

OntoDB

Database Internals by Example

Storage, indexing, query processing, concurrency, and recovery,
taught through a database management system for ontologies

Lam Nguyen

Contents

Design Decisions	vii
Preface	x
I Foundations	1
1 What a Database Engine Is	2
1.1 The layers	2
1.2 RDF in one page	3
1.3 SPARQL in one page	3
1.4 The running example	4
II Storage	6
2 Disks, Files, and Pages	7
2.1 The storage hierarchy	7
2.2 Pages	7
2.3 Slotted pages	8
2.4 Talking to the file system	9
3 The Buffer Pool	13
3.1 Frames, the page table, and pins	13
3.2 Choosing a victim	14
3.3 Steal and force	14
4 B-Trees and B+ Trees	20
4.1 Why a tree of pages	20
4.2 B-tree versus B+ tree	20
4.3 Search	21
4.4 Insertion and splits	21
4.5 Deletion	21
4.6 Bulk loading	22

4.7	Iterators and pins	22
5	Indexing RDF	27
5.1	Dictionary encoding	27
5.2	A key whose bytes sort correctly	28
5.3	Prefix scans	28
5.4	One ordering is not enough	28
6	The Log-Structured Alternative	34
6.1	How an LSM tree works	34
6.2	Three amplifications	34
III	Query Processing	36
7	From SPARQL to a Plan	37
7.1	Parsing and binding	37
7.2	Logical and physical plans	37
7.3	The iterator (Volcano) model	38
7.4	The operators	38
7.5	Sorting more than fits in memory	39
8	Joins and Query Optimization	43
8.1	Three join algorithms	43
8.2	Cardinality estimation	43
8.3	Searching for a join order	44
8.4	Explain analyze	44
9	Beyond Basic Graph Patterns	50
9.1	Where a conjunction loses rows	50
9.2	OPTIONAL is a left outer join	50
9.2.1	A filter inside OPTIONAL is part of the join	51
9.3	Executing a left join	51
9.4	UNION	52
9.5	Aggregation	52
9.6	Planning around the new operators	52
9.7	Updates whose triples come from a query	53

10	Materialized Inference	55
10.1	Two ways to answer with entailments	55
10.2	The rules	56
10.3	Naive and semi-naive evaluation	56
10.4	Keeping the closure current	57
10.4.1	Insert	57
10.4.2	Delete is harder	57
10.5	What it costs	58
11	Semantic Search	60
11.1	What the tree orders	60
11.2	A sort order cannot hold similarity	60
11.3	The exception: a hierarchy is a set of intervals	61
11.3.1	Three ways to answer a type query	61
11.4	Vector indexes	62
11.5	Filtering a vector search by a class	63
11.6	Why not an ART with sibling links	64
11.7	What it costs	64
IV	Transactions	68
12	Transactions and Isolation	69
12.1	ACID	69
12.2	Schedules and serializability	69
12.3	Anomalies	69
12.4	Isolation levels	70
13	Concurrency Control	73
13.1	Two-phase locking	73
13.2	Locking at several granularities	73
13.3	Deadlock	74
13.4	Phantoms	74
13.5	Latches are not locks	74
13.6	The alternative: multiversion concurrency	75

V Recovery	78
14 Logging and the Write-Ahead Rule	79
14.1 The failure model	79
14.2 Undo and redo	79
14.3 The write-ahead rule	80
14.4 Making commits cheap: group commit	80
14.5 When a record is damaged	80
14.6 What a page record holds	82
14.7 Changes that span pages	82
15 ARIES	85
15.1 The data structures	85
15.2 Checkpoints	85
15.3 Pass 1: Analysis	85
15.4 Pass 2: Redo	86
15.5 Pass 3: Undo	86
15.6 A worked example	86
15.7 Structure changes: nested top actions	87
15.8 Undo must survive being repeated	87
15.9 The crash harness	88
16 Durability in Practice	91
16.1 Where it sits in the steal/force table	91
16.2 Logical logging	91
16.3 One statement, one transaction	92
16.4 Recovery	92
16.5 Checkpoints as atomic snapshots	92
16.6 Damaged logs: recovery modes	93
16.7 Verification	93
16.8 Testing a recovery design	93
17 From Memory to Pages	96
17.1 What is in a directory	96
17.2 Opening is recovery	97
17.3 Statements log on pages	97

17.4 Loading: one record per triple, or one per load	97
17.5 Closing cleanly, and checkpoints	98
17.6 Checking a page database	98
17.7 Testing through the shell	99
18 Keeping the Log Small	100
18.1 What recovery still needs	100
18.2 Cutting the log	100
18.3 Torn pages and smaller page records	101
18.4 Fuzzy checkpoints	102
A A Map from Chapters to OntoDB	104
B Answers	105
Bibliography	115

Design Decisions

Every choice OntoDB makes is written up where the theory behind it is taught, in a box that names the choice, the alternatives it passed over, why, and what it costs. A title marked *planned* is a decision fixed in the plan for code not yet written.

1	One text interface	4
2	Store OWL as triples, with no reasoner	4
3	4 KiB pages	9
4	Page 0 is a real page	9
5	The file's length records allocation	10
6	The file only grows	10
7	Through the page cache; fsync at shutdown	10
8	Compact on delete	10
9	A term must fit on one page	11
10	Append to the last page	11
11	Spellings are never deleted	11
12	One I/O thread, first in, first out	15
13	LRU-K with $K = 2$	15
14	One latch for the whole pool, held across I/O	16
15	The pool's destructor does not flush	16
16	A full pool returns null	16
17	Pages only through guards	16
18	Pin, then latch; unlatch, then unpin	17
19	A latch belongs to a page, not to a frame	17
20	A write guard dirties only if written	17
21	Inspecting a page goes through the pool	18
22	Fixed slots, no key compression	22
23	The key is the record	23
24	Strict deletion	23
25	The invariant checker comes before insert	23
26	Hold the path, not parent pointers	24
27	Split in the middle	24
28	Bulk load packs leaves full	24

29	A scan holds nothing between rows	24
30	A load arrives as one batch	25
31	Small nodes in tests	25
32	Ids start at 1; 0 and the maximum are reserved	29
33	The dictionary is built from the engine's own pages	29
34	A fixed hash function	30
35	Big-endian keys, compared with memcmp	30
36	The upper bound saturates instead of failing	30
37	The catalog points at header pages, not roots	31
38	The catalog claims page 0 by writing it	31
39	SPO, POS, and OSP, measured	31
40	Permutations are data, and every write is all-or-nothing	31
41	The shell's indexed backend lives in a temporary file	32
42	Only inserts add terms to the dictionary	39
43	Unsupported syntax fails at parse time	39
44	Volcano, one row at a time	40
45	Stub plan nodes before their executors	40
46	ORDER BY uses a total order on values	40
47	External sort through the buffer pool	41
48	DISTINCT keeps a hash set	41
49	Rows have no order without ORDER BY	41
50	Exact counts for single patterns	45
51	Join estimates use distinct values	45
52	The inner side is re-opened with the outer row	45
53	Index nested loop probes the store directly	46
54	Hash join builds on the inner side	46
55	Forcing a join method rewrites the plan	46
56	Greedy, left-deep join ordering first	46
57	EXPLAIN ANALYZE wraps every executor	47
58	auto picks the operator per join, and is the default	47
59	Left joins build on the right (planned)	51
60	Hash aggregation, and what to do when groups outgrow memory (planned) . .	52
61	Evaluate WHERE first, then delete, then insert (planned)	53
62	Materialize, and maintain with DRed (planned)	57

63	Where inferred triples live (planned)	58
64	Ids stay first-seen; labels are a separate index	62
65	IVF lists are B+ tree ranges	63
66	Intervals instead of a materialized closure for type queries	65
67	Resolve small classes, mask large ones	66
68	Every statement is a transaction	70
69	A failed statement takes itself back	71
70	Read committed by default	71
71	Shared locks are kept to commit, even at read committed	71
72	Isolation is specified as executable schedules	72
73	Locking, not multiversioning	75
74	Detect deadlocks and abort the youngest	75
75	Upgrades wait their turn	76
76	Phantoms: a lock on the set at repeatable read, on everything at serializable	76
77	Locks name triples, not index entries	76
78	Pessimistic crabbing, and emptied leaves stay	76
79	Length and CRC-32C in every record	81
80	LSNs are counters, not byte offsets	81
81	A failed fsync is fatal	82
82	Whole page images, not physiological records	83
83	Page groups hold back the log, not the latches	83
84	Physical redo, logical undo	88
85	A sharp checkpoint	89
86	Recovery is tested against an oracle	89
87	Terms replay whatever their transaction did	94
88	Point-in-time recovery by default	94
89	A format file names the store (planned)	96
90	How a load is logged (planned, to be measured)	98
91	Truncate by copying the tail (planned)	101
92	Full image after a checkpoint, then diffs (planned)	101

Preface

A database engine is a small number of ideas, applied with care. Data lives in fixed-size pages because disks move blocks. A buffer pool keeps the hot pages in memory because memory is fast and small. A B+ tree turns a key into a page in three or four reads because its fanout is in the hundreds. A lock manager makes concurrent transactions look sequential. A write-ahead log, and an algorithm called ARIES, make a committed transaction survive a crash that happens at any instant.

This book explains those ideas from the ground up, and it explains each one twice: first as theory, then as it appears in a real system. The real system is **OntoDB**, a database management system for ontologies. Its data is RDF, a graph of subject–predicate–object triples, and its query language is SPARQL. Every chapter ends its theory with an *In OntoDB* box that shows what OntoDB does, and design-decision boxes that show what it chose instead, and why.

Why an ontology database

A relational engine teaches the same ideas, so why a triple store? Two reasons. First, a triple store strips the problem to its core: there is one table, with three columns, and every interesting question is a join. That makes indexing and join ordering impossible to wave away. Second, ontologies are where correctness matters most and is checked least: a scientific ontology edited by many hands needs transactions, a log, and recovery just as much as a bank ledger does.

How to read this book

The parts follow the path of a request through the engine, from the bottom up.

Foundations What an engine is made of, and enough RDF and SPARQL to follow the examples.

Storage Pages, the buffer pool, B+ trees, how RDF is indexed, and the log-structured alternative.

Query processing How a SPARQL query becomes a tree of operators, and how joins are chosen.

Transactions What isolation means, and how locking provides it.

Recovery Logging, the write-ahead rule, ARIES, and how OntoDB makes its databases durable today.

The boxes

Four kinds of box recur. A *Key idea* is the one thing to keep from a section. *In OntoDB* shows how the running example applies the theory. A *Design decision* names a choice OntoDB made, the alternatives, why it chose as it did, and what the choice costs; a list of them follows the contents,

and they are the best place to start if you want to know why the engine is built the way it is. *Status* says what runs today. Each chapter ends with questions; the answers are in appendix B.

Code appears only where a picture or a sentence would be less precise: a key layout, a log record, a few lines of pseudocode for an algorithm. The aim is to understand *why*, not to read an implementation.

Status

OntoDB is built feature by feature, and parts of this book describe designs that are specified but not yet implemented. Each chapter's *Status* box says which. At the time of writing these run: the SPARQL shell; the triple key; the buffer pool (disk manager, I/O scheduler, LRU-K, page guards, and inspection); the B+ tree; RDF storage on pages (term heap, catalog, disk dictionary, and the indexed store, which the shell switches to with `\set backend indexed`); the log records and the log manager; every query operator and the optimizer (statistics, join order, the choice of join, and `\explain analyze`); and durable databases (snapshots, a checksummed write-ahead log, recovery, and verification); bulk loading; and concurrency control (latch crabbing, transactions, locks, isolation, deadlocks, and phantoms); and page-level ARIES (page images in the log, page groups for splits and merges, abort by the log, a sharp checkpoint, the three passes, and a crash harness that kills a real store). The shell's `ontodb <dir>` still opens the memory database of chapter 16; the page store recovers in the tests. Four chapters describe the next features, all planned: the SPARQL forms beyond basic graph patterns (chapter 9), materialized RDFS inference (chapter 10), the page store in the shell (chapter 17), and log housekeeping (chapter 18).

Acknowledgments

The structure of this book owes much to Alex Petrov's *Database Internals*, whose division into storage engines and distributed systems it follows for the first half. The recovery chapters follow Mohan et al.'s ARIES paper closely, because nothing explains ARIES better than ARIES.

Part I

Foundations

What a Database Engine Is

A database management system (DBMS) accepts declarative requests, such as “find every student and their name”, and answers them correctly and quickly, while many clients read and write at once and while machines fail. Each of those words is a subsystem. *Correctly* under concurrency is the job of the transaction manager. *Quickly* is the job of indexes and the query optimizer. *While machines fail* is the job of the log and recovery.

1.1 The layers

Almost every engine, relational or not, is built from the same layers. A request travels down them, and data travels up.

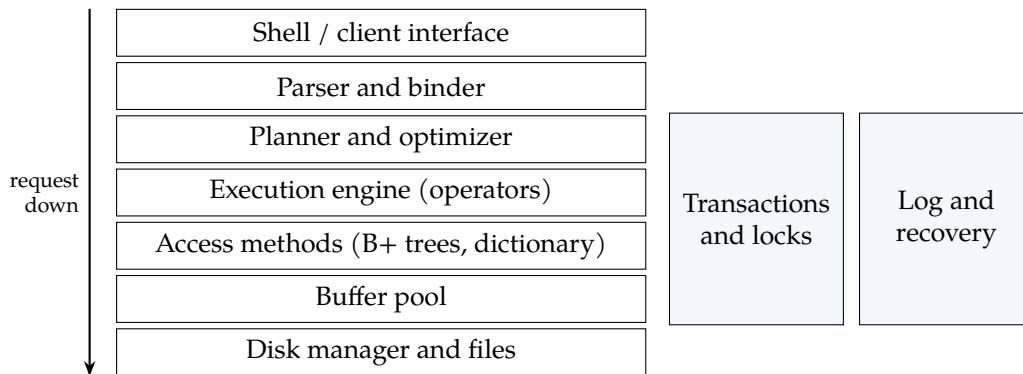


Figure 1.1: The layers of a database engine. Transactions and recovery cut across the stack.

Interface Reads text, returns results. For OntoDB this is the `ontodb` shell.

Parser and binder Turn text into a tree, check names, and replace each constant with an internal id.

Planner and optimizer Choose *how* to answer: which index to use, in what order to join, with which algorithm.

Execution engine Runs the chosen plan as a tree of operators that pass rows to each other.

Access methods The data structures that find data: B+ trees for ranges, hash tables for exact matches, the dictionary for term ids.

Buffer pool Caches disk pages in memory, decides which to evict, and enforces the write-ahead rule when a page goes back to disk.

Disk manager Reads and writes fixed-size pages of a file.

Two subsystems do not fit in one layer. The **transaction manager** and **lock manager** touch every read and write. The **log manager** and **recovery** touch every update, every page write, and every restart. Most of the difficulty of building an engine lives in how these cross-cutting pieces interact with the layers.

Key idea

Each layer hides a cost from the one above. The buffer pool hides the disk; the B+ tree hides how many pages a lookup needs; the optimizer hides that a query can be answered in a thousand ways. Understanding an engine means knowing what each layer hides, and when the hiding fails.

1.2 RDF in one page

RDF represents knowledge as a set of *triples*. A triple is (*subject, predicate, object*): the subject is a thing, the predicate a relationship, and the object another thing or a value.

<http://ex.org/Alice>	<http://ex.org/name>	"Alice"
<http://ex.org/Alice>	rdf:type	<http://ex.org/Student>
<http://ex.org/Metal>	rdfs:subClassOf	<http://ex.org/Material>

Things are named by IRIs (<http://...>); values are literals, optionally with a language ("Métal"@fr) or a datatype ("21"^^xsd:integer). An *ontology* is RDF that describes concepts and their relations: Metal subClassOf Material, Alloy subClassOf Metal. OWL, the standard ontology language, is itself written as RDF triples, so a triple store can store any OWL ontology without understanding it.

1.3 SPARQL in one page

SPARQL asks questions by *pattern*. A *triple pattern* is a triple in which some positions are variables. A *basic graph pattern* is a set of triple patterns that share variables; its answers are every way of binding the variables so that every pattern becomes a stored triple.

```
PREFIX ub: <http://example.edu/univ#>
SELECT ?name WHERE {
  ?s a ub:Student .           # ?s is a student
  ?s ub:name ?name .         # ... and has a name
}
```

Two patterns that share ?s are a *join*. That is the single most important fact about evaluating SPARQL: a query with *k* patterns is a *k*-way self-join of one large table of triples. Everything in chapter 5 and chapter 8 follows from it.

1.4 The running example

In OntoDB

OntoDB is a database management system for ontologies, written in C++20. It loads Turtle, N-Triples, and RDF/XML (so OWL files load as triples), and it answers SPARQL basic graph patterns with filters, `INSERT DATA`, and `DELETE DATA`. Its layers map one to one onto fig. 1.1: a shell (`tools/shell`), a lexer, parser, and binder (`src/parser`, `src/binder`), a planner and optimizer, volcano executors (`src/execution`), a dictionary and a store (`src/dictionary`, `src/store`), a buffer pool and disk manager (`src/buffer`, `src/storage`), and a log and recovery manager (`src/recovery`).

A single query, followed through OntoDB, touches every layer:

1. The shell reads lines until the query's braces balance.
2. The parser builds a syntax tree; the binder resolves prefixes, turns every IRI and literal into a 64-bit id from the dictionary, and gives each variable a column number.
3. The planner builds a tree: one scan per triple pattern, joins between them, a filter, a projection.
4. The execution engine pulls rows of ids up that tree, one at a time.
5. The shell turns ids back into text through the dictionary and prints them.

An insert adds a sixth step: before the shell says `inserted 1 triple`, a log record describing the insert must be on disk. Why that order is non-negotiable, and how it lets the engine survive a crash, is the subject of part V. First, storage.

Design decision 1: One text interface

Choice. The shell is the only way in. Every caller (the shell, the crash harness, the differential tests) goes through one facade, `Database`, which takes SPARQL text or a backslash command and returns rows. A future query window would send the same text and print the same rows; it would not get its own parser.

Alternatives. A client-server wire protocol, as PostgreSQL and MySQL have; or an embedded library API with one call per operation, as SQLite and RocksDB offer.

Why. One entry point means one code path to test. A crash test, a differential test, and a person at the prompt all exercise exactly the same parser, planner, and log.

Cost. No remote clients. Two “users” are two shell sessions on threads of one process, which is enough to test locking but not a network service.

Design decision 2: Store OWL as triples, with no reasoner

Choice. OWL and RDFS load as plain triples. `Alloy subClassOf Metal` and `Metal subClassOf Material` are two stored facts; the engine does not derive `Alloy subClassOf Material`.

Alternatives. Forward chaining (materialize the closure at load, so queries see derived triples), or backward chaining (rewrite each query to follow `subClassOf` at run time).

Why. The book and the engine are about storage, indexing, transactions, and recovery. Inference is a separate feature with its own costs, and it is on the list of later work, to be measured on LUBM.

Cost. Queries that depend on inference return a subset of the “intended” answer. Several LUBM benchmark queries do exactly that.

Check your understanding

- 1.1 Name the layer that hides each cost: (a) a lookup needs four page reads; (b) the disk is slow; (c) a query has a thousand possible plans.
- 1.2 Why does the transaction manager not fit into one layer of fig. 1.1?
- 1.3 A SPARQL query has five triple patterns that all share `?s`. Viewed as relational algebra over one table of triples, what is it?
- 1.4 A user loads an ontology with `Alloy subClassOf Metal` and `Metal subClassOf Material` and asks for every subclass of `Material`. What does OntoDB return, and why?

Answers are in appendix B.

Part II

Storage

Disks, Files, and Pages

Every design decision in a storage engine starts from one fact: memory is fast and volatile, and storage is slow and durable. A database must keep its data on the slow, durable device and still answer queries at the speed of the fast, volatile one. This chapter is about the unit that moves between the two: the *page*.

2.1 The storage hierarchy

Device	Typical access latency	Survives power loss
CPU cache	1–10 ns	no
Main memory (DRAM)	about 100 ns	no
NVMe SSD, random 4 KiB read	10–100 μ s	yes
Spinning disk, random read	5–10 ms	yes

Table 2.1: Orders of magnitude, not benchmarks. The gap between memory and storage is a factor of a hundred to a hundred thousand.

Two properties of storage shape everything above it.

Block granularity. A device reads and writes whole blocks (typically 512 bytes or 4 KiB). Reading one byte costs a whole block. So the engine should arrange data so that one block read brings many useful bytes.

Sequential beats random. Even on SSDs, sequential access is cheaper than random access, and on spinning disks it is dramatically cheaper. An engine that writes sequentially (a log) and reads mostly through an index (a tree) uses the device the way it wants to be used.

2.2 Pages

A *page* is the engine’s unit of I/O: a fixed-size block of the database file, usually 4, 8, or 16 KiB, aligned to the device’s block size. The engine never reads half a page and never writes half a page. Fixing the size has three payoffs:

- **Addressing is arithmetic.** Page i lives at byte offset $i \times \text{PAGE_SIZE}$ in the file. A page id is all the engine needs to find it.
- **Memory is interchangeable.** Every buffer-pool frame holds exactly one page, so any frame can hold any page (chapter 3).

- **Failure is bounded.** A write either lands a whole page, or, if power fails mid-write, produces a *torn page*: part old, part new. Because the size is fixed and known, a checksum over the page detects the tear.

Key idea

A page id is a stable name. Pointers inside the database (a B+ tree child, a heap record's location) are page ids, never memory addresses, so they remain valid across restarts and across evictions.

2.3 Slotted pages

Fixed-size records are easy to place: record k starts at byte $k \times \text{size}$. Real data is variable-sized: an IRI might be 20 bytes or 200. The standard answer is the *slotted page*.

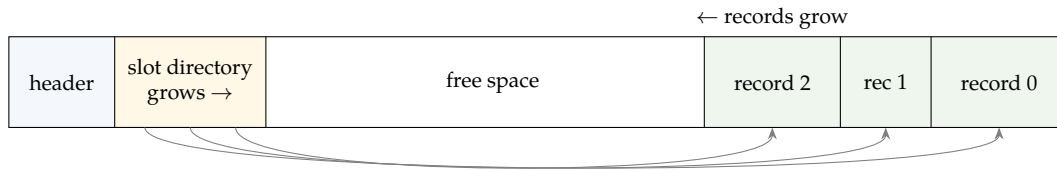


Figure 2.1: A slotted page. The slot directory grows forward from the header, records grow backward from the end, and free space sits between them.

The header records how many slots exist and where free space begins. Each slot holds a record's offset and length. A record is addressed by a *record id* (RID): $(\text{page id}, \text{slot number})$. Two properties make this work:

- **Records can move without changing their RID.** To compact the page, slide records together and update the slot offsets. The slot number, and therefore the RID, is unchanged.
- **Deletion is cheap.** Mark the slot free; a later insert may reuse it. The space is reclaimed when the page is compacted, or when a new record fits in the hole.

In OntoDB: pages and the term heap

Every OntoDB page is 4096 bytes (`PAGE_SIZE`). The first 8 bytes of *every* page, from the very first version, are reserved for the *pageLSN*, the log sequence number of the last update applied to that page; the payload starts at byte 8. Reserving the field before any logging existed means the file format did not have to change when the write-ahead log arrived (chapter 14).

The spellings of IRIs and literals live in a *term heap* made of slotted pages. The B+ tree leaves never contain strings: they contain 24-byte keys of three ids. That separation keeps the tree's fanout high, and it is why a long IRI costs nothing in the index (chapter 5). A spelling is never deleted when a triple using it is deleted: the dictionary keeps the id, so a log record written earlier can still name it.

2.4 Talking to the file system

An engine reads and writes its file through the operating system, and two details of that conversation decide whether data survives a crash.

The page cache. A write call returns as soon as the bytes are in the operating system’s page cache, in memory. They reach the device later. A crash after write returns can still lose them.

fsync. Calling `fsync` on a file asks the operating system to push that file’s dirty data to the device and wait until the device reports it durable. It is the only promise of durability the engine gets, and it is expensive: a few milliseconds on a disk, tens to hundreds of microseconds on an SSD. Two consequences follow.

1. **Batch the expensive call.** Engines amortize `fsync` across many transactions; the log manager’s *group commit* (chapter 14) exists for this.
2. **Never assume a failed `fsync` can be retried.** On several operating systems, after `fsync` reports an error the kernel may already have discarded the dirty pages and cleared the error, so a second `fsync` that “succeeds” has lost data. PostgreSQL learned this in 2018 and now treats a failed `fsync` as fatal [23, 22].

