hash lookup from term to row, and one dot product); from Table 14, $\gamma_m \approx 0.13$ to 0.20 μ s for classes of a few hundred members or more. Pre-filtering beats a masked graph search when $|\text{desc}(c)| < \theta = \gamma_h / \gamma_m$, where γ_h is the masked search’s time. For a query inside c , $\gamma_h \approx 105$ to 183 μ s, so $\theta \approx 800$ to 1,000, which matches the observed crossing between the 101–1,000 and 1,001–5,000 columns. For a query outside c the masked walk costs 0.9 to 4.4 ms on classes of 1,000 or more members, so θ is 7,000 to 25,000. A single threshold has to sit between the two; 4,096 stays below the outside-query crossing and loses at most a factor of 3.8 at the inside-query one.

The auto rule. 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 exception is a class of 1,001 to 5,000 members with an inside query: 269 μ s, against 70 μ s for post-filtered HNSW and 183 μ s for masked HNSW. Post-filtering would cost 5.5 ms for an outside query on the same class, so the rule gives up a factor of 3.8 in one cell to avoid a factor of 79 in its neighbor.

10.7 RQ6: Concurrency

We load LUBM(10) and run, for three seconds each, one to eight threads of lookups (the $(s, \cdot, \cdot)$ and $(s, p, \cdot)$ queries, 200 in rotation), one to eight threads of inserts of new triples with scattered subjects, and one writer beside one to eight readers. Figure 4 plots the lookups and Table 15 the rest.

Readers. ONTODB serves 416 thousand lookups per second with one thread and 1.46 million with eight, 3.5 times more, and its single thread already beats every other engine at eight threads except Oxigraph’s in-memory store, which reaches 460 thousand. At eight threads ONTODB serves 7.0, 3.2, 6.1, and 3.9 times the lookups of Oxigraph on RocksDB, Oxigraph in memory, Jena TDB2, and SQLite. Every engine scales with threads here; ONTODB’s lead is its per-lookup cost, 2.4 microseconds, kept as threads are added.

Readers beside a writer. With a writer inserting continuously, ONTODB’s eight readers still serve 1.39 million lookups per second,

Table 14: Ten nearest terms under a class c , by subtree size, query inside or outside c : recall@10 / μ s per query. GO, 38,092 terms. Masked and post-filtered searches use the index; pre-filter resolves the members and scores them exactly.

Strategy	≤ 10	11–100	101–1000	1001–5000	> 5000
<i>query inside c</i>					
pre-filter	1.00 / 3	1.00 / 9	1.00 / 62	1.00 / 292	1.00 / 1,404
post-filter, HNSW	1.00 / 37,553	1.00 / 1,739	1.00 / 427	1.00 / 70	1.00 / 68
post-filter, IVF	0.86 / 903	0.87 / 558	0.93 / 293	0.98 / 198	0.99 / 211
masked HNSW	1.00 / 16,653	1.00 / 15,846	1.00 / 1,150	1.00 / 183	1.00 / 105
masked IVF	1.00 / 1,253	0.95 / 111	0.96 / 104	0.98 / 124	0.99 / 161
auto	1.00 / 3	1.00 / 9	1.00 / 59	1.00 / 269	1.00 / 95
<i>query outside c</i>					
pre-filter	1.00 / 3	1.00 / 9	1.00 / 62	1.00 / 291	1.00 / 1,394
post-filter, HNSW	1.00 / 36,654	1.00 / 27,669	0.99 / 13,607	1.00 / 5,513	1.00 / 836
post-filter, IVF	0.07 / 966	0.02 / 1,009	0.17 / 902	0.33 / 811	0.64 / 591
masked HNSW	1.00 / 16,802	1.00 / 16,611	1.00 / 8,697	1.00 / 4,424	1.00 / 900
masked IVF	1.00 / 1,255	0.79 / 698	0.45 / 286	0.37 / 147	0.66 / 107
auto	1.00 / 3	1.00 / 9	1.00 / 62	1.00 / 295	1.00 / 860

Table 15: 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

3.5 times its one-reader rate, while the writer keeps inserting 111 thousand triples per second. Beside a writer, eight readers reach 74 thousand lookups per second on Jena TDB2, 136 thousand on Oxigraph on RocksDB, 288 thousand on Oxigraph in memory, and 322 thousand on SQLite, so ONTODB’s eight readers do 18.8, 10.2, 4.8, and 4.3 times as many.

Writers. ONTODB’s insert rate grows from 264 thousand per second with one thread to 950 thousand with eight, 3.6 times; Oxigraph on RocksDB gains 16%, Oxigraph in memory loses nearly three quarters, SQLite loses 4%, and Jena TDB2’s rate is bounded by its commit. The comparison favors ONTODB in one respect: its inserts here go through the store without the write-ahead log, while the other engines commit each insert with their default durability. The scaling of ONTODB’s own curve is the result; Oxigraph’s in-memory store, which is not durable either, inserts faster with one thread (679 thousand per second).