Atomic replacement. Renaming a file over another is atomic on POSIX file systems: a reader sees the old file or the new one, never a mixture. The idiom “write to a temporary file, `fsync` it, rename it over the target, `fsync` the directory” replaces a whole file atomically. OntoDB writes its snapshots this way (chapter 16).

Design decision 3: 4 KiB pages

Choice. `PAGE_SIZE` is 4096 bytes, fixed for the whole system.

Alternatives. 8 KiB (PostgreSQL), 16 KiB (InnoDB), or a size chosen per database at creation.

Why. 4 KiB matches the operating system’s page and a common device block, so one page is one block. It is also small enough that tests with a pool of three or four frames, and trees of a few hundred keys, exercise eviction, splits, and merges quickly.

Cost. Lower fanout than a 16 KiB page, so a very large index may need one more level; and fixed per-page overhead (the 8-byte `pageLSN`, the header) is a larger share of each page.

Design decision 4: Page 0 is a real page

Choice. Page ids are `int32`, and page 0 is a legal page. “No page” is `INVALID_PAGE_ID`, which is `-1`. The catalog (plan 3.3) will initialize its page explicitly, so a zero-filled file cannot be read as “the root is page 0”.

Alternatives. Reserve page 0 as a null page and use 0 as the sentinel.

Why. Zero is the value memory and files have by accident. A sentinel should be a value that never appears unless someone wrote it on purpose; then a missing initialization is a visible error, not a tree rooted at page 0.

Cost. Every reader of a stored page id must check for -1 , and the largest file is 2^{31} pages, 8 TiB at 4 KiB.

Design decision 5: The file's length records allocation

Choice. `AllocatePage` grows the database file by one zero page (`ftruncate`, which usually writes no data blocks). On open, the number of allocated pages is the file size divided by `PAGE_SIZE`, rounded up. There is no header page of the disk manager's own.

Alternatives. A header page (a superblock) holding the next page id and a free list, as SQLite's first page and InnoDB's space header do.

Why. Page 0 already belongs to the catalog. The file system updates a file's length as one fact, so there is no second record of allocation that a crash could leave disagreeing with the file.

Cost. The new length is durable only after an `fsync`, so after a crash a page that a log record names may lie past the end of the file. The disk manager therefore reads any page past the end as zeros and grows the file when one is written there, which lets redo apply such a record with no special case.

Design decision 6: The file only grows

Choice. `DeallocatePage` does not return the id for reuse. A freed page stays in the file, unused.

Alternatives. An in-memory free list, or a durable free-space map such as PostgreSQL's.

Why. An id that is reused while anything still names it (a log record awaiting redo, a stale parent pointer after a crash) turns an old reference into a write to someone else's page. It is the same argument as for term ids: an id's meaning should not change.

Cost. Space freed by merges and deletes is not reclaimed. Reclaiming it safely needs a logged free-space map, which is later work.

Design decision 7: Through the page cache; fsync at shutdown

Choice. Pages are read and written with `pread` and `pwrite` through the operating system's page cache. `WritePage` does not `fsync`; `ShutDown` does, once.

Alternatives. `O_DIRECT`, bypassing the page cache as InnoDB can; or `fsync` after every page write.

Why. Durability is the log's job: a committed change survives because its log record is forced (chapter 14), not because its page is. A page needs to be on disk only at a checkpoint or a clean shutdown. `pread` and `pwrite` carry their own offset, so threads share one file descriptor without a lock around a seek.

Cost. A cached page exists twice, in a pool frame and in the kernel's cache. Until page-level logging exists (plan 8.3), only a clean shutdown makes the page store durable.

Design decision 8: Compact on delete

Choice. Deleting a record from a term-heap page slides every record stored below it up by the record's length and updates their slot offsets, so the page's free space is always one gap. Slot numbers, and so RIDs, do not change.

Alternatives. Mark the slot free and leave a hole, compacting only when an insert cannot find room, as PostgreSQL defragments a page lazily.

Why. Free space is then exactly one number, an insert never has to search for a hole that fits, and the page code stays small.

Cost. A delete moves up to a page of bytes. Terms are rarely deleted (the dictionary never deletes them), so this cost is almost never paid.

Design decision 9: A term must fit on one page

Choice. A record is at most $4096 - 16 - 4 = 4076$ bytes. A longer spelling is refused with an error, and the load or statement that carried it fails.

Alternatives. Overflow pages chained from the record, or storing large values out of line as PostgreSQL's TOAST does.

Why. IRIs and almost all literals are far shorter, and a record that lives on one page is read with one pin and never reassembled.

Cost. A long `rdfs:comment` or an embedded document cannot be stored. Overflow records are the natural next step if real ontologies need them.

Design decision 10: Append to the last page

Choice. The heap is a chain of pages. An insert goes to the last page, and a new page is chained on when the record does not fit there.

Alternatives. A free-space map that remembers how much room each page has, so an insert can use space freed anywhere.

Why. Terms are added and almost never removed, so the last page is nearly always the only one with room. Appending keeps new terms together.

Cost. Space freed on an earlier page is reused only by records that are placed on that page later, which does not happen; it is lost until a compaction of the whole heap.

Design decision 11: Spellings are never deleted

Choice. Deleting the last triple that uses a term keeps the term and its id in the dictionary.

Alternatives. Reference-count terms and reclaim an id when its count reaches zero; or garbage-collect unused terms at checkpoint.

Why. Log records name terms by id. If id 18 could be reclaimed and reused, an old record that says "insert (18, 9, 4)" would silently mean a different triple on replay. Never deleting makes an id's meaning permanent.

Cost. The dictionary only grows. A workload that churns through unique literals keeps every one of them.

Status

The page frame and the pageLSN layout exist, and the disk manager runs (plan 1.1): it allocates pages by growing the file, reads and writes whole pages, and fsyncs on shutdown. The term heap runs too (plan 3.1): slotted pages that compact on delete, chained into a heap that appends to its last page.

Check your understanding

- 2.1 Page 37 of a 4 KiB-page file: at what byte offset does it start, and why does the engine never need a table to find it?
- 2.2 In a slotted page, a record is moved during compaction. Why does no index entry that points to it need to change?
- 2.3 A write returned successfully and then the machine lost power. Was the data durable? What call would have made it so?
- 2.4 fsync returns an error. Why is calling it again and continuing on success wrong?
- 2.5 Why is -1 a better “no page” sentinel than 0?
- 2.6 A 5,000-byte `rdfs:comment` literal arrives in a file being loaded. What does OntoDB do, and how would PostgreSQL store it?
- 2.7 After a crash, a log record names page 812, but the file holds only 800 pages. How can OntoDB’s allocation scheme produce this, and who must repair it?

Answers are in appendix B.

Further reading

Petrov [1], chapters 2 and 3, on file formats and slotted pages. Pillai et al. [22] on what file systems actually guarantee.

The Buffer Pool

The database is larger than memory, but queries must run at memory speed. The *buffer pool* bridges the two: a fixed array of in-memory *frames*, each able to hold one page, plus the book-keeping that decides which pages are in memory and when they go back to disk.

3.1 Frames, the page table, and pins

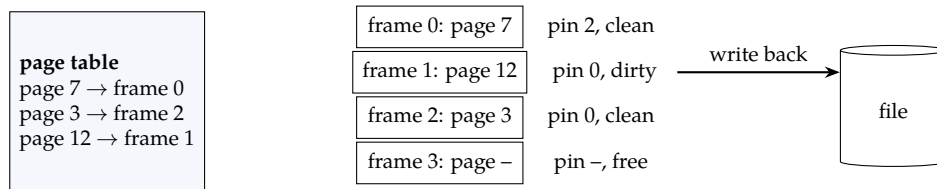


Figure 3.1: A buffer pool with four frames. The page table maps page ids to frames; each frame has a pin count and a dirty bit.

Each frame carries two pieces of state:

Pin count. How many threads are using the page right now. A pinned page must not be evicted: someone holds a pointer into it.

Dirty bit. Whether the in-memory copy differs from the one on disk. A dirty page must be written before its frame is reused.

The interface follows from that:

- `FetchPage(id)`: if the page is in a frame, increment its pin and return it (a *hit*); otherwise choose a victim frame, write it back if dirty, read the page into it, pin it (a *miss*).
- `UnpinPage(id, dirty)`: decrement the pin; mark the frame dirty if the caller changed it.
- `NewPage()`: allocate a page id on disk and return a zeroed, pinned frame.
- `FlushPage(id)`: write a dirty page now.

Key idea

A pin is a promise, and a leaked pin is a leak of memory the pool can never reclaim. Engines therefore wrap pins in a scoped object (a *page guard*) that unpins exactly once when it goes out of scope.

3.2 Choosing a victim

When every frame is occupied and a miss arrives, the pool must evict an unpinned page. The choice matters: a bad policy turns hits into misses, and every miss is a disk read.

LRU. Evict the least recently used page. Simple, and good for many workloads. Its weakness is *sequential flooding*: one large scan touches every page once, and each touch makes a cold scan page look “recent”, so the scan flushes out the hot index pages that every other query needs.

CLOCK. An approximation of LRU that avoids moving pages in a list on every access: each frame has a reference bit set on access, and a “clock hand” sweeps the frames, clearing bits and evicting the first frame whose bit is already clear. Cheap, and still vulnerable to flooding.

LRU-K. O’Neil, O’Neil, and Weikum’s fix for flooding [9]. Remember the last K access times of each page. The *backward K -distance* of a page is the time since its K -th most recent access; a page accessed fewer than K times has infinite distance. Evict the page with the largest distance. With $K = 2$, a page touched once by a scan has infinite distance, while an index root touched thousands of times has a small one. The scan page goes first. LRU is the special case $K = 1$.

Page	Accesses (time)	2nd most recent	Backward 2-distance at $t = 10$
A (index root)	1, 4, 8, 9	8	2
B (index leaf)	3, 6	3	7
C (scanned once)	7	—	∞

Table 3.1: LRU-K with $K = 2$. LRU would evict B (least recently used at time 6); LRU-K evicts C, the page with no second access, even though C was touched more recently than B.

When several pages have infinite distance, a tie-breaking rule is needed. A common choice is the page whose oldest remembered access is earliest; with $K = 2$ such a page has exactly one access, so this is plain LRU among the pages seen once.

3.3 Steal and force

Two policy questions connect the buffer pool to recovery, and their answers decide what the log must contain.

Steal? May the pool write a dirty page of an *uncommitted* transaction to disk, to free its frame? If yes, the disk can hold changes that must later be *undone*.

Force? Must every page a transaction changed be written to disk *before* it commits? If no, a committed change may exist only in memory at the time of a crash, and must later be *redone*.

Steal lets the pool evict any unpinned page, so a long transaction cannot exhaust memory. No force lets a commit write one small log record instead of every page it touched. Both are wins,

	Force	No force
No steal	no undo, no redo (slow, memory-bound)	redo only
Steal	undo only	undo and redo (fast, flexible)

Table 3.2: The four combinations. High-performance engines choose steal and no force, and pay for it with a log that supports both undo and redo.

and chapters 14 and 15 show how the log makes them safe.

In OntoDB: the buffer pool

OntoDB’s pool uses **LRU-K** with $K = 2$ by default (`\set lru_k`), tie-breaking by the older K -th access. Its policy is **steal / no force**. Pages are reached only through `ReadPageGuard` (shared latch, one pin) and `WritePageGuard` (exclusive latch, one pin); both unpin exactly once. The test harness asserts that the number of pinned frames is zero after every query and every destroyed iterator: a leaked pin fails the test even when the query’s rows were right.

When the pool writes a dirty page and a log manager is installed, the order is fixed: first flush the log up to the page’s `pageLSN`, then write the page. A named crash point sits between the two, so a test can kill the process after the log is durable and before the page is. That ordering is the write-ahead rule of chapter 14.

Design decision 12: One I/O thread, first in, first out

Choice. The pool never calls the disk manager itself. It hands a request to a `DiskScheduler`, whose single background thread runs reads and writes in the order they were scheduled. `Schedule` returns at once with a future; if the I/O fails, the error is rethrown from the future’s `get`.

Alternatives. Synchronous I/O on the query thread; a pool of I/O threads; an asynchronous kernel interface such as `io_uring`.

Why. In-order execution makes “write page 9, then read page 9” correct with no per-page bookkeeping. A background thread lets a flush or a prefetch overlap with query work, and errors arrive where the waiting caller can handle them.

Cost. No parallel I/O: an SSD reaches full throughput only with many requests in flight, and one slow read delays every request behind it.

Design decision 13: LRU-K with $K = 2$

Choice. Evict the unpinned frame with the largest backward 2-distance. Frames seen fewer than twice have infinite distance and go first, earliest oldest access first. Time is a logical clock that ticks on every access, not the wall clock, and a pinned frame keeps its history.

Alternatives. LRU, CLOCK, 2Q, ARC.

Why. SPARQL plans scan: an unselective pattern reads every leaf of a permutation once. LRU and CLOCK let that scan push out the index’s upper levels; LRU-K does not. A logical

clock and a fixed tie-break make eviction deterministic, so a test can predict exactly which frame goes; the replacer's tests replay table 3.1 and check that C, not B, is evicted. Keeping history while a frame is pinned stops a hot page from looking new each time it is unpinned.

Cost. Each frame keeps K timestamps, and choosing a victim scans every frame: linear in the pool size, which is cheap for the default 64 frames but would need a priority structure for pools of millions. A second policy is planned as a measured comparison, not as a replacement.

Design decision 14: One latch for the whole pool, held across I/O

Choice. A single mutex covers the page table, the free list, every pin count and dirty bit, and the replacer calls. It stays held while a miss reads its page or a victim is written back.

Alternatives. Partition the page table and give each frame an "I/O in progress" state, so other threads keep working during a read; this is roughly what PostgreSQL does with buffer-mapping partitions and per-buffer I/O locks.

Why. The pool's state is then always consistent: no frame is half-loaded while another thread looks it up, and the tests pass under ThreadSanitizer without subtle protocols. With a single I/O thread (design decision on the scheduler), finer locking would mostly move the waiting elsewhere.

Cost. A miss stalls every thread for one disk read. Finer-grained latching is the first thing to change when I/O becomes parallel.

Design decision 15: The pool's destructor does not flush

Choice. Destroying the pool discards dirty frames. Durable shutdown calls `FlushAllPages` explicitly (plan 3.6).

Alternatives. Flush every dirty page on destruction, which is convenient.

Why. Durability is the log's job. Tests need a path that behaves like a dead process, writing nothing, and a destructor also runs while an exception from a crash point unwinds; a flushing destructor would write exactly the page the test is trying to keep off disk.

Cost. A caller who forgets `FlushAllPages` loses work on a clean exit. The clean-restart test of plan 3.6 is there to catch it.

Design decision 16: A full pool returns null

Choice. When every frame is pinned, `NewPage` and `FetchPage` return `nullptr` at once.

Alternatives. Block until some frame is unpinned, or throw.

Why. The caller usually holds pins of its own; blocking could wait for a pin that only the waiter can release. Returning lets the caller give up the operation and release what it holds.

Cost. Every caller must check for null; guards and the tree turn that into an error close to the cause.

Design decision 17: Pages only through guards

Choice. Code above the pool reaches a page only through a `ReadPageGuard` or `WritePageGuard`: one pin plus a latch, released exactly once by `Drop` or the destructor. Guards move but never copy, and a moved-from guard is empty. Tests assert the pinned-frame count is zero after every query.

Alternatives. Callers call `UnpinPage` by hand, as early textbook pools do.

Why. A leaked pin returns the right rows and silently shrinks the pool; a double unpin lets a page in use be evicted. Neither is visible in query results. Tying the pin to a scope removes both.

Cost. Ownership rules (move-only, drop before pinning the sibling) that every caller must follow, and a little ceremony for one-line accesses.

Design decision 18: Pin, then latch; unlatch, then unpin

Choice. Each frame has a reader–writer latch. A guard asks the pool to pin the page, then takes the latch, outside the pool’s own latch. Dropping releases the latch first and the pin second. A null page from the pool becomes an error at the call that asked for it.

Alternatives. Take the page latch while still holding the pool latch; or release the pin before the latch.

Why. Waiting for a page latch while holding the pool latch would block every thread behind one busy page, and could deadlock with the thread that holds the page and needs the pool. Unpinning first would let the frame be evicted and reused while its latch is still held.

Cost. Between the pin and the latch, another writer may get the page first; callers must not assume what they will find.

Design decision 19: A latch belongs to a page, not to a frame

Choice. When a frame is given to a different page, it gets a new latch. This happens only while the frame is unpinned, and since guards pin before they latch, no thread holds or waits on the old one.

Alternatives. One latch per frame for the life of the pool.

Why. Latch order is a rule about pages: top-down and left to right. With a latch per frame, two unrelated pages that once shared a frame look like the same lock, and `ThreadSanitizer` reported a lock-order cycle in a bulk load on a three-frame pool that could never deadlock. A fresh latch makes the checker see the same thing the protocol is about.

Cost. A small allocation on every page load, next to a disk read that costs far more. (It has to be a new allocation: `ThreadSanitizer` forgets a mutex when its memory is freed, but not when a statically initialized `std::shared_mutex` is destroyed and rebuilt in place, which was the first attempt.)

Design decision 20: A write guard dirties only if written

Choice. A write guard marks the page dirty on drop only if its mutable GetData was called.

Alternatives. Every write guard marks its page dirty.

Why. B+ tree latch crabbing takes exclusive latches on pages it may never change. Marking those dirty would write clean pages back to disk.

Cost. Code that writes through a pointer obtained earlier, after dropping the guard, escapes the dirty bit, which is one more reason never to keep that pointer.

Design decision 21: Inspecting a page goes through the pool

Choice. `\page N` reads page N with a read guard, as any operator would, and prints its pageLSN and the first payload bytes. `\bpm` prints each frame's page, pins, dirty bit, and pageLSN.

Alternatives. Read the page straight from the file, bypassing the pool.

Why. The pool's copy is what the engine sees, including changes that are dirty and not yet on disk. Reading the file would show a page the engine has already moved past.

Cost. Looking is an access: it can load the page, evict another, and add to the page's LRU-K history. The observer changes what it observes, so `\bpm` is best read before `\page`.

Status

The disk manager, the I/O scheduler, the LRU-K replacer, and the pool run (plans 1.1 to 1.4), and so do page guards with per-frame reader-writer latches (plan 1.5) and the shell's `\bpm`, `\page`, and `\trace` (plan 1.6): the buffer pool is complete. It already flushes the log up to a page's pageLSN before writing the page. The default memory backend has no pool, so these commands come alive with the indexed store (plan 3.4). The experiment that checks them is simple: run a scan with a pool of 4 frames and of 64, with $K = 1$ and $K = 2$, and compare hits and misses in `\stats`.

Check your understanding

- 3.1 A frame has pin count 0 and is dirty. What must happen before a different page is read into it?
- 3.2 In table 3.1, add an access to C at time 10. Which page does LRU-K evict at time 11 now?
- 3.3 With $K = 3$, frame X has accesses at times 2 and 9, and frame Y only at time 5. Both are unpinned. Which does OntoDB evict, and why is X's more recent access irrelevant?
- 3.4 Explain sequential flooding, and why $K = 2$ resists it while $K = 1$ does not.
- 3.5 Which cell of the steal/force table needs *undo* information in the log, and why?
- 3.6 A thread schedules a write of page 9 and then, without waiting, a read of page 9 into another buffer. What does the read return in OntoDB, and what could go wrong with a pool of I/O threads?

- 3.7 Two threads miss on two different pages at the same moment. In OntoDB's pool, how long does the second thread wait, and why?
- 3.8 A test arms the crash point between the log flush and the page write, and the exception unwinds through the pool's destructor. Why would a destructor that flushes dirty pages ruin the test?
- 3.9 Why does a guard release its latch *before* it unpins the page, and not after?
- 3.10 You run `\page 7` twice and then `\bpm`. How has inspecting the page changed the pool?
- 3.11 A query returns the right rows, but afterwards one frame is still pinned. What will go wrong later, and how do OntoDB's guards prevent it?

Answers are in appendix B.

Further reading

Petrov [1], chapter 5 on buffer management. O'Neil et al. [9]. Effelsberg and Härder [10] on buffer management principles.

B-Trees and B+ Trees

An index answers “where is the data with this key?” without reading all the data. For data on disk the answer has been the same since 1970: the B-tree of Bayer and McCreight [5], and its variant the B+ tree, described by Comer as ubiquitous [6]. This chapter explains why it fits disks so well, and how it stays balanced.

4.1 Why a tree of pages

A binary search tree has fanout 2: each node has two children, so finding one of N keys takes about $\log_2 N$ steps. If every step is a page read, a billion keys need 30 reads. A B-tree makes each node a whole *page* and packs as many keys into it as fit. With fanout F , a lookup takes $\log_F N$ reads.

Fanout F	Height for 10^9 keys	Height for 10^{12} keys
2	30	40
100	5	6
500	4	5

Table 4.1: Height is $\lceil \log_F N \rceil$. A large fanout turns a billion-key lookup into four page reads, and the top levels are almost always in the buffer pool.

The upper levels are tiny (a tree of fanout 500 has 1 root and 500 children above a quarter-million leaves) and hot, so they stay in the buffer pool. In practice a lookup costs one or two actual disk reads.

Key idea

A B-tree is a binary search tree redesigned for block devices: make each node a page, make the fanout as large as the page allows, and the tree is shallow enough that its top fits in memory.

4.2 B-tree versus B+ tree

A *B-tree* stores keys and values in every node. A *B+ tree* stores values only in the *leaves*; internal nodes hold only *separator keys* that direct the search. Two consequences favor the B+ tree for databases:

- **Higher fanout.** Internal nodes hold no values, so more separators fit per page and the tree is shallower.

- **Fast range scans.** Leaves are linked to their right sibling. A range query finds its first key with one descent, then walks sideways along the leaf chain without climbing back up.

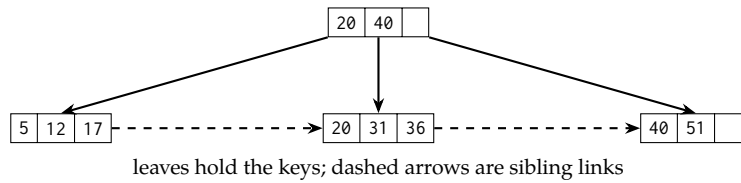


Figure 4.1: A B+ tree. Keys below separator 20 are in the left child; keys from 20 up to 40 in the middle; 40 and above in the right.

The *separator rule* is the tree's central invariant: in an internal node with separators $k_1 < k_2 < \dots < k_n$ and children $c_0, \dots, c_n$, every key in c_{i-1} is less than k_i , and every key in c_i is at least k_i .

4.3 Search

Start at the root. At each internal node, binary-search the separators to pick the child whose range contains the key, and descend. At the leaf, binary-search for the key. A range scan $[\ell, u)$ searches for ℓ , then follows sibling links until a key reaches u .

4.4 Insertion and splits

Insert into the leaf the search reaches. If the leaf has room, done. If it is full, *split* it:

1. Allocate a new leaf. Move the upper half of the keys into it.
2. Link it: the new leaf's sibling is the old leaf's sibling, and the old leaf's sibling becomes the new leaf.
3. Insert a separator into the parent: the first key of the new leaf, which is *copied up* (it stays in the leaf).
4. If the parent is now full, split it too. An internal split *pushes up* its middle key: the key moves to the grandparent and is removed from both halves.
5. If the root splits, create a new root with two children. This is the only way the tree grows taller, so every leaf is always at the same depth.

A classic bug lives in step 3: if the middle key is copied up *and* left in the *left* leaf, the separator rule breaks, because the right child's keys must be at least the separator while a copy of it sits on the left. Writing an invariant checker before writing insert catches this at once.

4.5 Deletion

Remove the key from its leaf. If the leaf is still at least half full, done. Otherwise:

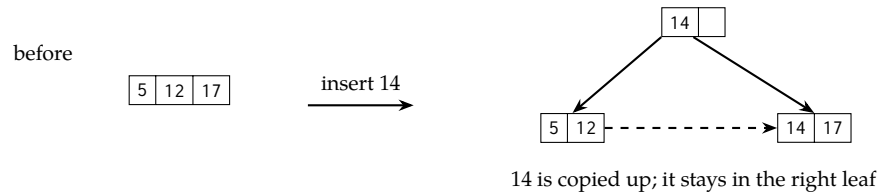


Figure 4.2: A full leaf with capacity 3 splits on insert. The new right leaf’s first key becomes the separator.

- **Redistribute:** if a sibling has more than half, move a key across and update the parent’s separator, which must now reflect the new boundary.
- **Merge:** otherwise combine the leaf with a sibling, remove the separator between them from the parent, and free the empty page. The parent may now underflow, and the process repeats upward; if the root is left with one child, that child becomes the root.

Many production systems relax deletion and allow underfull nodes, merging only when a node empties: deletes are rarer than inserts, and a tree that gets a little sparse rarely matters.

4.6 Bulk loading

Building a tree by inserting keys one at a time leaves leaves about two thirds full and touches pages in random order. When the keys are available up front, *bulk loading* is far better: sort the keys, fill leaves completely from left to right, link them, then build each internal level from the first keys of the level below. One sequential pass per level, and every page is full.

4.7 Iterators and pins

A range scan returns rows one at a time to the query engine, so it is an *iterator* that holds a position: the current leaf and an index inside it. The leaf must stay pinned while the iterator points into it. When the iterator moves to the sibling, it should unpin the current leaf *before* pinning the next one; holding both, plus a parent, can exhaust a small buffer pool and deadlock.

In OntoDB: the B+ tree

OntoDB’s index is a B+ tree. Its key is a 24-byte TripleKey (chapter 5), and the leaf stores the key itself: the triple *is* the record, so there is no separate tuple id. Leaves link to their right sibling, the header page stores the root (or INVALID_PAGE_ID for an empty tree), and a scan holds no pin and no latch between rows: it copies the rest of a leaf’s keys and lets the leaf go (design decision below).

An invariant checker (CheckInvariants) verifies sizes, key order, sibling links, and the separator rule, and the shell can draw the tree (\tree spo dot). Splits and merges are multi-page operations, so they have named crash points where a recovery test may kill the process; chapter 15 explains why a split is redone but never undone. Bulk load sorts ids, not strings, because the tree’s order is the order of the ids.

Design decision 22: Fixed slots, no key compression

Choice. A page header of 12 bytes follows the pageLSN: type, size, and max size. A leaf then holds its right sibling's id and an array of 24-byte keys, 169 per 4 KiB page. An internal page holds slots of a 24-byte key and a 4-byte child id, 144 keys and 145 children per page. Every key is stored whole.

Alternatives. Prefix compression, which stores each key as the bytes it does not share with its neighbour (most keys in an SPO leaf share their subject), or a slotted layout with variable-length keys.

Why. Fixed slots make `KeyAt(i)` an address computation and binary search trivial, and a key copied out of a page is complete on its own. With fanouts above 140, a billion triples still fit in a tree of five levels.

Cost. Keys that share a long prefix waste most of their bytes, roughly a third of each leaf for a subject-heavy SPO index. Leaf compression is on the list of later work, with the question of what it does to prefix bounds.

Design decision 23: The key is the record

Choice. A leaf stores the 24-byte `TripleKey` and nothing else. There is no value field and no record id pointing to a heap.

Alternatives. The relational layout: a heap file of tuples, with each index mapping a key to a record id.

Why. A triple of ids *is* its whole content. A heap would store the same 24 bytes again and cost an extra page read per match.

Cost. Each permutation is a complete copy of the data, so the number of permutations is a space decision (chapter 5).

Design decision 24: Strict deletion

Choice. Remove keeps every node except the root at least half full, by redistributing from a sibling or merging with it.

Alternatives. Lazy deletion, as many production systems do: let nodes become sparse and free a page only when it is empty.

Why. The invariant checker can then assert a fill factor, and merges become real multi-page structure changes with crash points, which the recovery chapters need.

Cost. More page writes on delete-heavy workloads, and the most intricate code in the tree.

Design decision 25: The invariant checker comes before insert

Choice. `CheckInvariants` (size, order, sibling links, the separator rule) is built and tested before `Insert`, and every insert and delete test runs it.

Alternatives. Test insert by looking up the keys afterwards.

Why. Lookups can succeed on a tree that is already wrong; the copied-up-key bug of this chapter does exactly that. A checker turns a structural mistake into a failure on the operation that caused it.

Cost. Code that never runs in production paths, and a full-tree walk in every test.

Design decision 26: Hold the path, not parent pointers

Choice. Pages store no pointer to their parent. An insert or delete keeps the pages it may have to change latched on its way down (the latch crabbing of chapter 13), so a split or merge finds its parent already in hand.

Alternatives. A parent pointer in every page.

Why. A split of an internal page with 145 children would have to rewrite the parent pointer of half of them, which is dozens of page writes for one insert. A parent pointer is also one more thing a concurrent split can make stale.

Cost. The latched path is pinned: an insert into a tree whose pages are all full pins every level, so the pool must be larger than the tree's height plus two. The first, single-threaded version remembered page ids instead and re-read parents; that was correct only while one thread changed the tree.

Design decision 27: Split in the middle

Choice. A full leaf splits into two halves, and the right half's first key is copied up. A full internal page splits around its middle key, which moves up.

Alternatives. Split at the insertion point when keys arrive in increasing order, leaving the left page full, as several engines do for append-heavy indexes.

Why. Triples arrive in no particular order across three permutations, and a middle split is the simplest rule that keeps every page at least half full.

Cost. Inserting sorted keys one at a time leaves every leaf half full, twice the pages a scan must read. Loading sorted data is what bulk load is for.

Design decision 28: Bulk load packs leaves full

Choice. BulkLoad fills each leaf to its maximum, left to right, then builds each internal level the same way. If the last node of a level would fall below its minimum, it and its left neighbour split their entries evenly.

Alternatives. Leave free space in each leaf, a *fill factor*, as PostgreSQL does for its B-tree indexes (90% by default).

Why. An ontology is loaded and then queried. Full leaves mean the fewest pages for every scan, and loading is one sequential pass per level.

Cost. The first insert into a full leaf splits it at once, so a tree that is loaded and then edited heavily ends up looking like one built by inserts.

Design decision 29: A scan holds nothing between rows

Choice. When the iterator reaches a leaf, it copies the leaf's remaining keys (up to the scan's upper bound), notes the right sibling's id, and releases the leaf. It latches the sibling only when those keys run out.

Alternatives. Keep the current leaf pinned and read-latched for as long as the iterator sits on it, as the first version did.

Why. A join keeps one scan open while it opens others on other trees, so held leaf latches were taken in both orders across trees, and Linux's ThreadSanitizer flagged the cycle. With a writer waiting between two such readers, that is a real deadlock, and a transaction waiting for a lock while its scan held a latch would be one the lock manager cannot see. A scan that holds nothing cannot take part in either.

Cost. Up to a page of keys copied per leaf. A key inserted into a leaf after the scan copied it is not returned, and the copied keys may include one deleted since; the scan sees each leaf as of the moment it read it, which is what an unlocked scan promises, and transactions add locks on top.

Design decision 30: A load arrives as one batch

Choice. The loader parses a file (or several) completely, interning terms as it goes, and hands the triples to the store in one call. The indexed store, if it is empty, sorts each permutation's keys and bulk-loads every tree; if it already holds triples, it inserts one at a time.

Alternatives. Insert each triple as it is parsed; or rebuild the trees from old and new keys on every load.

Why. Measured on LUBM(1), the 15 files in one batch load in 1.1 seconds instead of 1.7, into 10.8 MiB instead of 16.2, and every scan reads about half the pages, because bulk load packs the leaves that random inserts left half full. Building three trees costs barely more than one; the time left is the dictionary's.

Cost. A batch holds every triple of the load in memory, and only the first load into an empty store is bulk-loaded; later loads pay for inserts and leave half-full leaves again.

Design decision 31: Small nodes in tests

Choice. The tree's maximum node sizes default to what a page holds (169 and 144 keys), and tests pass sizes as small as 2 to grow deep trees from a few dozen keys. A randomized test inserts and deletes against a reference set with node sizes of 2 to 5, checking every invariant after every operation.

Alternatives. Test only with full pages, or shrink `PAGE_SIZE` for tests.

Why. With 169 keys per leaf, a tree needs tens of thousands of keys to reach three levels, so bugs in internal merges and root changes would hide. The same code runs with either size; only two numbers change.

Cost. Small nodes make some cases more common than they are in real use and others rarer, so the full-size tests stay as well.

Status

The B+ tree is complete (plans 2.1 to 2.7): the page layout, the invariant checker and ToDot, point search, insert with splits, the prefix iterator, bulk load, and delete with merge and redistribution, with all four crash points. It is safe for concurrent use by latch crabbing (plan 7.1), and the shell's `\tree` draws it on the indexed backend. Loading into an empty indexed store goes through bulk load (plan 6.1), and LUBM(1) loads and answers queries on a 32-frame pool (plan 6.2).

Check your understanding

- 4.1 OntoDB's internal page has 145 slots of 28 bytes after a 20-byte header. Where do the 28 bytes come from, and why does slot 0's key go unused?
- 4.2 A page holds 300 separators. About how many levels does a tree of 10^8 keys need, and how many of those are likely in the buffer pool?
- 4.3 During a leaf split the separator 14 is copied up. Which leaf must it stay in, and what breaks if it stays in the other one?
- 4.4 Why does an internal split *push up* its middle key, while a leaf split *copies* it?
- 4.5 Why should a scan unpin its current leaf before pinning the sibling?
- 4.6 Why does OntoDB not store a parent pointer in each page? What does the choice cost once several threads change the tree?
- 4.7 OntoDB bulk-loads 170 sorted keys with a leaf maximum of 169. What leaves does it build, and why not one leaf of 169 and one of 1?
- 4.8 Bulk load fills every leaf. Why can't a loop of Insert calls do the same?

Answers are in appendix B.

Further reading

Bayer and McCreight [5]; Comer [6]; Petrov [1], chapters 2, 4, and 6.

Indexing RDF

A relational table has a schema, and each column can be indexed on its own. An RDF store has one table with three columns, and a query can bind any subset of them. This chapter shows how a triple store turns that into fixed-size keys in B+ trees: first by replacing strings with integers, then by laying out the integers so that byte order is query order, and finally by keeping more than one ordering of the same triples.

5.1 Dictionary encoding

IRIs are long and repeated: the same predicate appears in millions of triples. Storing strings in the index would waste space and make every comparison a string comparison. Instead, a *dictionary* maps each distinct term to a dense integer id, once.

id	canonical spelling
1	<http://example.edu/univ#Alice>
2	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
3	<http://example.edu/univ#Student>
4	"Alice"

A triple becomes three integers, (1, 2, 3). Comparing ids is one machine instruction, and the index stores 24 bytes per triple regardless of how long the IRIs are.

Two rules keep the dictionary correct:

- **One spelling per term.** The loader and the query binder must produce the same canonical text for the same term, or one IRI becomes two ids and every join on it silently fails. Escapes, datatype IRIs, and language tags are part of the spelling.
- **Ids never change.** An id assigned once keeps its meaning across restarts. A log record that names id 18 must still mean the same term when it is replayed (chapter 16).

In OntoDB: the dictionary

Ids are 64-bit and start at 1; 0 is never a valid id, so a zeroed key never names a real term, and the maximum value means “unbound” and is never assigned (see the design decision below). A single codec produces canonical spellings for both the loader and the binder: <iri>, “lex”, “lex”@en, “lex”^^<datatype>, and _:b0. Numeric literals compare by value in filters; everything else compares by id for equality and by spelling for order.

5.2 A key whose bytes sort correctly

A B+ tree compares keys. The fastest comparison is `memcmp` on raw bytes, which works only if byte order equals the intended order. For a key made of three unsigned integers, that requires **big-endian** encoding: the most significant byte first.

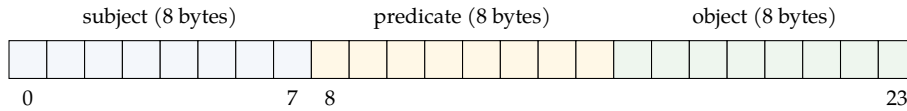


Figure 5.1: A 24-byte triple key: three big-endian unsigned 64-bit ids.

Why big-endian: little-endian stores the least significant byte first, so the id 256 (00 01 00. . . in little-endian) would compare below the id 1 (01 00 00. . .). With big-endian, integer order and byte order agree, and the comparison needs no decoding at all.

5.3 Prefix scans

A triple pattern with some positions bound is a *prefix* of the key, if the bound positions come first. “Every triple with subject 5” is every key whose first 8 bytes encode 5. Such a prefix is a contiguous range of keys, so a scan is one descent plus a walk along leaves. The range is $[lower, upper)$, where lower is the prefix padded with zeros and upper is the prefix’s last component plus one:

Bound components	Lower bound	Upper bound (exclusive)
0	(0, 0, 0)	past every key
1	(5, 0, 0)	(6, 0, 0)
2	(5, 7, 0)	(5, 8, 0)
3	(5, 7, 9)	(5, 7, 10)

5.4 One ordering is not enough

An index ordered subject, predicate, object (SPO) serves any pattern whose bound positions are a prefix of S-P-O. It does not serve “which subjects have object Material?”: the object is the last component, so the matching keys are scattered across the whole tree. The fix is to store the same triples in more than one order.

Hexastore [13] keeps all six permutations; RDF-3X [14] also keeps six but compresses them heavily and adds aggregated indexes. Each permutation is another full copy of the triples, so the right number is a measurement, not a default: three cover every pattern as a prefix, and the other three only add sorted orders that some joins can exploit.

Bound positions in the pattern	Answered as a prefix of
subject	SPO
subject, predicate	SPO
predicate	POS
predicate, object	POS
object	OSP
object, subject	OSP
all three	any (a point lookup)
none	any (a full scan)

Table 5.1: Three permutations (SPO, POS, OSP) make every bound/unbound combination a prefix scan.

Key idea

Extra permutations change the *cost* of answering a pattern, never the *answer*. With only SPO, every pattern is still answerable by scanning and filtering; the other permutations turn scans into prefix ranges.

In OntoDB: keys, permutations, and the catalog

OntoDB's TripleKey is exactly fig. 5.1: three big-endian uint64s, compared with `memcmp`. `PrefixLower(key, n)` and `PrefixUpper(key, n)` produce the bounds for $n = 0, 1, 2, 3$ bound components. The upper bound adds one to the prefix, read as a big-endian number, carrying left. When nothing lies above the prefix ($n = 0$, or every bound id is the maximum) it saturates to the all-0xFF key, three copies of `INVALID_TERM_ID`, which the dictionary never assigns. The catalog names three B+ tree roots, "spo", "pos", and "osp", and an insert or delete updates every permutation the catalog has, or none. The plan builds SPO first and then *measures* whether POS and OSP pay for their space on real SPARQL workloads before adding them; the next design decision records that measurement.

Design decision 32: Ids start at 1; 0 and the maximum are reserved

Choice. Term ids are dense 64-bit integers starting at 1. Id 0 is never used. The largest id, `INVALID_TERM_ID` ($2^{64} - 1$), is the engine-wide marker for "unbound": a row slot that no pattern has filled, a pattern position that is a variable. Dictionary::CheckAssignable refuses to hand it out, so every dictionary enforces the reservation, not just the arithmetic.

Alternatives. An optional per row slot (a flag beside every id), or a separate bitmap of which slots are bound.

Why. A row stays a plain array of uint64, which joins compare and hash directly. And because no stored triple can contain the maximum id, the all-0xFF key lies above every real key, which the prefix bounds use.

Cost. Two ids out of 2^{64} , and a contract every dictionary must keep; a dictionary that skipped the check could hand a real term an id that every operator reads as "unbound".

Design decision 33: The dictionary is built from the engine's own pages

Choice. On disk, spellings live in the term heap. Two B+ trees index them, both with ordinary 24-byte triple keys: one keyed $(id, page, slot)$, so a prefix scan on the id finds the record; one keyed $(hash, id, 0)$, so a prefix scan on a spelling's hash lists candidate ids, each checked against the heap. A header page holds the term count and the three page ids.

Alternatives. Keep the dictionary in memory and rebuild its hash maps from the heap at every open; or write a dedicated on-disk hash index, such as extendible hashing.

Why. It reuses parts that are already tested, works when the dictionary is larger than memory, and needs nothing rebuilt at startup. Comparing the spelling makes hash collisions harmless.

Cost. Looking up a spelling is two tree descents and a heap read, against one probe of an in-memory hash table, and each term costs 48 bytes of index on top of its spelling. A memory cache in front is the obvious optimization if lookups dominate a load.

Design decision 34: A fixed hash function

Choice. The spelling hash is 64-bit FNV-1a, written out in the code.

Alternatives. `std::hash`, or a faster modern hash from a library.

Why. A hash stored on disk must give the same value in every build, on every machine, forever. `std::hash` promises none of that; a different standard library would silently make every stored entry unreachable.

Cost. FNV-1a is slower and mixes less well than newer hashes; with spellings this short, and collisions resolved by comparison, neither matters.

Design decision 35: Big-endian keys, compared with `memcmp`

Choice. Each id is written most significant byte first, so comparing the 24 bytes with `memcmp` gives numeric (s, p, o) order. `Compare` returns the sign of `memcmp`.

Alternatives. Store native integers and compare by decoding three `uint64s`; or store little-endian and use a custom comparator.

Why. Pages, binary search, and the iterator can treat keys as opaque bytes, with no knowledge of their structure. Note the contrast: the `pageLSN` and log records use the machine's own byte order (little-endian on x86 and ARM), because nothing ever compares them as bytes.

Cost. A byte swap on every encode and decode on little-endian machines, which is negligible.

Design decision 36: The upper bound saturates instead of failing

Choice. `PrefixUpper` adds one to the prefix with carry. When there is nothing above the prefix ($n = 0$, or every bound id is the maximum), it returns the all-`0xFF` key rather than an error or a flag.

Alternatives. Return an optional bound meaning “to the end of the tree”, or a flag the

iterator checks.

Why. One key type for both ends keeps the scan loop to a single test: stop when the key reaches upper. Saturation is safe because of the reserved maximum id above.

Cost. The scan's correctness now depends on the dictionary's contract, which is why that contract is checked in code and tested.

Design decision 37: The catalog points at header pages, not roots

Choice. The catalog, page 0, maps a name ("spo") to the page id of that tree's *header* page; the header page holds the tree's current root.

Alternatives. Store each tree's root page id in the catalog directly.

Why. A root changes whenever the root splits or collapses. With a header page in between, only the tree's own header page changes, and the catalog is written only when an index is created or dropped.

Cost. One more page read to find a root, though the header pages are small in number and stay in the pool.

Design decision 38: The catalog claims page 0 by writing it

Choice. On first open, the catalog finds page 0 all zeros, writes a magic number, an empty table, and an `INVALID_PAGE_ID` dictionary root, and writes the page to the file immediately. A page 0 with other contents is refused, not reinterpreted.

Alternatives. Allocate the catalog like any other page and remember its id somewhere, or trust that page 0 is the catalog.

Why. The disk manager's allocation record is the file's length. Writing page 0 at once makes the file one page long, so no later allocation can hand out id 0 to something else. The magic number distinguishes "never initialized" from "a catalog whose dictionary root happens to be 0".

Cost. An extra page write when a database is created.

Design decision 39: SPO, POS, and OSP, measured

Choice. The default store keeps three permutations: SPO, POS, and OSP.

Alternatives. SPO alone, which answers everything with a scan and a filter; or all six, as Hexastore keeps.

Why. Measured on LUBM(1), about 100,000 triples, with each pattern shape run on a cold pool of 64 frames (table 5.2). With SPO alone, every pattern that binds the predicate or the object but not the subject reads the whole tree, and `?x rdf:type C`, LUBM's most common pattern, is one of them. POS and OSP turn those into range scans, 9 to 45 times fewer page reads. The other three permutations make no pattern cheaper.

Cost. Twice the space of SPO alone and about a quarter more load time. SOP, PSO, and OPS would be worth their space only to an operator that needs a second sort order, such as a merge join.

Permutations	Load (s)	File (MiB)	-p-	-o	-po
SPO	1.4	8.2	1131	1131	1131
SPO, POS, OSP	1.7	16.2	128	25	25
all six	2.9	29.1	143	25	25

Table 5.2: Page reads per pattern on a cold 64-frame pool, LUBM(1), averaged over 20 sampled patterns. The other five shapes cost the same under all three sets. Full results are in `benchmarks/indexing/results.md`.

Design decision 40: Permutations are data, and every write is all-or-nothing

Choice. An index is named by its key order ("spo", "pos", ...), and the store keeps whichever of the six the catalog lists; a fresh catalog gets a default set. An insert or delete goes to every index in turn, and if one throws, the indexes already changed are put back before the error propagates. A scan uses the index with the longest bound prefix and filters the remaining bound positions.

Alternatives. Hard-code SPO, POS, and OSP as three fields of the store; or write to the indexes without undoing a partial write.

Why. The number of permutations is exactly the decision the next section measures, so it must be a knob, not a rewrite. A triple present in one index and missing from another would make answers depend on which index a query happened to use.

Cost. Every write costs one tree insert per index. The undo is in-memory only: it does not survive a crash, which is the log's job (chapter 14).

Design decision 41: The shell's indexed backend lives in a temporary file

Choice. `\set backend indexed` builds a buffer pool, the catalog, a disk dictionary, and the indexes in a temporary database file, and moves every triple across by its spelling. `\set backend mem` moves them back and deletes the file. While the indexed backend is on, statements run one at a time.

Alternatives. Make the page store the database behind `ontodb <dir>`.

Why. A persistent page store needs page-level recovery (plans 8.3 to 8.7) to survive a crash. That recovery now runs in the tests and the crash harness, but the shell has not been moved onto it, so persistent databases still use the memory store and its logical log. A temporary file lets every query run on both backends today, which is what the differential test does, and the tree is single-threaded until latch crabbing (plan 7.1).

Cost. Indexed data disappears when the shell exits, ids differ between backends, and two sessions cannot run statements at the same time.

Status

The triple key runs (plan 0.2): encode, decode, compare, and both prefix bounds, with the carry and saturation cases tested. So do the catalog (plan 3.3), the disk dictionary (plan 3.2), and the indexed store (plan 3.4): every query file in the test suite returns the same rows on the memory store and on the B+ tree store, and `\set backend indexed` switches the

shell to it. The permutation set was measured (plan 3.5), and a store flushed and reopened keeps its triples and terms (plan 3.6).

Check your understanding

- 5.1 Encode the id 258 as 8 big-endian bytes and as 8 little-endian bytes. Which encoding sorts correctly under memcmp against the id 3?
- 5.2 Give the lower and upper bounds for the pattern (5, ?, ?) and for (5, 7, ?).
- 5.3 What is PrefixUpper of the key (5, $2^{64} - 1$, 9) with $n = 2$? With $n = 1$ and subject $2^{64} - 1$?
- 5.4 Which permutation serves ?s ?p <Material>? Which serves <Alice> ?p <Student>?
- 5.5 Why must the loader and the binder share one canonical spelling for terms? Give a literal that two careless implementations might spell differently.
- 5.6 What would go wrong if the dictionary assigned $2^{64} - 1$ to a real term?
- 5.7 In table 5.2, why does adding SOP, PSO, and OPS not reduce the page reads of any pattern? What kind of operator would make them worth their space?
- 5.8 With only the SPO index, which of the eight bound/unbound patterns are answered by a range scan, and which by a full scan plus a filter?
- 5.9 Two different spellings have the same 64-bit hash. What does the disk dictionary do when you look up the second one?
- 5.10 Why does the catalog record each index's header page rather than its root? How often does the catalog page change?

Answers are in appendix B.

Further reading

Weiss, Karras, and Bernstein [13]; Neumann and Weikum [14], the sections on the dictionary and on permutations.

The Log-Structured Alternative

A B+ tree updates pages *in place*: an insert finds the right leaf and changes it. The other great family of storage engines never updates in place. A *log-structured merge tree* (LSM tree), introduced by O’Neil et al. [11], turns every write into a sequential append and pays for it later, in the background. RocksDB, LevelDB, Cassandra, and many modern stores are LSM trees. Knowing why OntoDB does *not* use one teaches as much as knowing why it uses a B+ tree.

6.1 How an LSM tree works

1. **Memtable.** Writes go to an in-memory sorted structure (often a skip list), and to a write-ahead log for durability.
2. **Flush.** When the memtable is full, it is written to disk as an immutable sorted file, an *SSTable*, in one sequential write.
3. **Compaction.** SSTables accumulate. A background process merges them into larger SSTables, dropping overwritten and deleted entries. In *leveled* compaction, level $L + 1$ is about ten times larger than level L .
4. **Reads** check the memtable, then SSTables from newest to oldest, and stop at the first hit.
5. **Deletes** write a *tombstone*, a marker that hides older versions until compaction removes both.

A read that misses must look in many files. *Bloom filters*, one per SSTable, answer “is this key possibly here?” in memory with no false negatives, so most files are skipped without a read.

6.2 Three amplifications

Every storage engine trades three costs against each other.