What limits it. Heavy queries scale less: the 16,000-row $(\cdot, p, o)$ query goes from 148 to 330 queries per second with eight threads, because it copies leaf after leaf and decodes thousands of terms,

Table 16: 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

so it is bound by memory traffic. Splits and merges take the pessimistic path and serialize with each other on the pages they share, and inserts into one leaf serialize on its latch.

10.8 RQ7: Design Choices

Permutations. One tree, SPO, answers a pattern that does not bind the subject by a scan of the whole tree. Table 16 counts, on LUBM(1) with a cold 64-frame pool, the page reads of a pattern for one, three, and all six permutations (averages over 20 sampled patterns per shape). Three permutations cut the pages of $(\cdot, p, \cdot)$ by 7.9 $\times$ and those of $(\cdot, \cdot, o)$ and $(\cdot, p, o)$ by 40 $\times$; the other three trees cost 66% more space and change no pattern, because every shape already has a prefix scan in the first three.

Bulk loading. Loading the 15 files of LUBM(1) one triple at a time takes 1.74 s and 16.2 MiB for three permutations; the bulk loader takes 1.08 s and 10.8 MiB, a third less space, and a full scan reads 599 pages instead of 1,131 because inserted leaves are about half full and bulk-loaded leaves are full. With the loader, the time that remains is the dictionary’s.

Decode cache. Table 17 shows what the dictionary’s decode cache buys: the subject-bound pattern gets 4.3 $\times$ faster, patterns that return thousands of rows get 1.5 to 2.3 $\times$ faster, and the largest instance queries 2.0 $\times$. The cache does nothing for ids, which is why the interval scan alone is 6 $\times$ faster than the same query returning strings. A cache is the remedy Jena uses for the same cost; a cheaper id-to-spelling access path would help every cold query and is future work.

Table 17: 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 18: 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 [32]	RocksDB (LSM)	query only	in-memory
Jena TDB2 [3]	B+ trees	reasoner layer	no
RDF4J native [9]	B-trees	materialized	no
RDF-3X [28]	6 compressed B+ trees	query only	no
RDFox [26]	in memory	materialized	no
ONTODB	3 B+ trees	intervals	full engine

11 RELATED WORK

RDF stores. Triple stores keep triples in sorted permutations. Hexastore indexes all six [36]; RDF-3X keeps six compressed B+ trees and prunes joins with statistics [28]; QLever builds read-mostly permutation indexes in bulk [6]; HDT is a compressed read-only format [11]; Virtuoso stores quads in a relational engine [10]. Among systems we could run side by side, Apache Jena TDB2 uses B+ trees [3], RDF4J’s native store uses B-trees [9], and Oxigraph [32] keeps its indexes in RocksDB, a log-structured merge tree. ONTODB follows the B+ tree line with three permutations, which Section 10 shows suffice. The B+ tree [7, 8] and its concurrency and recovery machinery [13, 22, 25] are textbook; the contribution here is the choice of parts, their measured behavior on ontologies, and what is built on top (Table 18).

Inference. RDFS entailment is usually either *rewritten* into the query or *materialized* at write time, with maintenance by counting or by delete-and-rederive [16]. Section 6 adds a third option that stores no entailed triple. Agrawal, Borgida, and Jagadish [1] store a compressed transitive closure by numbering a spanning forest in postorder and giving each node a set of intervals; Grust [14] numbers XML trees in pre- and postorder so that axes become regions of a plane. We use the former on class DAGs inside a B+ tree store and measure it against the other two ways to answer a type query.

Vector search. Approximate nearest neighbor search has three families that matter here: inverted files and product quantization [20], proximity graphs, of which HNSW [24] is the standard, and disk-oriented graphs such as DiskANN [19]. FAISS [21] packages the first two. Pan, Wang, and Li survey vector database systems and the hybrid queries (attribute plus vector) that are hard for them [33]. ACORN [34] makes HNSW traverse only the

nodes that pass a predicate. Neo4j offers an HNSW vector index queried from Cypher [27]; Amazon Neptune Analytics offers vector similarity over graph nodes [2]; Fluree provides inline similarity functions and HNSW indexes for RDF [12]; GraphDB has had similarity indexes queried from SPARQL since version 8 [31]. In the documentation we read, none turns the class hierarchy into range-scannable intervals, in the engine that holds the triples, to restrict a vector search. Euclidean embeddings of names are what we use; hyperbolic embeddings [29] represent hierarchies more faithfully and would change the vectors, not the indexes.

Adaptive radix trees. An ART [23] orders keys by byte, so its range scan is a lexical range scan, the same one-dimensional order as a B+ tree, and Lemma 5.1 applies to it. Giving its leaves sibling links is possible and awkward: path compression and node growth never move a leaf, but lazy expansion does, so a link that pointed at the old leaf points at an inner node, and concurrent inserts must latch both neighbors. The usual alternative is a stack-based iterator that climbs and descends between leaves. ONTODB’s keys are three fixed-width integers, which leave an ART little to compress, so the B+ tree stays; DuckDB persists an ART for its own keys [18].

Databases in the browser. WebAssembly [17] lets a C++ engine run in a page. Oxigraph ships a JavaScript build with an in-memory store. ONTODB compiles the whole engine, including the paged B+ trees, the buffer pool, and the semantic extension, by replacing the two parts that need threads or a native library (Section 9).