Write amplification Bytes written to the device per byte of user data. An LSM tree rewrites data at every compaction; a B+ tree rewrites a whole page to change a few bytes.

Read amplification Pages read per lookup. A B+ tree reads one path; an LSM tree may consult several levels.

Space amplification Bytes stored per byte of live data. LSM trees hold obsolete versions until compaction; B+ trees hold half-empty pages.

	B+ tree	LSM tree
Writes	random, in place	sequential, batched
Point reads	one root-to-leaf path	memtable, then levels (Bloom filters help)
Range scans	walk linked leaves	merge iterators across levels
Deletes	remove now, merge pages	tombstone now, remove at compaction
Background work	none required	compaction, always
Best for	read-heavy, range-heavy	write-heavy, ingest-heavy

Table 6.1: The two families, in broad strokes.

Key idea

An LSM tree buys fast writes with deferred work. A B+ tree pays for each write immediately, and gets predictable reads and no background compaction in return.

In OntoDB: why a B+ tree

OntoDB's workload is a SPARQL engine's: a bulk load, then queries that are joins of prefix range scans. Range scans over sorted permutations are a B+ tree's best case, and bulk loading builds full, sequential leaves without any compaction. A delete in OntoDB's tree shrinks the tree in place, inside `BPlusTree::Remove`; there is no tombstone for a reader to skip and no background job that rewrites sorted runs.

The LSM ideas still appear in OntoDB, in smaller roles. Its persistent databases (chapter 16) are log-plus-snapshot: every change is appended to a log, and a checkpoint rewrites the whole state as one sorted image, which is a compaction with a single level. And Bloom filters are a natural fit for "has this concept been modified recently?" checks over a log.

Check your understanding

- 6.1 A workload inserts 10 million triples per hour and runs few queries. Which family fits better, and what does the other one pay?
- 6.2 Why does an LSM delete write a tombstone instead of removing the key?
- 6.3 A Bloom filter says "maybe present" for a key that is absent. What does that cost an LSM read? Can it say "absent" for a present key?
- 6.4 In what sense is OntoDB's snapshot-plus-log a one-level LSM tree?

Answers are in appendix B.

Further reading

O'Neil et al. [11]; Petrov [1], chapter 7; Bloom [12].

Part III

Query Processing

From SPARQL to a Plan

A query states *what* to find. The engine must decide *how*. This chapter follows a SPARQL query from text to an executable tree of operators, and introduces the iterator model that runs it.

7.1 Parsing and binding

The *parser* turns text into an abstract syntax tree: prefixes, the projected variables, the triple patterns, the filters. Syntax that the engine does not support should fail here, with a line and a column, rather than half work later.

The *binder* then resolves names against the database:

- Prefixed names become full IRIs: `ub:Student` becomes `<http://example.edu/univ#Student>`.
- Every constant becomes a dictionary id. For a query, a constant that is not in the dictionary can match nothing; for an insert, it is added.
- Every variable gets a *slot*: a column number, assigned in order of first appearance.

After binding, a row is simply an array of ids indexed by slot. The same variable in two patterns is the same slot, and that shared slot is the join key.

7.2 Logical and physical plans

A *logical plan* says what to compute in terms of algebra: scan each pattern, join the results, filter, project. A *physical plan* picks an algorithm for each step: which index and range to scan, which join algorithm, in what order. The planner produces a correct physical plan; the optimizer (chapter 8) rewrites it into a fast one that returns the same rows.

For the query

```
SELECT ?name WHERE {
  ?s a ub:Student . ?s ub:name ?name . ?s ub:age ?age .
  FILTER (?age > 21)
}
```

a straightforward plan is left-deep, joining the patterns in the order written:

```
Projection [?name]
  Filter (?age > 21)
    NestedLoopJoin
      NestedLoopJoin
```

```
TripleScan {?s rdf:type ub:Student}
TripleScan {?s ub:name ?name}
TripleScan {?s ub:age ?age}
```

7.3 The iterator (Volcano) model

Graefe's *Volcano* model [15] gives every operator the same three-call interface:

- `Init()`: prepare, or restart from the beginning.
- `Next(&row)`: produce the next row and return true, or return false when exhausted.
- `destruction`: release everything held (for a scan, its pinned leaf).

The root operator calls `Next` on its child, which calls `Next` on its children, and so on down to the scans. Rows flow up *one at a time*, on demand. Three properties make this model the default for decades:

Composability. Any operator can sit above any other, because they share an interface.

Pipelining. A row can travel from a scan to the client without any intermediate result being materialized. `LIMIT 10` stops the whole tree after ten rows.

Bounded memory. Only operators that must see all their input first (sort, hash-join build, aggregation) hold rows.

Its cost is one virtual call per row per operator. Modern analytical engines reduce that with *vectorized* execution (batches of rows per call) or by compiling the plan into tight loops, but the logical structure is unchanged.

Key idea

Planning builds a tree; execution pulls rows through it. Keeping the two separate is what allows `\explain` to print a plan without running it, and an optimizer to swap one tree for another.

7.4 The operators

Triple scan. Reads the triples that match one pattern. With an index, the bound positions choose a permutation and a prefix range (chapter 5). If a variable occurs twice in one pattern (`?x ex:knows ?x`), the scan also checks that those positions are equal.

Join. Combines rows from two children whose shared slots agree (chapter 8).

Filter. Passes rows whose expression is true. In SPARQL a type error makes the expression false, not the query fail.

Projection. Keeps the selected slots.

Distinct, limit, offset, sort. Distinct applies to *projected* rows (two triples can share a name). Sort compares values, not ids: ids are assigned in load order, which is not value order.

7.5 Sorting more than fits in memory

A sort must see all its input before it can emit the first row, so it is the one common operator whose memory grows with its input. When the rows do not fit in the memory the sort may use, an *external merge sort* spills to disk in two phases.

1. **Run generation.** Read as many rows as fit in B pages of memory, sort them, and write them out as a sorted *run*. Repeat until the input is exhausted: N pages of input become $\lceil N/B \rceil$ runs.
2. **Merging.** Read the first page of up to F runs at once, and repeatedly emit the smallest row at the front of any of them. One merge turns F runs into one. Repeat until a single run remains, or until few enough remain that the last merge can stream its rows straight to the operator above.

With fan-in F , the number of merge passes is about $\lceil \log_F \lceil N/B \rceil \rceil$, and each pass reads and writes every page once. Doubling memory saves at most one pass; the real lever is that F is large, so most sorts need a single merge.

In OntoDB: planning and execution

OntoDB's binder assigns slots in first-appearance order and represents "not yet bound" with a reserved id. The planner always produces the left-deep tree in textual order with nested-loop joins, which is correct but rarely fast; `\explain` prints it. An executor factory maps each plan node to its executor, and the engine pulls rows with `Init/Next`. An empty pattern, with no triple to match, returns one empty row rather than none: the identity input for a join that has nothing to bind yet.

Design decision 42: Only inserts add terms to the dictionary

Choice. The binder interns a constant only for `INSERT DATA`. In a query or a delete it looks the term up; a term the store has never seen gets no id, and the pattern that names it matches nothing without scanning.

Alternatives. Intern every constant the binder meets, as the first version did.

Why. A read-only query should not write: on the disk dictionary each new constant was a heap record and two tree inserts, and in a persistent database a term added by a query was logged later by an unrelated update. It also lets an impossible pattern take no locks, which the isolation specs rely on.

Cost. A filter constant is still interned, because comparisons look values up by id.

Design decision 43: Unsupported syntax fails at parse time

Choice. OPTIONAL, UNION, property paths, aggregates, and DELETE/INSERT WHERE are recognized by the parser and rejected there with a line and column.

Alternatives. Accept the full grammar and fail in the executor; or skip what is not understood.

Why. A query either runs completely or fails before anything happens. Ignoring an OPTIONAL would silently change the answer, which is worse than refusing.

Cost. The parser must know the shape of features the engine does not implement.

Design decision 44: Volcano, one row at a time

Choice. Every operator implements Init and Next and passes one row of ids per call.

Alternatives. Vectorized execution (a batch of rows per call), or compiling the plan into loops.

Why. It is the simplest model that composes, it makes LIMIT stop the whole tree, and it keeps pin ownership obvious: a scan holds one leaf while it has a current row.

Cost. A virtual call per row per operator. Large analytical scans would be several times faster vectorized.

Design decision 45: Stub plan nodes before their executors

Choice. Sort, Distinct, Limit, Insert, and Delete exist as plan nodes that \explain prints before their executors are written.

Alternatives. Add a plan node only together with its executor.

Why. The plan's shape and the planner's tests can be fixed first; each executor then arrives into a slot that is already wired and printed.

Cost. For a while, \explain showed plans that could not run, and running one threw "not built yet". Since plan 4.6 every node has its executor; insert and delete stopped being a special case in the engine and became an operator like the rest.

Design decision 46: ORDER BY uses a total order on values

Choice. Sort keys compare by value: unbound before blank nodes, before IRIs, before literals; numeric literals first among literals, by value; other literals by lexical form, then datatype, then language; and any remaining tie by id. Rows with equal keys keep their input order.

Alternatives. Compare ids, which is fast and meaningless; or use the filter comparisons, which call some pairs incomparable.

Why. Ids are assigned in load order, so sorting by id sorts by when a term was first seen. A sort needs every pair ordered, which the filter's SPARQL semantics do not give (comparing an IRI with a number is a type error there). Parsed terms are cached for the length of one sort.

Cost. A dictionary lookup and a parse per distinct value sorted. Without a dictionary

(some unit tests), the sort falls back to ids.

Design decision 47: External sort through the buffer pool

Choice. A sort may use half the buffer pool, at least four pages. If its input fits, it sorts in memory. Otherwise it writes sorted runs to chains of pool pages, merges up to (budget – 2) runs at a time, and streams the last merge to its consumer, choosing the earlier run on ties so the sort stays stable. Each run page is freed as soon as it has been read. On the memory backend there is no pool, and every sort is in memory.

Alternatives. Sort in memory always; or spill to temporary files outside the database, as many engines do.

Why. The pool already bounds memory and does the I/O, so the sort’s memory use is a fraction of a number the user set (pool_size), and its pages are ordinary pages that tests can count.

Cost. Spilled pages take new page ids, and the file only grows (chapter 2): every external sort leaves the database file larger, permanently. A sort also competes with the query’s own pages for frames.

Design decision 48: DISTINCT keeps a hash set

Choice. The distinct operator passes a row the first time it sees it, in its child’s order, and remembers every row it has passed in a hash set.

Alternatives. Sort the rows and drop adjacent duplicates.

Why. Hashing streams: the first answer comes out as soon as it is found, LIMIT above it still stops early, and the order the child chose is kept.

Cost. Memory grows with the number of distinct rows. A sort-based distinct could spill to disk; this one cannot.

Design decision 49: Rows have no order without ORDER BY

Choice. The engine promises no order unless the query has ORDER BY. Tests compare answers as multisets.

Alternatives. Always return rows in a fixed order, for example sorted by id.

Why. The order a plan produces depends on the store, the indexes, and the join operator: the memory store returns triples in insertion order, the indexed store in key order. Promising an order would force a sort on every query.

Cost. LIMIT without ORDER BY may return different rows on the two backends, or under different join operators. Both answers are correct, which is why the differential test checks such queries only when the limit is larger than the answer.

Status

Parsing, binding, planning, triple scans, the three joins, filter, projection, distinct, limit, offset, sort, insert, and delete run on both backends as operators (plans 4.1 to 4.6). On the

indexed backend a sort larger than half the pool spills to pages; a test sorts 3,000 rows on an 8-frame pool through three runs and one intermediate merge.

Check your understanding

- 7.1 In `?s a ub:Student . ?s ub:name ?name . ?s ub:age ?age`, which slot does each variable get?
- 7.2 Why does `DISTINCT` apply to projected rows, not to the rows the scans produce?
- 7.3 Why can't `ORDER BY` compare term ids?
- 7.4 A plan is `Limit 10` over a join of two large scans. How much work does the Volcano model do, and how much would a model that materializes each operator's output do?
- 7.5 Why does an empty pattern produce one empty row rather than none?
- 7.6 A sort's input is 1,000 pages and its budget is 10 pages, so its fan-in is 8. How many runs does run generation write, how many merge passes follow, and how many pages are written in all?
- 7.7 `SELECT ?name WHERE { ... } LIMIT 2` returns Alice and Bob on the memory backend and Alice and Cara on the indexed one. Which is wrong?

Answers are in appendix B.

Further reading

Graefe [15]; the W3C SPARQL 1.1 recommendation [18].

Joins and Query Optimization

A SPARQL query of k triple patterns is a k -way join, and its running time depends far more on *how* it is joined than on anything else. This chapter covers the three join algorithms, how an optimizer estimates the size of intermediate results, and how it searches for a good join order.

8.1 Three join algorithms

Let the outer input have $|R|$ rows and the inner input $|S|$ rows.

Nested-loop join. For each outer row, scan the whole inner input and keep the pairs whose shared slots agree. Cost: $|R| \times |S|$ comparisons, no extra memory. Correct for any join, and the fallback when nothing better applies.

Index nested-loop join. For each outer row, substitute its bindings into the inner pattern and *probe an index* for just the matching triples. If the outer row binds the subject and SPO exists, the inner side becomes a prefix range, often one or a handful of triples. Cost: $|R|$ index probes. This is the common, fast case in SPARQL: one pattern finds the entities, and each further pattern looks up an attribute of each entity.

Hash join. *Build* a hash table on the shared slots of one input (ideally the smaller), then *probe* it with each row of the other. Cost: linear in $|R| + |S|$, plus memory for the build side. It wins when both inputs are large and no index makes the probe selective, and it loses when the build side does not fit in memory, or when an index probe was already a point lookup.

Key idea

Index nested loop is fast when the outer side is small and the inner side is indexed on the join key; hash join is fast when both sides are large; nested loop is always correct. An optimizer's job is to know which case it is in.

8.2 Cardinality estimation

To compare plans, the optimizer needs the *cardinality* of each intermediate result: how many rows each operator produces. It estimates them from *statistics* gathered at load time: the total triple count $|T|$, the number of distinct subjects, predicates, and objects, and per-predicate counts.

For a pattern with predicate p and object o bound, the classic estimate assumes the positions are *independent*:

$$|\{(s, p, o)\}| = \frac{|p| \cdot |o|}{|T|},$$

where $|p|$ and $|o|$ are the counts of triples with that predicate and that object. Independence is convenient and frequently wrong. In RDF, predicates and objects are strongly correlated (rdf:type goes with class objects; a name predicate never has a class as its object), so independence can be off by orders of magnitude.

The q -error measures an estimate $\hat{c}$ against the true count c :

$$q(\hat{c}, c) = \max\left(\frac{\hat{c}}{c}, \frac{c}{\hat{c}}\right) \geq 1.$$

It treats over- and underestimates symmetrically, and errors multiply as they propagate up a plan: a join of two inputs each off by 10 can be off by 100.

8.3 Searching for a join order

A left-deep plan over k patterns has $k!$ orders, and each join can use several algorithms. Selinger's System R optimizer [16] made the search tractable with dynamic programming:

```
for each single pattern P: best[{P}] = cheapest access path for P
for size = 2 to k:
  for each subset S of patterns with |S| = size:
    for each pattern P in S:
      candidate = join(best[S - {P}], P) with the cheapest algorithm
      if cost(candidate) < cost(best[S]): best[S] = candidate
return best[all patterns]
```

It keeps one best plan per subset rather than per order, which reduces $k!$ to about $k \cdot 2^k$. A simpler *greedy* strategy (start with the most selective pattern, then repeatedly add the cheapest connected pattern) is a good first step and often good enough for the star-shaped joins common in SPARQL.

8.4 Explain analyze

The single most useful optimizer tool is to *run* the plan and print, next to each operator's estimated rows, the rows it actually produced. The node where the two diverge most is where a statistic is missing, usually a correlation that the independence assumption averaged away.

In OntoDB: the optimizer

OntoDB's optimizer takes the planner's tree and returns another tree that must return the same rows. The shell setting `\set join auto|nlj|inlj|hash` either lets it choose or forces one algorithm, and forcing happens in the optimizer, not the executor, so the plan that `\explain` prints is always the plan that runs. Statistics are collected at load: the triple count,

distinct counts per position, per-predicate counts, and counts for partially bound patterns. Estimates are finite, never negative, and binding one more position never increases them. Join ordering is greedy and left-deep.

Design decision 50: Exact counts for single patterns

Choice. Statistics are collected in one scan of the store: counts per subject, predicate, and object, per pair of positions, and the set of triples. Every single triple pattern of constants therefore has an *exact* cardinality; estimation, and its independence assumptions, begin only at joins.

Alternatives. Distinct counts and per-predicate counts only, combined with the independence formula above; histograms or samples; or RDF-3X's characteristic sets, which summarize which predicates occur together on one subject.

Why. The independence formula is wrong by orders of magnitude on RDF, where predicates and objects are correlated. For a store of this size, exact pair counts are affordable and remove that error from every scan estimate.

Cost. About seven hash entries per triple, collected at load, and stale after updates until the next collection, like a table that has not been re-ANALYZED.

Design decision 51: Join estimates use distinct values

Choice. A join's estimate is System R's $|L| |R| / \max(V(L, x), V(R, x))$ for each shared variable x , where V is the number of distinct values x takes. For a single pattern, V comes from statistics: a predicate's distinct subjects or objects, or the pattern's own count when the other positions are bound.

Alternatives. Assume each side's join values are all distinct, which gives $\min(|L|, |R|)$ for one shared variable; or a fixed selectivity per join.

Why. SPARQL joins come in two shapes that the simple rule cannot both get right. On a store of 100 students taking 3 of 10 courses, joining students to their names (a star on the subject) is 100 rows, and joining enrolments to the course class is 300. Distinct values give both exactly; $\min(|L|, |R|)$ says 10 for the second, 30 times too low.

Cost. Two more counts per predicate, and an assumption that remains: the join values of the smaller side are taken to appear on the larger side. A variable's V in an intermediate result is approximated by the smaller of its V in a pattern and the result's size.

Design decision 52: The inner side is re-opened with the outer row

Choice. Every executor has `InitWith(row)`, which by default is `Init`. A triple scan overrides it: each variable that the outer row already binds becomes a constant in the pattern it hands the store. The nested loop re-opens its inner side this way for every outer row.

Alternatives. Open the inner scan once with its own pattern and compare every inner row with every outer row; or materialize the inner side once and loop over the copy.

Why. With the outer row filled in, the inner pattern usually binds the subject, so on an indexed store each probe is a short prefix range instead of a scan of every triple with that

predicate. The cost per outer row drops from the size of the inner pattern to the size of its match.

Cost. One store scan opened, and one tree descent, per outer row, with no caching between rows that bind the same values.

Design decision 53: Index nested loop probes the store directly

Choice. The index nested-loop join requires its inner side to be one triple pattern. It has no inner executor: for each outer row it builds the probe pattern from the outer row and asks the store, which picks the index whose order starts with the bound positions.

Alternatives. Treat any nested loop over a rebound scan as an index nested loop, since with rebinding it already probes.

Why. With rebinding, the nested loop already probes an index, so the two operators read the same triples. The separate operator skips the inner executor and the generic row merge, and it names the strategy in the plan, where `\explain` and the optimizer can see it.

Cost. Only a scan can be the inner side; a join whose inner side is itself a join stays a nested loop.

Design decision 54: Hash join builds on the inner side

Choice. The hash join reads its inner (right) input once into a table keyed by the ids of the variables both sides produce, then streams the outer input through it. A row whose key slot is unbound is not hashed but compared with every row of the other side.

Alternatives. Build on whichever side is smaller, or partition both sides to disk when the table does not fit (a grace hash join).

Why. Choosing the smaller side needs cardinality estimates, which arrive with the optimizer (plans 5.1 and 5.2); until then the inner side of a left-deep plan is a single pattern, usually the smaller input.

Cost. The whole inner side lives in memory; a large inner pattern can make the build expensive or exhaust memory. There is no spilling.

Design decision 55: Forcing a join method rewrites the plan

Choice. `\set join hash (or nlj, inlj)` is applied by the optimizer (`Optimizer::ApplyJoinMethod`), which swaps each nested-loop node for the chosen one before the plan is explained or run. The executor factory only reads node types.

Alternatives. Keep one join node and let the executor read the setting at run time.

Why. `\explain` then always prints the plan that runs. A test can check that a forced hash plan contains a hash join without running it.

Cost. The optimizer must be invoked even when the user forces everything.

Design decision 56: Greedy, left-deep join ordering first

Choice. Start from the pattern with the fewest estimated rows and repeatedly add the pattern connected to what is already joined (sharing a variable with it) whose join is estimated smallest, producing a left-deep tree. A cross product is taken only when no remaining pattern is connected, and ties keep the order the query was written in. The optimizer runs whenever statistics exist, which in the shell is always.

Alternatives. Selinger’s dynamic programming over subsets, or a search that also considers bushy trees.

Why. Most SPARQL basic graph patterns are stars around one subject, for which greedy is close to optimal, and a greedy order is easy to explain in `\explain analyze`.

Cost. Chains and snowflakes, where a bushy plan or a non-greedy first step wins, can be ordered badly.

Design decision 57: EXPLAIN ANALYZE wraps every executor

Choice. `\explain analyze` builds the plan’s executors with each one wrapped in a counter, runs the query to the end, and prints every node with the optimizer’s estimate beside the rows it actually returned. The inner scan of an index nested-loop join is marked “probed by the join above”, since the join asks the store directly and the scan never runs.

Alternatives. Count only at the root; or sample timings per operator.

Why. The node where estimate and actual diverge most is where a statistic or an assumption failed, and only per-node counts show which node that is. Wrapping keeps the executors unaware of the measurement.

Cost. One extra virtual call per row per operator, and the query really runs: `explain analyze` of a slow query is slow. Updates are planned but not run.

Design decision 58: auto picks the operator per join, and is the default

Choice. With `\set join auto`, the default, each join of the reordered chain chooses its own operator: a cross product stays a nested loop; otherwise an index nested-loop join when the estimated outer side is no larger than the inner pattern, and a hash join, built on the smaller inner pattern, when it is larger.

Alternatives. Cost every operator with a formula for each and take the cheapest; or keep nested loop as the default and let users force the rest.

Why. With rebinding and indexes, a probe per outer row is cheap while the outer side is small, and a hash table pays off once the outer side outgrows the pattern it would probe. On LUBM (table 8.1) the optimizer’s order matters far more than the operator, and auto is never worse than the best forced setting.

Cost. The rule compares row counts, not costs: it ignores that an index probe is a tree descent and a hash build is memory. It can be wrong near the crossover, where both choices cost about the same.

Query	Rows	written	nlj	inlj	hash	auto
Q1	4	2.4	0.0	0.0	0.3	0.0
Q2	0	> 20 s	0.2	0.1	2.6	0.1
Q3	6	8.1	0.1	0.0	0.9	0.0
Q4	0	0.0	0.0	0.0	6.7	0.0
Q14	5916	8.3	7.6	7.8	7.9	7.7

Table 8.1: Milliseconds per LUBM query on LUBM(1), with the whole store in the pool. “written” is the query’s own order without the optimizer; the other columns use the optimizer’s order with the named operator. Many queries return nothing because they need inference. Query 2, written with three unconnected type patterns first, does not finish in 20 seconds without re-ordering. Full table: benchmarks/joins/results.md.

Status

Today the planner always joins in textual order with nested loop, and the nested loop re-binds its inner scan for every outer row (plan 4.1). The index nested-loop join (plan 4.2) and hash join (plan 4.3) run, chosen with `\set join`, and the differential test checks every query file under all three against the memory store. Statistics (plan 5.1), cardinality estimation (plan 5.2), and greedy join ordering (plan 5.3) run: a query written with `?s ?p ?o` first is reordered to start from its selective pattern, and `\explain` shows the reordered plan. The automatic choice of join (plan 5.4) runs and is the default, and `\explain analyze` prints estimated against actual rows for every node (plan 5.5): the optimizer is complete. A query can carry a deadline, which every scan checks; the joins benchmark uses it.

Check your understanding

- 8.1 Outer side 1,000 rows, inner pattern 1,000,000 triples, and the join binds the inner subject with SPO available. Compare nested loop and index nested loop.
- 8.2 When does a hash join beat index nested loop?
- 8.3 With exact pair counts, what is OntoDB’s estimate for $(?, p, o)$, and where can its estimates still go wrong?
- 8.4 A store has $|T| = 10^6$, $|p| = 10^4$, $|o| = 10^3$. What does the independence formula estimate for $(?, p, o)$? The true count is 1,000. What is the q-error?
- 8.5 Why do estimation errors multiply up a plan?
- 8.6 A hash join builds on `?c a ub:Course` (2,000 rows) and probes with `?s ub:takesCourse ?c` (50,000 rows). What does it keep in memory, and how many hash lookups does it make? What changes if the sides are swapped?
- 8.7 `?s a ub:Student . ?s ub:name ?n`: the first pattern matches 1,000 students and the second 1,000,000 triples. How many triples does the nested loop read with rebinding on an indexed store, and how many without it?