12 CONCLUSION

An ontology store is asked for patterns, for hierarchies, and for meaning. This paper described ONTODB, which answers all three inside one disk-based engine. A B+ tree store of three permutations, with crabbing insert and delete, a range scan that holds no latch between rows, and a bulk loader that fills pages, answers selective patterns faster than an LSM store, a Java B+ tree store, and a relational store, and is within a factor of two of the best on large results. Labelling the class DAG with interval sets turns “everything under *c*” into a few range scans of the same tree, which a plan rewrite uses to answer RDFS entailment of type patterns with the answers of materialization and none of its storage, at 2.0 intervals per class on the Gene Ontology and a fraction of the cost of materializing the closure, and makes membership in a class’s subtree a binary search, which a filtered vector search needs. A sorted index orders and does not say what is close in meaning, which the bandwidth argument and the measurements agree on; exact pre-filtering through the intervals is the right answer for classes up to about a thousand members and a masked graph search for the largest. The whole engine runs in a browser from a 0.85 MB module.

Limitations. ONTODB implements a SPARQL subset without OPTIONAL, UNION, property paths, or aggregates, and a reasoner only for `rdfs:subClassOf`; a user who needs the full language needs one of the other systems we measured. It is the slowest to load, single-threaded, and hungry for memory while loading, because its optimizer’s exact statistics sit in hash tables. Decoding an id costs about 0.7 μ s and dominates large results, which the decode cache hides on a warm store and does not remove. Labels are rebuilt,

not maintained, when the hierarchy changes. The vectors come from one ontology and one small model that embeds names only, the instances of the Gene Ontology are synthetic, vectors are held in memory, and the semantic measurements are single-threaded. Queries are terms of the ontology; free text would need an embedding model that ONTODB does not link. We did not find a vector index that makes the hierarchy order unnecessary: neighbors by meaning and descendants are different sets (a third of a term's ten nearest neighbors are within two `is_a` steps, two thirds are not), which is why the two are combined at query time and not at index time.

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] David Arthur and Sergei Vassilvitskii. 2007. k-means++: The Advantages of Careful Seeding. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1027–1035.
- [5] Michael Ashburner et al. 2000. Gene Ontology: Tool for the Unification of Biology. *Nature Genetics* 25 (2000), 25–29.
- [6] 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.
- [7] Rudolf Bayer and Edward M. McCreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (1972), 173–189.
- [8] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137.
- [9] Eclipse RDF4J. [n.d.]. Eclipse RDF4J: a Java framework for RDF. <https://rdf4j.org/>. Accessed October 2026.
- [10] Orri Erling and Ivan Mikhailov. 2010. Virtuoso: RDF Support in a Native RDBMS. In *Semantic Web Information Management*. Springer, 501–519.
- [11] 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.
- [12] Fluree PBC. 2026. Fluree Vector Search. Fluree documentation. <https://labs.fluree.com/docs/db/indexing-and-search/vector-search>, accessed October 2026.
- [13] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.
- [14] Torsten Grust. 2002. Accelerating XPath Location Steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 109–120.
- [15] Yuanbo Guo, Zhengxiang Pan, and Jeff Hefflin. 2005. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics* 3, 2–3 (2005), 158–182.
- [16] 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*.
- [17] 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.
- [18] Pedro Holanda. 2022. Persistent Storage of Adaptive Radix Trees (ART) in DuckDB. DuckDB blog. <https://duckdb.org/2022/07/27/art-storage>.
- [19] 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)*.
- [20] 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.
- [21] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] 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.
- [26] 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*.
- [27] Neo4j, Inc. 2026. Neo4j Vector Index and Search. Neo4j documentation. <https://neo4j.com/developer/genai-ecosystem/vector-search/>, accessed October 2026.
- [28] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal* 19, 1 (2010), 91–113.
- [29] Maximilian Nickel and Douwe Kiela. 2017. Poincaré Embeddings for Learning Hierarchical Representations. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [30] 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.
- [31] 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>.
- [32] Oxigraph contributors. [n.d.]. Oxigraph: a SPARQL graph database written in Rust. <https://github.com/oxigraph/oxigraph>. Version 0.5; accessed October 2026.
- [33] James Jie Pan, Jianguo Wang, and Guoliang Li. 2024. Survey of Vector Database Management Systems. *The VLDB Journal* 33 (2024), 1591–1615.
- [34] 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.
- [35] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings Using Siamese BERT-Networks. In *Proceedings of EMNLP-IJCNLP*.
- [36] 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.

Algorithm 9: 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;
```

Algorithm 10: 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  $\varphi$ ;
```

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

```
1 for each permutation  $\pi \in \{\text{SPO, POS, OSP}\}$  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 7: BEGINRANGE(a, b), TAKE, and SETTLE

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

A.3 Bulk load

Algorithm 8: 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, ...;  $\Lambda \leftarrow$  the (first key, id) of the
     new pages;
6 set the root in the header to the page of  $\Lambda[0]$ ;
```

A.4 The concurrent algorithms