- 8.8 Why does forcing a join method in the optimizer, and not in the executor, matter for \explain?
- 8.9 In table 8.1, query 2 takes over 20 seconds in its written order and 0.1 ms after reordering. What does its written order do first, and why can no join operator rescue it?

Answers are in appendix B.

Further reading

Selinger et al. [16]; Graefe [15]; Leis et al. [17] on how good cardinality estimates really are; Neumann and Weikum [14] on RDF selectivity.

Beyond Basic Graph Patterns

A basic graph pattern is a conjunction: every triple pattern must match, or the row is gone. Most questions people ask of an ontology are not conjunctions. “List every material, with its melting point *if we know it*” keeps a material whose melting point is missing. “Metals or ceramics” is a disjunction. “How many subclasses does each class have” collapses rows into groups. This chapter adds the operators behind those questions, OPTIONAL, UNION, and aggregation, and the update form whose triples come from a query. Each one is a relational operator the earlier chapters did not need, and each one constrains the optimizer in a way an inner join does not.

9.1 Where a conjunction loses rows

Take three materials, two of them with a melting point:

```
:Steel    :meltsAt 1370 .
:Copper   :meltsAt 1085 .
:Steel a :Material .   :Copper a :Material .   :Glass a :Material .
```

The pattern $\{ ?m \text{ a :Material . } ?m \text{ :meltsAt ?t } \}$ is a join of two scans on $?m$ (chapter 8), and it returns two rows. Glass is not wrong; it is unknown, and the inner join cannot say “unknown”. The fix is an operator whose output keeps every row of one side.

9.2 OPTIONAL is a left outer join

SPARQL’s algebra works on *solution mappings*: partial functions from variables to terms. Two mappings are *compatible* when they agree on every variable both bind. For multisets of mappings Ω_1 and Ω_2 and a filter expression e [18, 26]:

$$\text{LeftJoin}(\Omega_1, \Omega_2, e) = \text{Filter}(e, \text{Join}(\Omega_1, \Omega_2)) \cup \{ \mu_1 \in \Omega_1 \mid \neg \exists \mu_2 \in \Omega_2 : \mu_1 \sim \mu_2 \wedge e(\mu_1 \cup \mu_2) \}.$$

The first part is the inner join; the second keeps each left mapping that found no compatible partner, with the right side’s variables *unbound*. For the example, $?m \text{ a :Material OPTIONAL } \{ ?m \text{ :meltsAt ?t } \}$ gives:

$?m$	$?t$
:Steel	1370
:Copper	1085
:Glass	(<i>unbound</i>)

An unbound value is not a value. It is a hole in the row, and the executor has to carry it: rows gain a “not bound” marker per column, which the result printer shows as an empty cell.

9.2.1 A filter inside OPTIONAL is part of the join

The e in the definition matters. Compare

```
SELECT ?m ?t WHERE { ?m a :Material OPTIONAL { ?m :meltsAt ?t FILTER(?t > 1200) } }
SELECT ?m ?t WHERE { ?m a :Material OPTIONAL { ?m :meltsAt ?t } FILTER(?t > 1200) }
```

The first keeps all three materials: Steel with 1370, and Copper and Glass with ?t unbound, because the filter only decides which right rows may join. The second filters the finished rows, and a comparison with an unbound value is an *error*, which FILTER treats as false: it returns Steel alone. Moving a filter across an OPTIONAL boundary changes the answer, so the planner must not do it.

Key idea

An outer join is not commutative, and filters do not move freely across it. The left side is preserved; the right side only decorates it.

9.3 Executing a left join

Both join algorithms of chapter 8 extend to outer joins with one extra bit of state.

Nested loop. For each left row, rebind and scan the right side. If no right row matched (after the join filter), emit the left row once with the right columns unbound. The bit is “did this left row match”, and it lives for one iteration of the outer loop.

Hash join. Build the hash table on the *right* side and probe with the left. A probe that finds no partner emits the left row padded. If the build side were the left one, every left row would need a “matched” flag in the table and a final pass over the table to emit the unmatched ones; that works, but it makes the operator blocking at its end as well as its start.

An index nested-loop left join is the common case in practice: the left side is a selective pattern, and the optional part is a lookup by subject on SPO.

Design decision 59: Left joins build on the right (planned)

Choice. The hash left join builds on the optional (right) side and probes with the preserved (left) side.

Alternatives. Build on the left with a matched flag per entry, and emit unmatched entries after the probe phase.

Why. Padding happens at probe time, the operator streams its output, and the optimizer’s choice is only whether the right side fits in memory.

Cost. When the right side is much larger than the left, building on it costs memory a left build would not.

9.4 UNION

`{ P1 } UNION { P2 }` evaluates both sides and concatenates their rows. SPARQL union is a bag union: a row produced by both sides appears twice, and `DISTINCT` removes duplicates across sides. The sides may bind different variables, so a row from one side has the other side's variables unbound, the same marker `OPTIONAL` introduced. The executor is the simplest operator in the engine, a concatenation of two child iterators; the work is in the binder, which computes the output columns as the union of both sides' variables.

9.5 Aggregation

`GROUP BY` partitions rows by the values of its keys and computes one row per group:

```
SELECT ?c (COUNT(?sub) AS ?n) WHERE { ?sub rdfs:subClassOf ?c }
GROUP BY ?c HAVING (COUNT(?sub) > 2)
```

There are two classic implementations [15]:

Sort aggregation. Sort by the group keys (the external merge sort of chapter 8 already exists), then scan once, finishing a group whenever the key changes. Output comes out ordered by the keys.

Hash aggregation. Keep a hash table from group key to an accumulator and update it per row. One pass, no sort, but the table must hold every group.

Accumulators are small: `COUNT` is a counter, `SUM`, `MIN`, and `MAX` one value, `AVG` a sum and a count, divided at the end. `COUNT(DISTINCT x)` is the exception: it needs the set of distinct values per group, so its memory grows with the data, not with the number of groups. `HAVING` is a filter on the output of the aggregation.

Design decision 60: Hash aggregation, and what to do when groups outgrow memory (planned)

Choice. Hash aggregation. When the table exceeds its memory budget, the plan records whether to spill or refuse; the leading design is to partition the input by a hash of the key into files, then aggregate each partition in memory, as hybrid hash join does for joins.

Alternatives. Sort aggregation always; or refuse with a clear error past the budget.

Why. Ontology workloads group by class or property, and there are far fewer of those than triples, so the table almost always fits and one pass beats a sort.

Cost. Output is unordered, so `ORDER BY` on the keys still sorts; the spill path is code that runs rarely, which is why it needs its own tests.

9.6 Planning around the new operators

The join-order search of chapter 8 treats a basic graph pattern as a set of patterns to permute. With `OPTIONAL`, the pattern becomes a tree, and only parts of it may be reordered:

- Patterns inside the left side may be reordered among themselves, and so may patterns inside the right side. A pattern may not move from one side of an OPTIONAL to the other: that would turn an optional match into a required one, or the reverse.
- The cardinality of a left join is at least that of its left side, since every left row survives. An estimate below that is a bug, not a guess.
- `\explain analyze` shows the new operators with estimated and actual rows, as it does for joins, so a bad estimate is visible.

Pérez, Arenas, and Gutierrez [26] identify the *well-designed* patterns, those where every variable shared between an optional part and the outside also appears on the left; for them, more rewrites are safe, and evaluation is easier. Patterns written by people are almost always well-designed.

9.7 Updates whose triples come from a query

`DELETE { . . . } INSERT { . . . } WHERE { . . . }` rewrites every match: move every class under `:Metal` to `:MetallicMaterial`, for example. The order of the three parts is the whole semantics:

1. Evaluate the WHERE pattern *in full*, producing every solution.
2. Instantiate and delete the DELETE templates for every solution.
3. Instantiate and insert the INSERT templates for every solution.

Interleaving them is wrong for the same reason as the *Halloween problem* of relational updates: a scan that sees its own inserts can match them again, and an update that adds a superclass link per subclass could loop forever or apply twice. Materializing the solutions first gives the update a snapshot of the data as it was before the statement, which is what the specification means. The whole statement is one transaction, under the same two-phase locks as `INSERT DATA` and `DELETE DATA` (chapter 13).

Design decision 61: Evaluate WHERE first, then delete, then insert (planned)

Choice. The solutions of WHERE are materialized before any triple changes; deletes run before inserts.

Alternatives. Pipeline solutions straight into the updates; or insert before delete.

Why. It is the specification's semantics and it avoids the Halloween problem. Deleting first means a triple both deleted and inserted by one statement ends up present, which is what a rewrite intends.

Cost. The solutions are held in memory (or spilled) for the statement's duration.

Status

Planned (plan 12.1–12.5): today OPTIONAL, UNION, aggregates, and DELETE/INSERT WHERE are parsed far enough to fail with a line and a column. Every new form will run on both backends and join the differential test that compares the indexed store with the memory store.

Check your understanding

- 9.1 Write the rows of `?m a :Material OPTIONAL { ?m :meltsAt ?t FILTER(?t < 1200) }` for the three materials above.
- 9.2 Why can a hash left join not simply swap its build and probe sides when the left side is smaller, as an inner hash join can?
- 9.3 A union's left side binds `?a` and its right side binds `?b`. What columns does the result have, and what is in them?
- 9.4 Why does `COUNT(DISTINCT ?x)` need more memory than `COUNT(?x)`, and how does that interact with hash aggregation?
- 9.5 An optimizer estimates 40 rows for a left join whose left input it estimates at 100. What is wrong?
- 9.6 `INSERT { ?c rdfs:subClassOf :Thing } WHERE { ?c rdfs:subClassOf ?d }` runs with the inserts visible to the scan. What can go wrong, and how does materializing `WHERE` first prevent it?

Answers are in appendix B.

Further reading

The SPARQL 1.1 specification [18], section 18, defines the algebra; Pérez, Arenas, and Gutierrez [26] give its semantics and complexity, including well-designed patterns; Graefe's survey [15] covers outer joins and hash and sort aggregation.

Materialized Inference

An ontology says more than it stores. If Alloy `subClassOf` Metal and Metal `subClassOf` Material, then every alloy is a material, though no triple says so; chapter 1 ended on exactly that gap, with a query for materials that missed Alloy. A store that answers ontology queries has to close it. This chapter treats inference as a database problem: the entailed triples are a *view* of the stored ones, and the questions are the old ones for views: compute it on read, or store it and keep it current on every write.

10.1 Two ways to answer with entailments

Rewrite at read time (backward chaining). Turn `?x a :Material` into `?x a ?c . ?c rdfs:subClassOf* :Material`, a property path, and let the executor follow it. Nothing extra is stored; every query pays for the closure it needs, and a query planner must estimate a recursive path.

Materialize at write time (forward chaining). Compute every entailed triple once, store it in the same indexes, and answer `?x a :Material` with one scan of POS, as if the triples had been asserted. Reads stay as fast as they are; writes and space pay.

	Rewrite	Materialize
Read cost	a recursive path per query	one scan
Write cost	none	maintain the closure
Space	asserted triples	asserted plus entailed
Planner	estimate recursion	unchanged

Ontologies are read far more often than they change, and their queries ask for classes *and* their subclasses almost every time, so OntoDB materializes, like RDFS and GraphDB.

10.2 The rules

RDFS entailment is a handful of rules [27]. Written as Datalog, with $T(s,p,o)$ for “the store holds triple (s,p,o) ”:

$T(c, sc, e) \leftarrow T(c, sc, d), T(d, sc, e)$	(rdfs11, subclass is transitive)
$T(x, type, d) \leftarrow T(x, type, c), T(c, sc, d)$	(rdfs9, types go up)
$T(p, sp, r) \leftarrow T(p, sp, q), T(q, sp, r)$	(rdfs5, subproperty is transitive)
$T(x, q, y) \leftarrow T(x, p, y), T(p, sp, q)$	(rdfs7, triples go up)
$T(x, type, c) \leftarrow T(x, p, y), T(p, domain, c)$	(rdfs2)
$T(y, type, c) \leftarrow T(x, p, y), T(p, range, c)$	(rdfs3)

Each rule body is a two-pattern basic graph pattern, so each rule is one join of two scans, and the executor of chapter 8 already evaluates it. The closure is the least set of triples containing the data and closed under the rules.

10.3 Naive and semi-naive evaluation

The obvious way to compute the closure is to apply every rule to everything, add what is new, and repeat until a round adds nothing. That is *naive evaluation*, and it is wasteful: round i rederives every triple of rounds 1 to $i - 1$ before finding the new ones.

The fix is to notice that a triple can be new in round i only if one of the triples it was derived from was new in round $i - 1$. *Semi-naive evaluation* [28] keeps the set Δ of triples new in the last round and evaluates each rule with at least one body pattern restricted to Δ :

$$\Delta_{i+1} = \left(\bigcup_{\text{rules}} (\Delta_i \bowtie T_i) \cup (T_i \bowtie \Delta_i) \right) \setminus T_i, \quad T_{i+1} = T_i \cup \Delta_{i+1}.$$

For a chain of subclasses $A \sqsubseteq B \sqsubseteq C \sqsubseteq D \sqsubseteq E$:

Round	Δ (new subclass triples)	Joins evaluated
0	the 4 asserted links	—
1	$A \sqsubseteq C, B \sqsubseteq D, C \sqsubseteq E$	$\Delta_0 \times T_0$
2	$A \sqsubseteq D, A \sqsubseteq E, B \sqsubseteq E$	$\Delta_1 \times T_1$
3	(none)	$\Delta_2 \times T_2$; stop

Naive evaluation would rejoin all four asserted links in every round. On a deep hierarchy the difference is the difference between joining the frontier and joining the whole closure, round after round.

Key idea

Inference is recursive query evaluation, and semi-naive evaluation is to recursion what a delta is to an update: only what changed can cause anything new.

10.4 Keeping the closure current

10.4.1 Insert

An inserted triple is a closure computation that starts small: run the same rounds with $\Delta_0 = \{t\}$, inside the inserting transaction, so that the closure is complete when the transaction commits. Inserting `Bronze subClassOf Alloy` into a hierarchy above it derives `Bronze subClassOf Metal` and `Bronze subClassOf Material` in two rounds, and the types of every bronze instance in the same rounds.

10.4.2 Delete is harder

Deleting `Alloy subClassOf Metal` should remove `Alloy subClassOf Material`, unless it still follows some other way, for example because a curator also asserted `Alloy subClassOf Material` directly. A triple can have several derivations, and deleting one support must not delete a triple another support still holds. Two classic answers [29]:

Counting. Store, for each triple, how many derivations it has; a delete decrements the counts of what it supported and removes triples whose count reaches zero. Exact for non-recursive rules, but under recursion a triple can support itself through a cycle, and its count never reaches zero.

Delete and rederive (DRed). Three steps: (1) *overdelete*: remove the deleted triple and, by semi-naïve rounds, everything it could have helped derive; (2) *rederive*: for each overdeleted triple, check whether it still follows in one step from what remains, and put it back if so; (3) propagate the rederived triples by semi-naïve insertion.

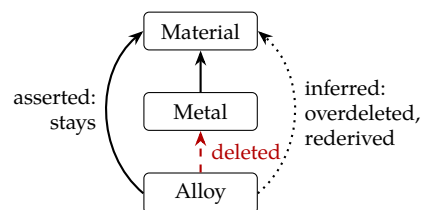


Figure 10.1: DRed on one delete. Overdeletion removes the inferred `Alloy subClassOf Material`; rederivation finds it is also asserted, so the triple is back before the transaction commits.

DRed is correct with recursion because rederivation asks a positive question, “does this follow from what remains”, instead of trusting a count. Its cost is that one delete near the top of a hierarchy overdeletes a large part of the closure and rederives most of it.

Design decision 62: Materialize, and maintain with DRed (planned)

Choice. Entailed triples are computed by semi-naïve evaluation, stored in the indexes, extended on insert, and maintained on delete by delete and rederive. `\set inference rdfs|off`, off by default.

Alternatives. Query rewriting with property paths; counting derivations; recomputing

the closure from scratch after each delete.

Why. Reads stay one scan and the planner stays unchanged. DRed is correct under recursion, which counting is not, and it touches only what the deleted triple could have supported.

Cost. More triples and more index work per write, and a delete high in a hierarchy is expensive. A test checks after random inserts and deletes that the store equals a closure computed from scratch.

Design decision 63: Where inferred triples live (planned)

Choice. To be decided by measurement: a mark on each triple in the existing SPO, POS, and OSP indexes, or separate permutations for inferred triples that scans also read.

Alternatives. The two above.

Why. A mark keeps one scan per pattern but makes deleting all inferred triples (`\set inference off`) a full pass; separate permutations make that a drop, but double the scans of every pattern.

Cost. Either way, a delete must know which triples were asserted, so an asserted triple that is also inferred keeps its asserted status.

10.5 What it costs

Materialization trades space and write time for read time, and the exchange rate depends on the hierarchy's depth. A triple deep in a hierarchy of depth d entails up to d type triples, and a subclass chain of n classes has $n(n-1)/2$ entailed links. LUBM's ontology is shallow, so its closure adds a modest fraction of triples; an anatomy ontology several levels deep adds many more. The plan measures load time and size with and without inference on LUBM(1), and checks that the LUBM queries that need RDFS return the official answer counts.

Status

Planned (plan 13.1–13.5). Today OntoDB stores OWL as triples and has no reasoner: the query for materials in chapter 1 misses Alloy, as it should.

Check your understanding

- 10.1 Give one workload for which rewriting at read time beats materializing, and one for which it does not.
- 10.2 Run semi-naïve evaluation on $A \sqsubseteq B$, $B \sqsubseteq C$ and the type triple $x \text{ a } A$. Which triples are new in each round?
- 10.3 Why does the semi-naïve formula join Δ_i with T_i in both orders?
- 10.4 With counting, a delete in a subclass cycle $A \sqsubseteq B \sqsubseteq A$ leaves both links' counts above zero.

Why, and how does DRed avoid the problem?

- 10.5 An asserted triple is also inferred. A user deletes it. What should the store hold afterwards, and what must the store know to get this right?

Answers are in appendix B.

Further reading

Abiteboul, Hull, and Vianu [28] on Datalog and semi-naive evaluation; Gupta, Mumick, and Subrahmanian [29] on counting and DRed; the RDF 1.1 Semantics [27] for the RDFS rules; Motik et al. [30] on materialization in RDFox; Guo, Pan, and Heflin [31] on LUBM.

Semantic Search

Two questions come up the moment a store holds an ontology. *What is under this class?* is a question about structure, and the store can answer it exactly. *What is close in meaning to this term?* is a question about similarity, and no index built so far in this book can answer it. The B+ trees of chapter 4 are sorted, and it is tempting to think that sorted means close. This chapter shows why that fails, what a hierarchy offers instead, and where the vectors that do answer the second question live.

11.1 What the tree orders

A triple is a 24-byte key of three dictionary ids compared with `memcmp` (chapter 5). The dictionary gives out ids densely, from 1, in the order terms are first seen: the memory dictionary takes `next_++`, and the disk dictionary takes `Size() + 1`. The loader interns the subject, predicate, and object of each statement in file order. So the order of every tree in OntoDB is *first-seen order*. A class sits next to another in the index only if their IRIs happened to appear close together in the input. Nothing sorts by hierarchy, by label, or by meaning.

What the tree does give is *structural* grouping: a bound prefix is one contiguous range, which is why `?x rdf:type C` is a single scan of POS.

11.2 A sort order cannot hold similarity

Suppose we wanted one order on terms so that a range scan around a term returned its neighbors in meaning. Embeddings live in hundreds of dimensions; the question is already hard in two.

Lemma 11.1 (Bandwidth). *Let G be a graph on n vertices with diameter D . Every linear order of the vertices has an edge whose endpoints are at least $(n - 1)/D$ apart in the order.*

Proof. The first and the last vertex of the order are $n - 1$ apart in it, and joined by a path of at most D edges. Some edge of that path covers at least $(n - 1)/D$ of the distance. $\square$

Take the neighbor graph of a $k \times \dots \times k$ grid in d dimensions: $n = k^d$, diameter $d(k - 1)$. Some two grid neighbors are at least $(k^d - 1)/(d(k - 1)) \geq k^{d-1}/d$ positions apart, in *any* order. A scan of width w around a point cannot hold all its neighbors unless w is that large: $\Omega(n^{(d-1)/d}/d)$. With a million points in two dimensions some neighbors are about 500 positions away; in 384 dimensions the bound says ordering buys nothing in the worst case. Space-filling curves (Z-order, Hilbert) keep neighbors close on average and need several range scans to cover a ball; they do not escape the lemma.

Key idea

A B+ tree, or an adaptive radix tree, is a one-dimensional order. It is the right structure for a prefix, a range of keys, or a subtree of a hierarchy. It is the wrong structure for “close in meaning”; that needs a vector index.

11.3 The exception: a hierarchy is a set of intervals

A class hierarchy *does* have an order in which everything below a class is contiguous.

Lemma 11.2 (Trees). *Number the nodes of a rooted forest $1..n$ in postorder, and let $lo(c)$ be the smallest number in the subtree of c . Then d lies in the subtree of c if and only if $lo(c) \leq po(d) \leq po(c)$.*

Proof. A postorder walk numbers a subtree as one contiguous block that ends at its root. $\square$

So “all subclasses of C ” is one range scan over an index keyed by po . Real ontologies are directed acyclic graphs: a class can have several parents (the Gene Ontology has 57 456 is_a edges over 38 092 terms). Following Agrawal, Borgida, and Jagadish [34], choose a spanning forest, number it in postorder, and give every class a *set* of intervals: its own tree interval merged with the intervals of every child.

Lemma 11.3 (Interval sets). *With $I(c) = \text{merge}(\text{own}(c) \cup \bigcup_{c \rightarrow v} I(v))$, class d is at or below c if and only if $po(d)$ lies in an interval of $I(c)$.*

Proof. Induction over a reverse topological order: the numbers covered by $I(c)$ are exactly those of the descendants of c . $\square$

Lemma 11.4 (Size). *Let $E'(c)$ be the non-tree edges whose tail is at or below c . Then $|I(c)| \leq 1 + |E'(c)|$.*

Proof. A descendant d is reached by a path from c . If the path uses no non-tree edge, d is in the tree subtree of c . Otherwise let v be the head of the last non-tree edge on the path; d is in the tree subtree of v . The covered set is therefore $\text{own}(c) \cup \bigcup_{e \in E'(c)} \text{own}(\text{head}(e))$, a union of $1 + |E'(c)|$ intervals, which has at most that many maximal intervals. $\square$

A tree has no non-tree edges, so every class has one interval. Classes on a cycle ($A \sqsubseteq B \sqsubseteq A$) denote the same class; OntoDB collapses them into one component before numbering. Which parent carries a class in the spanning forest changes how many intervals the DAG needs; OntoDB takes the parent with the longest path from a root.

11.3.1 Three ways to answer a type query

Under RDFS, $?x \text{ a } C$ means “ x has a type at or below C ”.

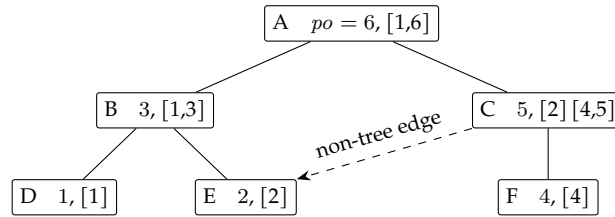


Figure 11.1: A DAG in which E has two parents, B and C . The spanning forest keeps E under B ; the dashed edge is outside it. Each node shows its postorder number and its interval set. C needs two intervals: its own tree interval $[4, 5]$ and $[2]$, the label of E .

Strategy	Extra stored	One query costs
Rewrite	nothing	one POS scan per class below C
Materialize (chapter 10)	a type triple per ancestor of each type	one POS scan
Intervals	one key per type triple	one range scan per interval of C

The interval index stores $(\text{po}(\text{type}), \text{instance})$ once per type triple. An instance insert adds one key. The price is that labels change when the hierarchy does, so OntoDB rebuilds them in $O(n + m)$ instead of maintaining them (Agrawal et al. describe incremental maintenance with gaps; it is not built). A related idea, numbering a tree in preorder and postorder so that axes become regions of a plane, was used for XML by Grust [35].

Design decision 64: Ids stay first-seen; labels are a separate index

Choice. The dictionary keeps handing out dense ids in first-seen order. The postorder labels live in their own B+ tree, $(\text{label}, \text{member}, \emptyset)$, which OntoDB rebuilds when the hierarchy changes.

Alternatives. Renumber the dictionary so ids follow the hierarchy; then the existing SPO, POS, and OSP trees would give subtree scans with no extra index.

Why. Ids are stored in three permutations, in the log, and in snapshots. A hierarchy edit would renumber all of them. A label is derived data.

Cost. One more small tree, and a rebuild after the hierarchy changes.

11.4 Vector indexes

The second question needs vectors. OntoDB keeps one unit-length float vector per term id, so cosine similarity is a dot product. Embeddings come from outside; a small sentence model on a CPU embeds the 38 092 Gene Ontology names in under a minute. OntoDB stores and searches them; it does not run the model.

Flat. Score every vector: $O(nd)$ and exact. With 8-bit scalar quantization (one scale per vector,

$s = \max |x|/127$) a stored coordinate is off by at most $s/2$, so for a unit query a score is off by at most $(s/2)\sqrt{d}$, at a quarter of the memory.

IVF (inverted file [36]). Cluster the vectors into L lists with k -means [41]. A query scores the L centroids, then only the p nearest lists: cost $Ld + p(n/L)d$, minimal at $L = \sqrt{pn}$, where it equals $2d\sqrt{pn}$. OntoDB stores each list as a range of a B+ tree: key (list, row, 0), so a list is a prefix scan and an insert is a tree insert.

Product quantization. Cut a vector into m pieces and replace each by the index of its nearest of 256 learned centers: m bytes per vector. A query builds m tables of 256 dot products ($256d$ work), and a stored code scores as the sum of m table entries. That equals the dot product with the decoded vector exactly, so all the error is in the stored side.

HNSW (hierarchical navigable small world [37]). A layered proximity graph searched greedily: memory $O(nM)$, the slowest to build, usually the best recall for a given time, and no worst-case bound.

Design decision 65: IVF lists are B+ tree ranges

Choice. A posting list is the set of keys (list, row, 0) in a B+ tree of the same kind as the triple indexes.

Alternatives. A separate array per list in memory, as FAISS keeps them [42].

Why. The tree already pages, caches, and latches; an insert or delete is a tree operation; a list is a prefix scan, the access path OntoDB is built around.

Cost. The vectors themselves are still fetched by row from the vector store, one random access per candidate.

11.5 Filtering a vector search by a class

A hybrid query asks for the k terms nearest t that lie under class C . Let $s = |\text{subtree}(C)|/n$.

Pre-filter. Resolve C to its members first (one range scan per interval), then score only those, exactly: about snd work. Best when s is small.

Post-filter. Ask the index for more than k , keep those under C (a membership test is a binary search over C 's intervals; no scan), and ask for four times as many until k remain. If the filter were independent of the similarity one would need about k/s candidates, so it fails when s is small.

In-index filter. Resolve C , then give the index the set as a mask: IVF skips list entries outside it, HNSW still crosses masked nodes to stay connected but never returns them [38].

In an ontology the filter and the vectors are *not* independent: the names of a class's descendants are close to the class and to each other. A query from inside C finds most of its neighbors inside C , so post-filtering works far better than k/s predicts; a query from outside finds almost none, and it works far worse. Section 11.7 measures both.

11.6 Why not an ART with sibling links

An adaptive radix tree [40] orders keys by byte, so its range scan is a *lexical* range scan: every label that starts with *rub*. That is a real query, and the ART is good at it, but it is the same one-dimensional order as the B+ tree, and lemma 11.1 applies unchanged.

Sibling links on an ART's leaves are possible. Path compression and node growth never move a leaf. Lazy expansion does: an insert that meets a leaf turns it into an inner node and puts a *copy* of the leaf under it, so a link that pointed at the old object now points at an inner node. Every insert must also find its sorted neighbors, a partial descent, and concurrent inserts must latch both. A B+ tree has the link for free: a leaf is a page, and a split is local. The usual alternative is a stack-based iterator: no link, climb and descend a stack of [node, keys, next child], which pays pointer chasing between leaves and costs nothing on insert. OntoDB's keys are three fixed-width integers, which gives an ART little prefix to compress, so the B+ tree stays.

Pessimistic and optimistic prefixes. Path compression stores a node's shared prefix. *Pessimistic* storage keeps the bytes in the node, so a descent compares them and fails at the first wrong byte; the cost is variable-length storage. *Optimistic* storage keeps only the length: the descent skips those bytes and the full key is checked once, at the leaf. Nodes are smaller, and a wrong path is found late. For a range scan the difference is in the seek. To find the first key at or above *lo* the seek must know whether a whole subtree lies below or above *lo*. With stored bytes it decides at the node; with skipped bytes it must load a key from the subtree and compare. The upper bound needs no jumping: a second seek finds the first key past *hi*. The only binary search in a scan is over the sorted edge bytes of one node.

11.7 What it costs

These measurements are on the Gene Ontology (release 2026-07-26, without obsolete terms: 38 092 classes, 57 456 *is_a* edges), with the term names embedded by a 22-million-parameter sentence model on a CPU. LUBM and a set of synthetic instances give the type queries something to run on. Everything is one thread on an Apple M2 Pro, and benchmarks/semantic/run.sh reproduces it; the full tables are in benchmarks/semantic/results.md.

Does an order put neighbors together? For 2 000 random terms, the ten nearest by cosine; table 11.1 gives the share within $\pm W$ positions.

Order	median gap	± 100	± 1000	± 5000
dictionary id (first seen)	1739	.211	.284	.438
hierarchy postorder	45	.377	.561	.735
label, alphabetical	1749	.334	.408	.558
random	10823	.005	.049	.242

Table 11.1: Share of the ten nearest neighbors found within $\pm W$ positions.

The postorder of the hierarchy is the best order, and far better than first-seen ids: the nearest

neighbor is a median of 45 positions away instead of 1 739. It still finds only 38% of the ten within 100 positions, and 56% within 1 000, as lemma 11.1 says to expect. First-seen ids are not random either: the Gene Ontology minted its ids in related batches, so id order carries locality that belongs to the file, not to the engine. A term's neighbor is a parent or child in 12% of cases and within two is_a steps in 32%, in a two-step neighborhood of 45 terms (0.12% of all terms): the hierarchy is informative about meaning without being sufficient.

Intervals. The Gene Ontology needs 76 570 intervals for 38 092 classes: 2.01 per class on average, 598 KiB in all, at most 741 for one class, inside the bound of lemma 11.4 (19 368). Materializing the subclass closure would store 393 448 ancestor pairs. Whether d is below c takes 27 ns, 29 times faster than a search up the parents. The descendants of a class with 1 000 or more subclasses take 101 range scans and 40 page reads, against 3 304 scans and 694 pages for one scan per class.

Three ways to answer ?x a c. On 200 000 synthetic entities over the Gene Ontology's classes (495 552 triples):

Strategy	triples	file (MiB)	load (s)
rewrite	495 552	63.6	5.1
intervals	495 552	72.9	5.3
materialize	4 017 973	309.5	180.3

Materializing stores 8.1 times the triples and takes 36 times as long to load. For a class with 1 000 or more subclasses a query reads 265 pages with intervals, 182 with the materialized store, and 12 803 by rewriting (48 times more than intervals). All three return the same answers on every class sampled. On LUBM(1), whose hierarchy has 43 classes and little depth, the three are within a factor of about two; intervals have little to save where the hierarchy is shallow.

Design decision 66: Intervals instead of a materialized closure for type queries

Choice. A query for the instances of a class is one range scan per interval of the class, over an index of (label of type, instance).

Alternatives. Materialize the entailed type triples (chapter 10); rewrite into one scan per subclass.

Why. On a deep, wide hierarchy the interval index costs 15% more space instead of 4.9 times as much, loads in 5 seconds instead of 180, and reads within 1.8 times of the materialized store's pages.

Cost. The labels are rebuilt when the hierarchy changes, and only subClassOf is understood: no subPropertyOf, domain, or range. A materialized closure handles those rules with the same machinery and an interval index does not.

Vector indexes. 38 092 vectors of 384 floats; recall@10 against the exact answer, time per query on one thread.

Index	setting	recall	μs	MiB
flat float32	exact	1.000	2 238	55.8
flat PQ $m=48$	codes only	0.778	819	1.7
IVF $L=512$	nprobe 16	0.929	255	56.5
IVF $L=512$	nprobe 64	0.976	928	56.5
HNSW $M=16$	ef 10	0.993	44	59.7
HNSW $M=16$	ef 40	0.997	108	59.7

HNSW dominates on this data: 0.993 recall at 44 μs , 51 times faster than the exact scan. A word on that baseline: the first run scored with a plain dot-product loop, which a compiler may not vectorize without `-ffast-math`, and the exact scan took 8.3 ms. With eight independent accumulators it takes 2.2 ms, and a speed-up over “exact” is only as meaningful as the exact baseline is fast. IVF lists stored in the B+ tree cost about 2.7 times as much per scored row as a flat scan (157 ns against 59 ns): a decoded key and a random read of the vector, the price of lists that page and update like the triples. Product quantization is for memory; re-ranking its best 100 exactly gives 0.998 recall, but needs the floats.

The ten nearest terms under a class. Table 11.2 gives time and recall for five subtree sizes, with a query from inside the class.

Members of C	≤ 10	11–100	101–1000	1001–5000	>5000
pre-filter	3	9	62	292	1 404
post-filter, HNSW	37 553	1 739	427	70	68
masked HNSW	16 653	15 846	1 150	183	105
masked IVF	1 253	111	104	124	161
auto	3	9	59	269	95

Table 11.2: Microseconds per query, query inside C . Pre-filter, post-filtered and masked HNSW are exact at these sizes; masked IVF has recall 0.95 to 1.00.

Pre-filtering through the interval index is exact and the fastest up to about a thousand members, by factors from 1.7 to more than 10 000. The filter is correlated with the vectors: of the unfiltered top 100 neighbors of a term inside C , 4% to 72% lie under C , and for a term outside it 0% to 12%. The independence estimate k/s is therefore wrong in both directions, and post-filtering IVF recalls 0.07 of the right answers on the smallest classes for a query outside them. A masked HNSW search wins above 5 000 members (105 μs inside and 900 μs outside, exact). One more lesson: resolving the members into a bitmap before a masked search cost 560 μs of a 669 μs query, while testing each touched row’s label against the intervals, one binary search, cost nothing extra and brought it to 105 μs . kAuto takes the exact path up to 4 096 members and a masked HNSW search above, and is within a factor of about two of the best strategy everywhere but one cell.

Design decision 67: Resolve small classes, mask large ones

Choice. Up to 4 096 members: resolve the members by range scans and score them exactly. Above: search HNSW with a predicate, “the row’s label lies in an interval of C ”.

Alternatives. Always post-filter; always mask; always resolve.

Why. Resolving is exact and fastest while a class is small. Post-filtering fails for a query

outside a small class. The threshold sits below the point where an outside query makes masking slower than resolving (7 000 to 25 000 members) and loses at most a factor of 3.8 where an inside query makes masking faster (about 1 000).

Cost. One threshold for both kinds of query; a class of 1 001 to 5 000 members queried from inside runs 3.8 times slower than it could.

Status

Built (plan 14.1–14.9): class labels, the interval indexes, the vector store, flat, IVF, and HNSW indexes, hybrid search, the shell commands (`\hindex`, `\under`, `\vload`, `\vindex`, `\similar`), and the benchmarks of this chapter (`benchmarks/semantic/`). Vectors live in memory, not in buffer pool pages, and the structures are not updated by inserts: they are rebuilt.

Check your understanding

- 11.1 Why does a sort order of dictionary ids not put similar classes next to each other? What would have to be true of the input file for it to?
- 11.2 Use lemma 11.1 to bound how far apart two neighbors must be in the best order of a 10×10 grid.
- 11.3 Give the postorder labels and the interval sets of the diamond $D \sqsubseteq B, C \sqsubseteq A$ for a spanning forest that keeps D under B .
- 11.4 A class has 5 non-tree edges below it. How many intervals can it have at most, and why?
- 11.5 When is pre-filtering better than post-filtering for a class filter, and when is it not?
- 11.6 Why can post-filtering work better than the independence estimate k/s says, in an ontology?
- 11.7 What does a seek for a lower bound have to do differently if prefixes are stored optimistically?

Answers are in appendix B.

Further reading

Agrawal, Borgida, and Jagadish [34] on interval labelling of transitive relationships; Jégou, Douze, and Schmid [36] on inverted files and product quantization; Malkov and Yashunin [37] on HNSW; Patel et al. [38] on filtered vector search; Pan, Wang, and Li [39] for a survey of vector database systems; Leis, Kemper, and Neumann [40] on the adaptive radix tree.

Part IV

Transactions

Transactions and Isolation

A *transaction* is a group of reads and writes that the database treats as one unit. Transactions let an application reason about one operation at a time, even though many run concurrently and any of them may be interrupted by a crash.

12.1 ACID

Atomicity. All of a transaction's writes happen, or none do. A transaction that aborts, or is cut off by a crash, leaves no trace. (Provided by the log: chapters 14 and 15.)

Consistency. A transaction takes the database from one valid state to another, where "valid" is defined by the application's invariants. The engine helps with constraints, but consistency is mostly the application's promise.

Isolation. Concurrent transactions do not see each other's intermediate states; the result is as if they ran in some order. (Provided by concurrency control: chapter 13.)

Durability. Once a commit is acknowledged, its effects survive any later crash. (Provided by the log and fsync.)

12.2 Schedules and serializability

A *schedule* is the interleaved order in which the operations of concurrent transactions actually execute. A schedule is *serializable* if its effect equals that of *some* serial order of the same transactions. Serializability is the gold standard of isolation: the application cannot tell that concurrency happened.

Two operations *conflict* if they come from different transactions, touch the same item, and at least one is a write. A schedule is *conflict-serializable* if it can be turned into a serial one by swapping adjacent non-conflicting operations. The test is a graph: draw an edge $T_i \rightarrow T_j$ for every pair of conflicting operations where T_i 's comes first. The schedule is conflict-serializable exactly when the graph has no cycle.

12.3 Anomalies

Weaker isolation admits specific bad interleavings. Naming them is the clearest way to define the isolation levels [19].

Dirty read. T_2 reads a value T_1 wrote but has not committed. If T_1 aborts, T_2 read data that never existed.

Non-repeatable read. T_1 reads an item, T_2 changes it and commits, and T_1 reads it again and sees a different value.

Phantom. T_1 reads the set of items matching a condition (“every student”), T_2 inserts a new matching item and commits, and T_1 repeats the query and sees an extra row.

Lost update. Both transactions read a counter, both add one, both write: one increment disappears.

12.4 Isolation levels

Level	Dirty read	Non-repeatable read	Phantom
Read uncommitted	possible	possible	possible
Read committed	—	possible	possible
Repeatable read	—	—	possible
Serializable	—	—	—

Table 12.1: The SQL standard’s levels, defined by the anomalies they forbid.

Berenson et al. [19] showed that defining levels by these phenomena alone is ambiguous, and that real systems’ levels (notably snapshot isolation, which forbids all three but admits a “write skew”) do not fit the table cleanly. The table remains the right first mental model.

Key idea

Isolation is a dial, not a switch. Lower levels allow more concurrency and specific anomalies; serializable allows none. Each level is defined by what it forbids, and each forbidden anomaly corresponds to a specific lock held for a specific time (chapter 13).

In OntoDB: transactions

Every shell statement is its own transaction unless `\begin` opens an explicit one, ended by `\commit` or `\abort`. Isolation is a setting, `\set isolation ru|rc|rr|ser`, defaulting to read committed. The isolation contract is written as executable schedules: twelve specs, each a scripted interleaving of two sessions with expectations such as “session 2 is blocked”, “the read returns Dana”, or “session 2 is aborted”. They cover dirty reads, non-repeatable reads, lost updates, lock upgrades, two deadlock shapes, write–write conflicts, locks released on abort, and phantoms.

In a persistent database today, each update statement (or `\load`) is one atomic, durable transaction (chapter 16).

Design decision 68: Every statement is a transaction

Choice. A statement outside `\begin` runs in its own implicit transaction, which takes locks like any other and commits when the statement ends; in a persistent database it is also one durable log transaction. `\load` takes an exclusive lock on the whole database for its length rather than one lock per triple.

Alternatives. Require an explicit `BEGIN`; or apply statements immediately with no atomicity.

Why. An ontology edit is usually one statement, and a load should be all or nothing: a half-loaded OWL file is a corrupt ontology.

Cost. A very large `\load` is one very large transaction, logged and held in memory until its commit.

Design decision 69: A failed statement takes itself back

Choice. `INSERT DATA` and `DELETE DATA` run as one operator that remembers the triples it has changed. If a later write throws, it undoes those changes, newest first, before the error reaches the caller. The store, for its part, writes each triple to every index or none.

Alternatives. Leave a half-applied statement in place and rely on the caller; or rely on the log to undo it.

Why. A statement should be atomic on both backends, including the temporary indexed one, which has no log. Undo in memory is enough for an exception; the log is still needed for a crash.

Cost. The undo list lives only in memory: if the process dies in the middle of a statement, only the log can take it back (chapter 15), and the memory backend without a directory keeps nothing anyway.

Design decision 70: Read committed by default

Choice. `\set isolation defaults to rc`.

Alternatives. Serializable, or repeatable read (MySQL's InnoDB default).

Why. It matches PostgreSQL's default, forbids dirty reads, and releases shared locks at the end of each statement, so readers block writers only briefly.

Cost. Non-repeatable reads and phantoms are possible unless a session raises its level.

Design decision 71: Shared locks are kept to commit, even at read committed

Choice. Every level that takes shared locks (all but read uncommitted) keeps them until the transaction ends. The levels differ in what else is locked:

Level	Locks a scan takes (exclusive locks: always, to commit)
ru	none
rc	S on each triple read
rr	S on each triple read, and S on the set of triples
ser	S on the whole database for each query

Alternatives. The textbook rule, which releases read committed's shared locks at the end of each statement.

Why. The specs are the contract, and two of them, lock-upgrade and upgrade-deadlock, run at read committed and expect a key read by an earlier statement to block a later delete. Isolation levels are minimums, so keeping locks longer is allowed; read committed still

forbids dirty reads and still lets new triples appear (nonrepeatable-rc).

Cost. Read committed holds more locks for longer than most engines' read committed, so read-then-write transactions conflict, and deadlock, more often.

Design decision 72: Isolation is specified as executable schedules

Choice. The isolation contract is a set of spec files, each a scripted interleaving of two sessions with expected outcomes (blocked, returns a value, aborted).

Alternatives. Prose definitions, or randomized stress tests.

Why. Each anomaly becomes one small, deterministic scenario that either happens or does not. A stress test can pass by luck; a schedule cannot.

Cost. Only the interleavings someone wrote down are checked.

Status

Transactions and isolation run (plans 7.2 to 7.6): explicit transactions with `\begin`, `\commit`, and `\abort`; the four isolation levels; and all twelve specs, on both backends. A persistent database runs each statement as its own transaction and does not accept `\begin` yet.

Check your understanding

- 12.1 Draw the conflict graph of $r_1(A) w_2(A) r_2(B) w_1(B)$. Is the schedule conflict-serializable?
- 12.2 Give a schedule with a non-repeatable read, and say which isolation level first forbids it.
- 12.3 Why is consistency mostly the application's promise and not the engine's?
- 12.4 `INSERT DATA` names three triples, and the store throws on the third. What is in the store afterwards, on the memory backend without a directory?
- 12.5 Two sessions both run "read the count of Students, insert one more". Which anomaly can occur, and at which levels?

Answers are in appendix B.

Further reading

Gray and Reuter [3]; Berenson et al. [19]; Petrov [1], chapter 5.

Concurrency Control

Isolation levels say *what* must hold; concurrency control is *how*. This chapter covers locking, the approach OntoDB takes: two-phase locking, multi-granularity locks, deadlock handling, phantoms, and the separate problem of protecting the B+ tree's own structure.

13.1 Two-phase locking

A transaction takes a *shared* (S) lock to read an item and an *exclusive* (X) lock to write it. S is compatible with S; X is compatible with nothing. *Two-phase locking* (2PL) adds one rule: a transaction may not acquire any lock after it has released one. Its life has a growing phase and a shrinking phase.

2PL guarantees conflict-serializability: the order in which transactions reach their *lock point* (the moment they hold all their locks) is an equivalent serial order.

Plain 2PL has a flaw. If T_1 releases an X lock before committing and T_2 reads the value, then T_1 aborts, T_2 has read a value that never existed, and must abort too: a *cascading abort*. *Strict 2PL* holds every X lock until commit or abort, which rules this out.

Key idea

Exclusive locks are held to the end of the transaction at every isolation level, because releasing one early lets others see a write that may still be undone. The isolation levels differ only in how long *shared* locks are held, and whether ranges are locked.

Level	Shared locks
Read uncommitted	none
Read committed	released at the end of each statement
Repeatable read	held until commit
Serializable	held until commit, plus range or predicate locks against phantoms

Table 13.1: Isolation levels as locking rules, given strict 2PL for writes.

13.2 Locking at several granularities

Locking every triple is precise but expensive for a query that reads the whole store; locking the whole store is cheap but serializes everything. Gray et al. [20] let a transaction choose its granularity, using *intention* locks on the coarser levels: before locking a key in S, take IS (“intends to share something below”) on the index and on the database.

Held \ Requested	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	—
IX	yes	yes	—	—	—
S	yes	—	yes	—	—
SIX	yes	—	—	—	—
X	—	—	—	—	—

Table 13.2: Multi-granularity lock compatibility. SIX is S on the whole node plus the intention to write below it.

The point of intention locks: a transaction that wants to lock a whole index in S can check one lock on the index, instead of every key below it, because any writer below must already hold IX there.

13.3 Deadlock

T_1 holds a lock T_2 wants, and T_2 holds a lock T_1 wants: both wait forever. Engines either *prevent* deadlock (order locks, or abort on conflicts by timestamp, as in wait-die and wound-wait) or *detect* it: maintain a *waits-for graph* with an edge $T_i \rightarrow T_j$ when T_i waits for a lock T_j holds, find a cycle, and abort one transaction in it, the *victim*. Choosing the youngest transaction (the one that has done the least work) is a common and deterministic policy.

13.4 Phantoms

Repeatable read locks the triples a transaction read. A phantom is a triple it did *not* read, because it did not exist yet; no lock covers it. Preventing phantoms needs a lock on what *could* match, not on what does:

Predicate locks lock the query’s condition itself. Precise, but testing whether two predicates overlap is expensive in general [21].

Key-range (next-key) locks lock the gap between index keys, so an insert into the scanned range must wait. Practical, and natural on a B+ tree.

13.5 Latches are not locks

A B+ tree split rewrites several pages at once. Even a single-transaction engine with many threads must stop two threads from splitting the same page at the same time. That protection is a *latch*: a short-term reader-writer mutex on a page, held for the duration of one tree operation, never for a transaction. Locks protect *logical contents* for transactional isolation; latches protect *physical structure* for thread safety.

The classic latch protocol for B+ trees is *crabbing*: latch the child, and release the parent if the child is *safe*, meaning it cannot split (for an insert) or merge (for a delete) as a result of this

operation. A thread therefore holds latches only on the part of the path a structure change could reach. An *optimistic* variant takes shared latches down to the leaf, latches only the leaf exclusively, and restarts pessimistically if the leaf turns out to need a split. Lehman and Yao's B-link tree [7] goes further and lets readers run without latching the path at all, by following right-links when a split has moved their key.

13.6 The alternative: multiversion concurrency

Locking makes readers and writers wait for each other. *Multiversion concurrency control* (MVCC) keeps old versions of items, so a reader sees a consistent snapshot as of its start time and never waits for a writer. Writers still conflict with writers. The cost is storing old versions and cleaning them up once no reader can need them. Most modern systems use MVCC with snapshot isolation by default.

In OntoDB: concurrency control

OntoDB uses strict 2PL with five lock modes on three granularities: the database, an index ("spo", "pos", "osp"), and a key, with FIFO grant order and in-place upgrades that do not jump ahead of waiters. A blocked shell session prints why it is waiting, for example session 2 waiting: X lock held by txn 5. A background thread builds the waits-for graph and aborts the *youngest* transaction in each cycle (the larger id), so deadlock outcomes are deterministic and testable. Serializable adds one phantom mechanism, chosen and documented in the code. B+ tree operations use latch crabbing, and the whole feature runs under ThreadSanitizer. MVCC is a deliberate future comparison, not part of the plan.

Design decision 73: Locking, not multiversioning

Choice. Strict two-phase locking with multi-granularity locks.

Alternatives. MVCC with snapshot isolation, as PostgreSQL and most modern engines use; or optimistic concurrency control.

Why. Under locking, each anomaly is prevented by one specific lock held for one specific time, which makes isolation levels concrete and testable. It also pairs directly with ARIES, which was designed for it.

Cost. Readers and writers wait for each other. MVCC is planned as a measured comparison: what a reader no longer waits for, and what keeping old versions costs.

Design decision 74: Detect deadlocks and abort the youngest

Choice. A background thread wakes every 20 ms, builds the waits-for graph (an edge from each waiting request to every transaction holding an incompatible lock, or waiting ahead of it in the same queue), and aborts the transaction with the largest id in each cycle. The victim's Lock call throws; its owner then undoes it and releases its locks.

Alternatives. Prevention by wait-die or wound-wait, or lock-wait timeouts.

Why. Detection aborts only when a cycle really exists. Choosing the youngest is deterministic, so a deadlock spec can name the victim, and it usually discards the least work.

Cost. A detector thread, and transactions sit blocked until it runs.

Design decision 75: Upgrades wait their turn

Choice. Each resource has a FIFO queue. A transaction holding S that asks for X keeps its S lock and queues behind any request already waiting; it never jumps ahead.

Alternatives. Let upgrades go to the front of the queue, as many lock managers do.

Why. FIFO grants are easy to predict and test, and no request can be starved by a stream of upgrades.

Cost. Two transactions that both read a key and then both try to change it deadlock: each waits for the other's S lock. The detector breaks the cycle by aborting the younger, which is exactly what the upgrade-deadlock spec expects. Jumping the queue would not avoid that deadlock either; the usual cure is to take the stronger lock when reading ("select for update").

Design decision 76: Phantoms: a lock on the set at repeatable read, on everything at serializable

Choice. A repeatable-read scan takes S on the set of triples, so no insert can enter what it scanned until it commits. A serializable query takes S on the whole database: a predicate lock whose predicate is "every triple", which blocks every writer, including an insert into a range whose scan found nothing or never ran.

Alternatives. Key-range or gap locks on the index, which block only inserts into the scanned range; or true predicate locks on each pattern.

Why. Locks name triples, not index entries (next decision), so there is no index gap to lock; the coarse locks are simple and obviously correct. A pattern that names a term the store has never seen matches nothing and its scan does not run, which is why repeatable read lets the phantom-rr insert through while serializable still blocks phantom-ser's.

Cost. Concurrency, badly: one serializable reader stops every writer, and one repeatable-read scan stops every insert. Gap locks are the natural refinement once locks are taken per index.

Design decision 77: Locks name triples, not index entries

Choice. The lock hierarchy is the database, the set of triples, and one triple keyed by its SPO key, whichever index a scan used and however many indexes store the triple.

Alternatives. Lock each index's entries separately, as a relational engine locks index keys.

Why. A triple is one logical item stored three times; locking it once means a writer and a reader using different permutations still meet on the same lock. The same hierarchy serves the memory store, which has no indexes.

Cost. No range locks are possible on a resource with no order, so phantoms are handled coarsely (previous decision).

Design decision 78: Pessimistic crabbing, and emptied leaves stay

Choice. Readers latch each child before releasing its parent. Writers latch the header and the path in write mode and release every latched ancestor as soon as the newest page is safe: not full, for an insert; above its minimum, for a delete. A split writes its two leaves and releases them before allocating the parent's new page. A leaf emptied by a merge is not freed: it keeps its sibling link and stays as an empty page.

Alternatives. Optimistic crabbing (read-latch to the leaf, restart if the leaf must split); B-link trees, whose right links let readers skip latching altogether; or one latch for the whole tree.

Why. Pessimistic crabbing is the simplest protocol that lets operations on different parts of the tree run at once, and it reuses the single-threaded split and merge code with latched pages in place of re-read ones. An iterator moves right after releasing its leaf, so it may hold the id of a leaf that a merge has just emptied; keeping that page readable, with a link onward, means the iterator simply steps through it.

Cost. A writer on a full path latches and pins the whole path, which serializes inserts near the root and needs a pool larger than the height plus two. Emptied leaves are never reclaimed. A test with 3-key nodes, four writers, and a concurrent scanner checks every key and the invariants under ThreadSanitizer.

Status

Concurrency control is complete (plans 7.1 to 7.6): latch crabbing, the transaction and lock managers, the four isolation levels, deadlock detection, and phantom protection. The twelve isolation specs pass on both backends, and the concurrency tests run under ThreadSanitizer. The isolation harness already exists and checks itself.

Check your understanding

- 13.1 Why does plain 2PL allow cascading aborts, and how does strict 2PL prevent them?
- 13.2 A transaction wants to read every triple in the "spo" index. Which locks does it take, and why is checking one lock enough to know no writer is below?
- 13.3 T_3 waits for T_5 , and T_5 waits for T_3 . Which one does OntoDB abort?
- 13.4 What is the difference between a latch and a lock? Give one thing each protects.
- 13.5 In crabbing, when is a child "safe" for an insert? For a delete?

Answers are in appendix B.

Further reading

Gray et al. [20]; Eswaran et al. [21]; Lehman and Yao [7]; Bayer and Schkolnick [8]; Petrov [1], chapters 5 and 6.

Part V

Recovery

Logging and the Write-Ahead Rule

The buffer pool steals and does not force (chapter 3). So at the moment of a crash, the disk can hold changes from transactions that never committed, and can lack changes from transactions that did. The *log* is what lets the engine repair both. This chapter explains what goes in it, and the one rule that makes it trustworthy.

14.1 The failure model

Transaction failure. A transaction aborts: a deadlock victim, a constraint violation, or the user's choice. Its effects must be removed.

System failure. The process or machine stops. Memory is lost; the disk survives. Committed work must reappear; uncommitted work must vanish.

Media failure. The disk is lost. Only a backup or a replica helps; it is outside the scope of the log alone.

The log handles the first two. It is an append-only file of *log records*, each describing one change, written sequentially, which is the access pattern storage handles best.

14.2 Undo and redo

An update record carries enough to apply the change again (*redo*) and to reverse it (*undo*). For a page-oriented engine that might be the bytes before and after; for a triple store it can be logical: "insert triple (5, 7, 9)", whose undo is "delete (5, 7, 9)".

Every record has a *log sequence number* (LSN), increasing, unique, and often simply the record's byte offset in the log. Records of one transaction are chained by a *prevLSN* field, so the engine can walk a transaction's history backward when it must undo it.

Every page stores a *pageLSN*: the LSN of the last logged change applied to it. The pageLSN is how recovery knows whether a logged change already reached the page on disk: if $\text{pageLSN} \geq \text{record's LSN}$, it did.

14.3 The write-ahead rule

Key idea

Write-ahead logging (WAL). (1) A page may be written to disk only after every log record describing a change to it is durable: flush the log up to the page's pageLSN first. (2) A transaction commits only when its commit record, and therefore all its earlier records, are durable.

Rule 1 makes steal safe: if an uncommitted change reaches the disk, its undo information is already in the log. Rule 2 makes no-force safe: if a committed change has not reached the disk, its redo information is already in the log. Between them, every possible state of the disk after a crash can be repaired.

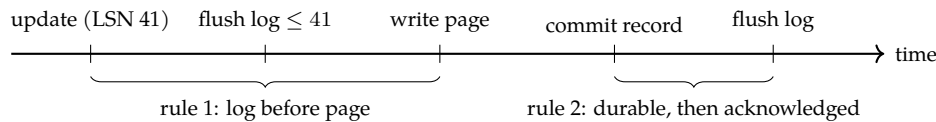


Figure 14.1: The two orderings the write-ahead rule imposes.

14.4 Making commits cheap: group commit

Rule 2 means every commit waits for an fsync of the log. If each commit issued its own fsync, throughput would be bounded by the device's fsync rate. *Group commit* lets commits that arrive while a flush is in progress share the next one:

```
function Commit(txn):
    lsn = Append(commit record for txn)      // into the in-memory log buffer
    wait until persistentLSN >= lsn          // a background flusher wakes us

flusher loop:
    wait until someone asked, or a short interval passes
    batch = swap out the log buffer
    write(batch); fsync()                    // one fsync for many commits
    persistentLSN = last LSN in batch; wake all waiters
```

Under concurrency, the number of fsync calls grows much more slowly than the number of commits. A single client still pays the full latency of one flush.

14.5 When a record is damaged

A crash can interrupt the last write to the log, leaving a *torn* record: fewer bytes than its header promises. Bit rot or a misbehaving device can damage a record whose bytes are all present. Robust logs detect both with two fields in every record header: its **length**, so a short record is recognized, and a **checksum** over its bytes, so a damaged one is. PostgreSQL, RocksDB, and

InnoDB all put a CRC on every log record. A recovery procedure then has a policy choice, discussed in chapter 16: stop at the first bad record and keep everything before it, or refuse to start.

In OntoDB: log records and the log manager

Every OntoDB log record has this layout (native byte order):

```

u32 size      whole record, header included
u32 crc32c    over every byte after this field
u8  type      begin | commit | abort | update | clr | checkpoint | term | page
i64 lsn, i64 txn_id, i64 prev_lsn
then, by type:
  update      i32 page_id, u8 is_insert, u64 subject, u64 predicate, u64 object
  clr         the update fields, then i64 undo_next
  checkpoint  u32 n, then n active transaction ids
  term        u64 term_id, u32 n, then n bytes of canonical spelling
  page        i32 page_id, then the page's bytes after its pageLSN

```

The checksum is CRC-32C, the same checksum PostgreSQL, RocksDB, and InnoDB use. A short buffer raises an ordinary storage error (a torn write), while a record whose bytes are present but wrong raises a distinct corruption error, so the two cases can be handled differently. LSNs start at 1; reopening a log continues after its last valid record.

The log manager appends records to an in-memory buffer and a background thread performs group commit; a test runs four threads committing twenty times each and requires fewer fsyncs than commits. A failed fsync is reported, never retried, for the reason in chapter 2.

Design decision 79: Length and CRC-32C in every record

Choice. Each record starts with its size and a CRC-32C over the rest. A short record raises a storage error (a torn write); a full-length record with a wrong checksum raises a distinct corruption error.

Alternatives. No checksum; a plain CRC-32; a faster non-cryptographic hash such as xxHash.

Why. A crash leaves a torn tail, and the two cases deserve different handling. CRC-32C has hardware instructions on x86 and ARM and is what PostgreSQL, RocksDB, and InnoDB use.

Cost. Eight bytes of header per record, and a checksum per append and per read.

Design decision 80: LSNs are counters, not byte offsets

Choice. An LSN is a 64-bit counter starting at 1. On open, the next LSN is one past the larger of the snapshot's LSN and the last valid record.

Alternatives. Use the record's byte offset in the log, as ARIES and PostgreSQL do, so an LSN locates its record directly.

Why. A checkpoint of the memory database empties the log file. A counter keeps increasing across that truncation, so "skip records at or below the snapshot's LSN" stays correct;

an offset would restart at zero.

Cost. Finding a record by LSN needs a scan, not a seek.

Design decision 81: A failed fsync is fatal

Choice. When a log flush fails, the error is reported and the commit is not acknowledged. The flush is never retried.

Alternatives. Retry until it succeeds.

Why. After a failed fsync, the kernel may have dropped the dirty pages and cleared the error, so a retry that succeeds can still have lost the data (chapter 2).

Cost. A transient device error stops the database instead of being ridden out.

14.6 What a page record holds

Rule 1 needs every change to a page to be logged before the page can be written, and redo needs enough in each record to put the change back. There are three common answers to “what does the record hold”. *Physical* logging records the bytes: a whole page image, or a byte range and its new contents. *Logical* logging records the operation (“insert key k into the SPO index”), and redo performs it again. *Physiological* logging, which ARIES and most disk engines use, is a mix: the record names one page (physical) and describes an operation on that page (logical), such as “insert this key at slot 7 of page 12”.

Each choice moves work around. Page images are the simplest to redo (copy the bytes) and need no knowledge of what is on the page, but they are large. Logical records are small, but redo must run the operation against a structure that may itself be damaged, which is why engines that log logically above the page level still log pages physically below it.

14.7 Changes that span pages

A B+ tree split changes three pages: the old leaf loses its upper half, a new leaf receives it, and the parent gains a separator. Each page gets its own record. Suppose the log reaches the disk with the records for the two leaves but not the parent’s, and then the machine stops. Redo faithfully rebuilds both leaves, and the parent still has no pointer to the new one. A full scan, which walks the leaf chain, finds every key; a point lookup for a key in the new leaf descends to the old leaf and misses. Nothing in the log is wrong, yet the tree is.

The write-ahead rule is about single pages, so it cannot prevent this. A multi-page change needs its own rule: its records must reach the log all together or not at all. Systems get there in different ways:

Nested top actions (ARIES). Log the split’s page changes with undo information, then a dummy CLR that marks the split complete. If the crash comes before the dummy CLR, undo reverses the half-done split page by page.

Mini-transactions (InnoDB). Hold every latch the change took until it is finished, then append all its records to the log buffer in one step.

Tolerate the half state (PostgreSQL). B-link trees give every page a right link and a high key, so a search that lands on the old leaf can follow the right link. A split missing its parent pointer is legal, and the next writer that notices finishes it.

In OntoDB: page images and page groups

OntoDB logs physically, by whole page. When a write guard on a page it changed is released, still latched, the buffer pool appends a page record holding the page's bytes after the pageLSN and sets the pageLSN to that record's LSN (`BufferPoolManager::LogPageWrite`). Every page written through a guard is covered by this one path: B+ tree nodes, tree headers, and the catalog. Triple changes get a logical update record too, appended before the statement changes any page; undo uses those (chapter 15).

A tree Insert or Remove opens a PageGroup before its first latch and closes it after its last. While a group is open, the log manager writes nothing from the group's first LSN onward, so the flusher can cut the log only between groups. The buffer pool, choosing a victim, passes over any dirty page whose pageLSN is still held back, since writing it would break rule 1. A test stops a tree at each named crash point inside a split or a merge, flushes everything the log allows, copies the files, and checks that redo yields a sound tree that holds either every key from before the operation or every key from after it. Without page groups, all seven cases fail.

Design decision 82: Whole page images, not physiological records

Choice. A page record carries the whole page after its pageLSN. Redo copies it when the record is newer than the page.

Alternatives. Physiological records, "insert key k at slot i of page p ", as ARIES and InnoDB use; byte-range diffs.

Why. Redo needs to know nothing about the layout of a leaf, an internal node, a header, or the catalog, so one short loop redoes them all, and a bug in a page's code cannot also be a bug in its redo. The guard already knows when a page changed; it does not know how.

Cost. About 4 KB per page per operation. One triple touches at least three trees, so it logs at least three pages, where its logical update record is a few dozen bytes. A real engine would log diffs, or log a full image only the first time a page changes after a checkpoint, as PostgreSQL's full-page writes do.

Design decision 83: Page groups hold back the log, not the latches

Choice. A group marks its first LSN. The log does not flush past it, and the pool does not evict a dirty page the group holds back, until the group ends.

Alternatives. Nested top actions with before-images and undo of an unfinished split; InnoDB's mini-transaction, which keeps every page latched until the end; B-link trees that accept a half split.

Why. No before-images, no recovery pass for unfinished splits, and no change to latching.

Holding latches longer would have broken the tree's rule that siblings are latched left to right, since a merge releases a page to latch its left sibling.

Cost. While any group is open, every later record waits too, so a commit that arrives during a split waits for it. A group's pages cannot be evicted, so the pool must hold more frames than the largest split touches. A pool so small that every unpinned frame is held back cannot find a victim.

Status

Log records (plan 8.1), the log manager with group commit (plan 8.2), and the write-ahead rule with pageLSNs set on every change (plan 8.3) run. The rule applies wherever a buffer pool is given a log manager: the recovery tests and the crash harness. The shell's indexed backend is a temporary file and runs without a log, and `ontodb <dir>` opens the memory database of chapter 16.

Check your understanding

- 14.1 State the two write-ahead rules, and say which buffer-pool policy each one makes safe.
- 14.2 A page has pageLSN 40. Recovery reads an update record for it with LSN 35. Redo it or skip it?
- 14.3 Ten sessions commit within one millisecond. Without group commit, how many fsyncs? With it, at least how many?
- 14.4 A log's last record has a valid header claiming 60 bytes, and only 41 are in the file. Torn or corrupt? What if all 60 are there but the checksum fails?
- 14.5 Why does OntoDB number LSNs with a counter rather than a byte offset?
- 14.6 A split logs the old leaf, the new leaf, and the parent, in that order. Only the first two records reach the disk before a crash. After redo, which reads still work, and which fail?
- 14.7 Why may the buffer pool not evict a dirty page whose pageLSN is newer than the log's flushable LSN, even though it may wait for an ordinary log flush?
- 14.8 OntoDB logs whole page images. Name one thing this makes simpler and one thing it makes more expensive.

Answers are in appendix B.

Further reading

Mohan et al. [2], sections 1–3; Gray and Reuter [3], chapters 9–10; Petrov [1], chapter 5.

ARIES

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics), published by Mohan, Haderle, Lindsay, Pirahesh, and Schwarz in 1992 [2], is the recovery algorithm behind most disk-based database systems. It supports steal and no force, fine-grained locking, and partial rollbacks, and it recovers correctly even if the system crashes again *during* recovery. Its central idea fits in three words: **repeat history, then undo**.

15.1 The data structures

Log records as in chapter 14: LSN, transaction id, prevLSN, and type-specific fields.

pageLSN on every page: the LSN of the last change applied to it.

Transaction table for every active transaction: its status and *lastLSN*, the LSN of its most recent record.

Dirty page table for every page dirty in the buffer pool: its *recLSN*, the LSN of the first record that made it dirty since it was last read from disk. Any change to the page before *recLSN* is already on disk.

Compensation log records (CLRs) written when a change is undone. A CLR describes the undo, and carries *undoNextLSN*: the next record of that transaction still to undo.

15.2 Checkpoints

Without checkpoints, recovery would read the log from the beginning of time. A *fuzzy checkpoint* writes the transaction table and the dirty page table into the log without stopping the system or forcing any pages, and records the checkpoint's LSN in a well-known place. Recovery then starts from the last checkpoint: the tables in it tell analysis everything that happened before.

15.3 Pass 1: Analysis

Scan the log forward from the last checkpoint, rebuilding the two tables as of the crash:

- On any record of a transaction not in the table, add it. Update its lastLSN.
- On a commit, mark it committed; on an end record, remove it.
- On an update or CLR for a page not in the dirty page table, add the page with *recLSN* = this LSN.

At the end, every transaction still in the table without a commit is a *loser*. The *redoLSN* is the smallest recLSN in the dirty page table: no earlier change can be missing from disk.

15.4 Pass 2: Redo

Scan forward from redoLSN, and reapply every update and every CLR, for winners and losers alike, unless the change is provably already on disk:

```
for each update or CLR record r, from redoLSN forward:
  if r.page not in dirty page table:      skip // page was clean at the crash
  if dirtyPageTable[r.page].recLSN > r.lsn: skip // flushed after r
  fetch page
  if page.pageLSN >= r.lsn:               skip // r already on the page
  apply r to the page; page.pageLSN = r.lsn
```

Redo **repeats history**: it brings the database to exactly its state at the crash, uncommitted changes included. That sounds wasteful, but it makes undo simple: undo now starts from a precise, known state, and can use the same logical operations that a normal abort uses.

15.5 Pass 3: Undo

Undo the losers, newest change first, across all of them at once:

```
toUndo = { lastLSN of each loser }
while toUndo is not empty:
  r = record with the largest LSN in toUndo; remove it
  if r is a CLR:
    if r.undoNextLSN exists: add r.undoNextLSN to toUndo
    else: write an end record for r.txn
  else if r is an update:
    undo r; write CLR(undoNextLSN = r.prevLSN)
    if r.prevLSN exists: add r.prevLSN to toUndo
    else: write an end record for r.txn
```

Key idea

A CLR is redone but never undone. If the system crashes during undo, the next recovery redoes the CLRs (history repeats), then follows their undoNextLSN pointers past everything already undone. Recovery work is therefore bounded no matter how many times the system crashes during recovery.

15.6 A worked example

Two transactions run. T_1 commits; T_2 does not. The buffer pool happened to write page P2 (stealing T_2 's uncommitted change), but never wrote P1.

LSN	Transaction	Record	prevLSN
10	T_1	update P1	—
20	T_2	update P2	—
30	T_1	commit	10
40	T_2	update P1	20

crash. On disk: P1 has pageLSN 0; P2 has pageLSN 20.

Analysis. T_1 committed; T_2 is a loser with lastLSN 40. The dirty page table is {P1: recLSN 10, P2: recLSN 20}, so redoLSN is 10.

Redo. LSN 10 on P1: pageLSN 0 < 10, so redo; P1's pageLSN becomes 10. LSN 20 on P2: pageLSN 20 $\geq$ 20, so skip; the stolen write already put it there. LSN 40 on P1: pageLSN 10 < 40, so redo; pageLSN becomes 40. The database is now exactly as it was at the crash, including T_2 's two changes.

Undo. toUndo = {40}. Undo LSN 40 on P1, write CLR 50 with undoNextLSN 20. Undo LSN 20 on P2, write CLR 60 with no undoNextLSN, then an end record for T_2 . T_1 's change survives; T_2 left no trace.

If the system had crashed right after CLR 50, the next analysis would find T_2 with lastLSN 50; redo would reapply CLR 50; undo would read CLR 50, jump straight to LSN 20, and never undo LSN 40 a second time.

15.7 Structure changes: nested top actions

A B+ tree split moves keys between pages. Suppose T inserts a key, which splits a leaf, and later aborts. Undoing “the split” would be wrong: another transaction may have committed an insert into the new leaf since then. The split is a change to the *structure*, which is valid regardless of who caused it; only the *logical* insert belongs to T .

ARIES logs a split as a *nested top action*: the split's page records, followed by a dummy CLR whose undoNextLSN points to the record just before the split. Redo replays the split like any other change. Undo, following the dummy CLR, skips over it. The logical “insert key k ” is logged separately, and its undo is the logical “delete key k ”, performed wherever that key now lives, which is why undo in ARIES is logical while redo is physical.

15.8 Undo must survive being repeated

Undo is logical: “take back insert t ” becomes “delete t ”. A crash can strike in the middle of that delete, and the next recovery will run it again, because the CLR is written after the change. So every logical undo must give the right answer when it runs against a state where it already half happened.

In a triple store, one triple lives in several indexes. Deleting it is three tree operations. A page group (chapter 14) makes each of the three atomic, but nothing makes the three atomic together. If the crash comes after SPO has lost the triple but before OSP has, the repeated undo meets a triple that is absent from SPO and present in OSP.

In OntoDB: ARIES

On open, LogRecovery (src/recovery/recovery.cpp) reads the whole log and runs the three passes:

Analysis. A transaction with a commit record is a winner. One with a begin, an update, or a CLR but neither a commit nor an abort record is a loser; an abort record is written when an abort or a recovery undo finishes, so it doubles as ARIES's end record. Checkpoints are sharp, so the dirty page table starts at the last checkpoint: each page's first page record after it is the page's recLSN, and the smallest is the redo LSN.

Redo. For each page record from the redo LSN on, fetch the page through the pool's raw interface, so redo is not logged again, and copy the image if the record is newer than the pageLSN. Redo is physical and the same for every page.

Undo. The catalog is opened only after redo, since redo may be what rebuilds page 0. Each loser's update records are undone newest first through the same UndoChain a normal abort uses: delete what it inserted, insert what it deleted, a CLR after each, then an abort record.

\checkpoint on a page store is CheckpointManager::Checkpoint: write every dirty page (the pool forces the log first), then append the checkpoint record listing the active transactions, and force it.

IndexedStore::Insert and Delete change every index that disagrees, rather than asking the first index and stopping. Repeating an interrupted undo therefore repairs the indexes it did not reach.

15.9 The crash harness

The bar for this part is scripts/crashtest -seeds 200. For each seed, a child process opens a real store: a database file, three trees with at most four keys a node so that a few dozen triples split and merge pages, a 16-frame pool so pages are stolen, and the log. The parent feeds it a random schedule of transactions, then kills it. A third of the seeds die between operations. The rest die at a randomly chosen reach of a named crash point (inside a split, inside a merge, or between a log flush and a page write), half of them after flushing everything the log allows, so that any half-finished change that *could* reach the disk does. The parent then reopens the store (a torn last record is cut off, then the three passes) and compares it with an oracle that applied only acknowledged commits. It also checks every tree's invariants, a point lookup for every triple, and that SPO, POS, and OSP hold the same triples.

The harness is tested by breaking the store on purpose. Removing redo fails 30 of 30 seeds. Removing page groups fails 25 of 200, as broken trees and indexes that disagree. Going back to the first-index-only Insert and Delete fails 4 of 200, as POS or OSP holding a triple SPO lacks.

Design decision 84: Physical redo, logical undo

Choice. Redo copies page images. Undo replays the inverse of each logical triple change through the store, writing a CLR for each.

Alternatives. Physical undo from before-images; logical redo of triple changes.

Why. This is ARIES's own split. Redo must work on pages that may be inconsistent with each other, so it cannot depend on the tree's structure. Undo runs after redo has repeated history and the tree is sound, and it must take back one transaction's triple even if other transactions have since moved it to a different page.

Cost. Two kinds of record for one statement, and the store's operations must be idempotent, since undo can be repeated after a crash part way through.

Design decision 85: A sharp checkpoint

Choice. Checkpoint flushes every dirty page, then writes the checkpoint record with the active transactions. Updates must not run while it does.

Alternatives. ARIES's fuzzy checkpoint, which writes the transaction and dirty page tables without flushing anything or stopping the system.

Why. After a sharp checkpoint nothing older than its record needs redo, so analysis builds the dirty page table from the page records after it, and the checkpoint needs no table of its own. Fuzzy checkpoints are an optimization to add once the three passes are known to be correct.

Cost. Updates stall while every dirty page is written. The log is not yet truncated after a checkpoint, so analysis still reads the whole file to find losers that began before it.

Design decision 86: Recovery is tested against an oracle

Choice. A crash harness kills a real store at random points and at named crash points, reopens it, and compares it with an oracle that applied only acknowledged commits, plus structural checks. The harness is itself tested on deliberately broken stores.

Alternatives. Hand-written recovery scenarios only.

Why. The bugs in recovery live in interleavings nobody thinks to write. The first version of this harness only killed between operations on a workload too small to split a page. It passed while a crash inside a split could leave a broken tree. Making it grow deep trees and die inside operations found that bug and the non-idempotent undo within minutes.

Cost. Slow tests (about two minutes for 200 seeds), and failures that need a seed to reproduce. Tree descents and the leaf iterator now fail with an error on a cycle, so a corrupt store fails a seed rather than hanging the harness.

Status

Abort by the log (plan 8.4), the sharp checkpoint (plan 8.5), the three passes (plan 8.6), and the crash test (plan 8.7) run, and 200 seeds pass. What is not built yet: the shell does not open a page store from a directory (ontodb <dir> is the memory database of chapter 16), the log is never truncated, checkpoints are not fuzzy, and a page record is a full image.

Check your understanding

- 15.1 In the worked example, suppose the pool had also written P1 after LSN 40. What does redo do now, and what does undo do?
- 15.2 Why does ARIES redo the losers' changes, only to undo them right after?
- 15.3 The system crashes after CLR 60 but before T_2 's end record. What does the next recovery do?
- 15.4 Why is a B+ tree split never undone, even when the transaction that caused it aborts?
- 15.5 What do the dirty page table's recLSNs let redo skip?
- 15.6 A loser inserted triple t . Recovery's undo deletes t from SPO and POS, then the machine stops again. Why must the next recovery's undo still delete t from OSP, and what would go wrong if Delete looked only at SPO?
- 15.7 Why does OntoDB open the catalog after redo instead of before analysis?
- 15.8 A crash harness passes 200 seeds. What else must be true before that means anything?

Answers are in appendix B.

Further reading

Mohan et al. [2], the whole paper; Ramakrishnan and Gehrke [4], chapter 18, for a gentler walkthrough; Petrov [1], chapter 5.

Durability in Practice

ARIES is the general answer for a page-based engine. A simpler engine can make a simpler choice and still be correct. OntoDB's persistent databases, which run today, are such a design: an in-memory database made durable by a **snapshot** plus a **logical, redo-only write-ahead log**. This chapter explains the design through the ideas of the previous chapters, because every choice in it is one of those ideas in a smaller form.

16.1 Where it sits in the steal/force table

The database lives entirely in memory; the files are its durable copy. Nothing is ever written to the data files except a whole snapshot, and a snapshot is taken only between statements, when every change in memory is committed. In the terms of table 3.2:

- **No steal.** Uncommitted changes never reach the snapshot. So recovery never needs to *undo* anything on disk.
- **No force.** A commit does not rewrite the snapshot; it appends to the log. So recovery must *redo* committed changes made since the last snapshot.

No steal plus no force is the *redo-only* cell: the log needs redo information and nothing else, and recovery is a single forward pass.

Key idea

Choosing a cell of the steal/force table chooses the recovery algorithm. Steal and no force need undo and redo (ARIES). No steal and no force need only redo. The price of no steal is that all uncommitted state must fit in memory, which an in-memory database accepts by definition.

16.2 Logical logging

Page-based ARIES logs *physical* or *physiological* changes: this page, these bytes, or this page, this slot. The memory database has no pages, so it logs *logical* operations: “insert triple (18, 9, 4)”. Logical records are small and independent of storage layout, and they are safe here only because there is no partially written page that could make a logical operation ambiguous on replay.

Ids in those records must mean the same terms after a restart, so the log also carries the dictionary: a *term record* for each spelling the first time a transaction uses it.

16.3 One statement, one transaction

1. The statement runs against a *recording store*, a decorator around the real store that remembers every triple it actually changed.
2. The log receives **begin**, a **term** record for each new spelling, one **update** per changed triple, and **commit**.
3. The commit is forced (chapter 14) before the shell prints the result.

If the statement fails part way, the recording store's changes are undone in memory, in reverse order, and nothing it needs reaches the log. A statement that changed nothing logs nothing.

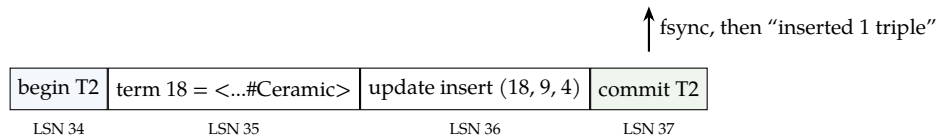


Figure 16.1: The log records of one INSERT DATA in a persistent database.

16.4 Recovery

Opening a database directory:

1. Load the snapshot, if there is one: every term in id order, then every triple.
2. Scan the log and stop at the first record that is torn or fails its checksum (chapter 14).
3. Skip records at or below the snapshot's LSN; the snapshot already contains them.
4. Apply *every* term record in order, whatever its transaction did: the dictionary only grows, and a term no committed triple uses is invisible.
5. Apply update records only for transactions that have a commit record, in log order.

Step 5 is the whole of “undo” in a redo-only design: an uncommitted transaction's records are simply never applied.

16.5 Checkpoints as atomic snapshots

A checkpoint writes the full state to `snapshot.tmp`, fsyncs it, renames it over `snapshot`, fsyncs the directory, and then empties the log. Each step is safe to crash after:

Crash after	On reopen
writing <code>snapshot.tmp</code>	old snapshot, full log: nothing lost
the rename	new snapshot, old log: records at or below its LSN are skipped
emptying the log	new snapshot, empty log

This is the log-structured idea of chapter 6 with a single level: an append-only log, periodically compacted into one sorted image.

16.6 Damaged logs: recovery modes

When the scan in step 2 stops early, there are two defensible policies, named after RocksDB's WAL recovery modes:

Point in time (the default). Keep every transaction before the bad record, truncate the log there, and report how many bytes were discarded. A torn final record is the normal shape of a crash, and this is the right answer for it; a transaction whose commit is after the damage is treated as uncommitted, exactly as a crash would.

Absolute consistency (`-strict`). Refuse to open, so an operator can inspect the files before anything is truncated.

A damaged snapshot always refuses to open: it has no valid prefix to fall back to.

16.7 Verification

Detecting damage early is the other half of durability. The `\verify` command follows the tradition of PostgreSQL's `pg_amcheck`, MySQL's `CHECK TABLE`, and RocksDB's `VerifyChecksum`: it changes nothing and reports four checks, one line each.

```
snapshot: ok (lsn 37, 18 terms, 15 triples)
wal: ok (0 records, 0 bytes)
replay: matches the live database (15 triples)
invariants: ok (no duplicate triples, every id in the dictionary)
verify: ok
```

The replay check is the strongest: it rebuilds the database from the two files into a fresh store and compares it with the one in memory. If they differ, the files would not reproduce what the user sees.

16.8 Testing a recovery design

A recovery path that is never exercised does not work. OntoDB tests this one with *crash points*: named places in the code, such as just before a commit record is appended and just after a snapshot is renamed, where a test can make the operation stop as if the process died. Scenario tests then reopen the directory and check what survived: a crash before commit leaves no trace, a crash after the snapshot rename does not replay records twice, a torn tail keeps the committed prefix, a flipped byte in the middle of the log drops everything after it, and a tampered log fails verification.

In OntoDB: what runs

All of this chapter runs today: start the shell as `ontodb mydb. \load` and every update are durable transactions, `\checkpoint` writes a snapshot, `\log` prints the decoded log, `\verify` checks the files, and `\crash` exits without flushing, to try recovery by hand. Seventeen scenario tests cover the cases above, and they pass under AddressSanitizer, UndefinedBehaviorSanitizer, and ThreadSanitizer.

The page store uses the same log records, log manager, and group commit. What changes there is that pages can be stolen, so recovery gains undo, pageLSNs, page images, and the three passes of chapter 15.

Design decision 87: Terms replay whatever their transaction did

Choice. On recovery, every term record is applied in order, even if its transaction never committed; update records apply only for committed transactions.

Alternatives. Replay terms only for committed transactions.

Why. Ids are dense, and each term record carries the id it was given. If an aborted transaction's term were skipped, every later term would land on a different id than the log recorded. A term no committed triple uses is invisible.

Cost. Aborted transactions can leave unused terms behind.

Design decision 88: Point-in-time recovery by default

Choice. A damaged log is truncated at the first bad record, and the database opens with everything before it, reporting what was dropped. `-strict` refuses to open instead.

Alternatives. Strict by default; or skip the bad record and continue.

Why. A torn final record is the ordinary result of a crash, and the right response is to open. Skipping a bad record would apply later transactions on top of a missing one.

Cost. Damage in the middle of the log silently loses every later commit apart from the report, which is why `\verify` exists.

Check your understanding

- 16.1 Why can the memory database's log omit undo information?
- 16.2 A crash happens after the snapshot rename but before the log is emptied. Which records does reopen apply?
- 16.3 A transaction logs a term record for id 19 and then fails. A later committed transaction logs id 20. Why must replay still apply the term for 19?
- 16.4 When would you choose `-strict` over point-in-time recovery?
- 16.5 Why is the replay check of `\verify` stronger than checking checksums alone?

Answers are in appendix B.

Further reading

Gray and Reuter [3] on redo-only logging; DeWitt et al. [24] on recovery for main-memory databases; the RocksDB WAL recovery-mode documentation [25].

From Memory to Pages

OntoDB now has two complete recovery designs. The memory database of chapter 16 is redo-only: no steal, a logical log, and snapshots. The page store of chapter 15 steals pages, logs page images, and recovers with ARIES, but only tests open it. This chapter is about the step that joins the second design to the shell, so that `ontodb <dir>` keeps its data on pages. Most of the work is not new algorithms; it is deciding what a directory *is*, what opening one does, and which operations get which kind of log record. Those decisions are where real systems differ.

17.1 What is in a directory

A database directory must say which design wrote it, because the two have incompatible files. The memory database keeps snapshot and wal; the page store keeps db, the database file of chapter 2, and its own wal. A one-line format file names the store and its version, pages 1, much as PostgreSQL keeps a `PG_VERSION` file in every data directory.

Directory holds	Opens as
format = pages 1	a page store
snapshot, no format	a memory database, as today
format naming another store or version	refused, saying which
nothing	a new page store (or memory, with <code>-memory</code>)

Refusing an unknown format is the important row. A newer OntoDB that changes the page layout must not be opened by an older one that would misread it, and the cheapest protection is a version number checked before any page is read.

Design decision 89: A format file names the store (planned)

Choice. New directories get format with the store and version; a directory without it but with a snapshot is a memory database.

Alternatives. Detect the store from which files exist; or put the version in the database file's header page.

Why. Detection by file names breaks the day both designs share a name, and a header page cannot describe a store that has no pages. A separate file is read before anything else and is trivial to inspect.

Cost. One more file to create atomically with the others.

17.2 Opening is recovery

On the memory path, opening means loading a snapshot and replaying the log. On the page path, opening a directory runs all of chapter 15, in an order that matters:

1. **Cut a torn tail.** Scan the log and truncate it at the first record that is incomplete or fails its checksum. The recovery modes of chapter 16 apply: point in time truncates and reports; -strict refuses.
2. **Analysis and redo.** Rebuild the transaction table and the dirty page table, then repeat history from the smallest recLSN, so every page is as it was at the crash, uncommitted changes included.
3. **Open the catalog, the dictionary, and the indexes.** Only now are the pages that describe them trustworthy, since redo has brought them up to date.
4. **Undo.** Roll back the loser transactions, writing compensation records, exactly as an abort does.

Step 3 sits between redo and undo for a reason. Redo is physical: it needs page ids and bytes, not trees. Undo of an index change is logical, “remove this key from that tree”, because a split may have moved the key to another page since it was inserted; logical undo needs the tree, and the tree needs its root page, which the catalog names.

Key idea

Redo works on pages and needs nothing above the buffer pool. Undo works on structures and needs the access methods. Recovery opens the engine layer by layer, in that order.

17.3 Statements log on pages

On a page directory the transaction manager is given the log manager, and from then on the machinery of chapters 14 and 15 runs for every statement: each page change logs a record, commit forces the log, and abort undoes by the log. The dictionary’s pages are pages like any other, so a new term is covered by the page records of the term heap and the dictionary tree; there is no separate term record as on the memory path. A term added by a transaction that then aborts stays in the dictionary, for the same reason as there: ids are handed out densely, and an unused term is invisible.

The test for this step is blunt and sufficient: commit a triple, kill the process before any page is written, reopen, and find the triple. Only the log can have put it there.

17.4 Loading: one record per triple, or one per load

\load of a large file is one transaction, and it is where the two obvious logging designs diverge most:

Log every change. Each inserted triple dirties leaf pages of three trees, and each change is logged. Simple and uniform: a load is just a big transaction. The log grows with the data, roughly three page records per triple until pages fill, and the load runs at the speed of logging.

Minimal logging. Build the trees bottom-up from sorted input (chapter 4’s bulk load), write every page to the file, force the file, and log one record that names the load. Undo of the load is “drop the new trees”. SQL Server calls this minimal logging, and PostgreSQL, with `wal_level` set to `minimal`, skips the log for a table created in the same transaction, for the same reason.

Minimal logging is only correct when the trees it builds start empty (or are new trees swapped in at commit): there is nothing older on those pages for a partial load to have damaged, so undo can discard them whole.

Design decision 90: How a load is logged (planned, to be measured)

Choice. Decide between per-triple logging and minimal logging by measuring both on LUBM(1): log bytes and load time. A crash during a load must leave the database as it was before it, either way.

Alternatives. The two above; or per-page images of the newly built trees.

Why. The answer depends on numbers this engine has not produced yet: how much faster a bulk build is than three tree inserts per triple, and how much log a load writes.

Cost. Minimal logging needs empty target trees and forces every page before commit; per-triple logging makes the log as large as the data.

17.5 Closing cleanly, and checkpoints

A clean exit (`\quit`, or end of input) writes every dirty page, forces the log, and closes the file. The next open then finds nothing to redo, and its summary says so; that is the cheapest test that the close was clean. `\checkpoint` runs the sharp checkpoint of chapter 15 with every statement held out, because a sharp checkpoint must not see a page change in the middle of being made. Chapter 18 replaces it with a fuzzy one.

17.6 Checking a page database

`\verify` keeps its promise of changing nothing and grows three checks: every tree satisfies its invariants (chapter 4); SPO, POS, and OSP hold the same set of triples; and every id in them is in the dictionary. The second check is the one specific to RDF: three indexes that disagree return different answers depending on which one the optimizer chooses, the worst kind of bug to find from the query side.

17.7 Testing through the shell

The crash harness of chapter 15 kills a store driven by test code. The last step drives the real Database with SPARQL text on a page directory and kills it between statements and at named crash points. Test-only options make tree nodes tiny and the buffer pool small, so a short script splits, merges, and steals pages constantly; otherwise a test database never leaves its first page, and the hard paths never run.

Status

Planned, and next (plan 10.1–10.7). Today `ontodb <dir>` opens the memory database of chapter 16, and the page store’s recovery runs only in tests and in the crash harness.

Check your understanding

- 17.1 Why must an older OntoDB refuse to open a directory whose format names a newer version, rather than try?
- 17.2 Why are the catalog and the indexes opened after redo but before undo?
- 17.3 A transaction commits one `INSERT DATA`, and the process is killed before any page is written. Where does the reopened database find the triple?
- 17.4 Under what condition is minimal logging of a load correct, and what does undo of such a load do?
- 17.5 What does a clean close guarantee about the next open, and how can a test observe it?
- 17.6 Give a query whose answer depends on which of SPO, POS, or OSP the optimizer scans if the three disagree.

Answers are in appendix B.

Further reading

Mohan et al. [2] on logical undo of index changes; Gray and Reuter [3] on recovery-time ordering; the SQL Server documentation on minimally logged operations [32].

Keeping the Log Small

A write-ahead log only grows. Every commit appends, nothing removes, and a database that runs for months would keep every record since its creation and replay from the beginning of time on every crash. ARIES as described in chapter 15 is correct but not yet livable. This chapter covers the three pieces of housekeeping a long-lived page store needs: cutting the log after a checkpoint, making page records smaller, and checkpointing without stopping the world. Each is a statement about what recovery can still need, made precise.

18.1 What recovery still needs

After a crash, ARIES redo starts at the smallest *recLSN* in the dirty page table: the first change to any page not yet written. Undo walks back through every loser transaction, so it needs each active transaction's records back to its first one. Everything older than both is dead weight:

$$keep = \min \left(\min_{p \text{ dirty}} recLSN(p), \min_{T \text{ active}} firstLSN(T) \right).$$

A record below *keep* describes a change already on disk, made by a transaction already finished; no crash can need it. A checkpoint is the moment these two minima are known and written down, which is why truncation follows a checkpoint.

Key idea

The log is a queue of obligations: redo for pages not yet written, undo for transactions not yet finished. Its tail can be cut exactly at the oldest obligation still open.

18.2 Cutting the log

Chapter 16 empties its log after a snapshot, because the snapshot holds everything. The page store cannot empty it: after a checkpoint, dirty pages and active transactions still depend on records near the end. It keeps the suffix from *keep* on, and must do so without ever leaving a log that recovers differently:

1. Write the records from *keep* on to `wal.tmp`, and `fsync` it.
2. rename it over `wal`, and `fsync` the directory.

Crash after	On reopen
writing <code>wal.tmp</code>	the old, full log: recovery as before
the rename	the cut log: same recovery, less to read

LSNs are counters, not file offsets, so the records keep their LSNs and every pageLSN on disk still points where it did. The file only needs to know the LSN of its first record, which goes in its header.

Design decision 91: Truncate by copying the tail (planned)

Choice. After a checkpoint, copy the needed suffix to a new file and rename it over the log.

Alternatives. A log made of fixed-size segment files, where truncation deletes whole old segments, as PostgreSQL and InnoDB do.

Why. One file keeps the log manager, the scanner, and the crash harness as they are, and copying is cheap right after a checkpoint, when the suffix is short.

Cost. Copying a long suffix while a long transaction stays open; segment files would avoid the copy and cost a directory of files to manage.

18.3 Torn pages and smaller page records

Chapter 15 logs a page's full image with every change. That is simple and robust, and it makes the log as large as the pages it touches: insert one 24-byte key into a 4 KiB leaf, and the log grows by 4 KiB.

Logging only the changed bytes is the obvious saving, and it has one hazard. A page write is not atomic: the device may write some sectors of a page and not others, leaving a *torn page* that is neither the old version nor the new. Redo that applies a byte diff to a torn page produces garbage, because the diff assumes the bytes around it are the old ones.

PostgreSQL's answer, *full-page writes*, is the one the plan adopts: the *first* change to a page after a checkpoint logs the page's full image, and later changes until the next checkpoint log only the byte ranges that differ from the page as it was when latched. Recovery starts from the checkpoint, so for every page redo touches, it meets that page's full image first and rebuilds the page whole, torn or not; every later diff then applies to a known image.

Inserting 100 keys into one leaf, one checkpoint	Log bytes
Full image every change	$100 \times 4,096 \approx 410 \text{ KB}$
Full image first, then diffs (about 40 bytes each)	$4,096 + 99 \times 40 \approx 8 \text{ KB}$

A diff must also say what it applies to. Redo applies a diff only to a page whose pageLSN equals the LSN of that page's previous record, so a diff can never land on the wrong version of the page.

Design decision 92: Full image after a checkpoint, then diffs (planned)

Choice. Log a page's full image on its first change after each checkpoint, and byte-range diffs for later changes.

Alternatives. Full images always (today); diffs always, with a doublewrite buffer to repair torn pages (InnoDB).

Check your understanding

- 18.1 The dirty page table holds p3 (recLSN 140) and p9 (recLSN 212); T7 is active with first LSN 120 and T8 with first LSN 300. From which LSN must the log be kept?
- 18.2 Why may truncation keep the old LSNs instead of renumbering from zero?
- 18.3 What goes wrong if redo applies a byte diff to a torn page, and how does a full image after each checkpoint prevent it?
- 18.4 Why does a diff record need the LSN of the page's previous record?
- 18.5 What does a fuzzy checkpoint write, and why may its tables be out of date when they reach the log?
- 18.6 Why does a fuzzy checkpoint make the page cleaner more important?

Answers are in appendix B.

Further reading

Mohan et al. [2] on fuzzy checkpoints and the dirty page table; Gray and Reuter [3] on log management; the PostgreSQL documentation on WAL and full-page writes [33].

CHAPTER A

A Map from Chapters to OntoDB

Each chapter’s ideas, the OntoDB plan items that implement them, and where the code lives. Plan items are listed in the repository’s `PLAN.md`; run `scripts/check <feature>` to build and test one.

Chapter	Plan items	Where
1 The engine	0.1, 0.3	<code>tools/shell</code> , <code>src/common/database.cpp</code>
2 Pages	1.1, 3.1	<code>src/storage/disk</code> , <code>src/storage/page</code> , <code>src/storage/term</code>
3 Buffer pool	1.2–1.6	<code>src/buffer</code> , <code>src/storage/page/page_guard.cpp</code>
4 B+ trees	2.1–2.7, 6.1–6.2	<code>src/storage/index/b_plus_tree.cpp</code>
5 Indexing RDF	0.2, 3.2–3.6	<code>src/storage/index/triple_key.cpp</code> , <code>src/dictionary</code> , <code>src/catalog</code> , <code>src/store</code>
6 LSM trees	—	(contrast only)
7 Plans	4.4–4.6	<code>src/parser</code> , <code>src/binder</code> , <code>src/planner</code> , <code>src/execution</code>
8 Joins	4.1–4.3, 5.1–5.5	<code>src/execution/executors</code> , <code>src/execution/row_merge.cpp</code> , <code>src/optimizer</code> , <code>src/stats</code>
9 Beyond BGPs	12.1–12.5	(planned)
10 Inference	13.1–13.5	(planned)
11 Semantic search	14.1–14.9	<code>src/semantic</code> , <code>tools/bench/semantic.cpp</code> , <code>docs/semantic-search.md</code>
12 Transactions	7.2, 7.4	<code>src/concurrency</code> , <code>test/isolation/specs</code>
13 Concurrency	7.1–7.6	<code>src/concurrency</code> , B+ tree crabbing
14 Logging	8.1–8.3	<code>src/recovery/log_record.cpp</code> , <code>src/recovery/log_manager.cpp</code>
15 ARIES	8.4–8.7	<code>src/recovery</code> , <code>test/recovery</code> , <code>scripts/crashtest</code>
16 Durability	9.1–9.7	<code>src/recovery/durability.cpp</code> , <code>src/recovery/snapshot.cpp</code> , <code>test/durable</code>
17 Memory to pages	10.1–10.7	(planned, next)
18 Log housekeeping	11.1–11.3	(planned)

The OntoDB manual (<https://lamng3.com/ontodb/>) is the companion to this book: it documents how to use each feature, while this book explains why it is built the way it is.

CHAPTER B

Answers

Short answers to the questions at the end of each chapter. Try a question before reading its answer; most of them take one line of reasoning, and that line is the point.

chapter 1: What a Database Engine Is

- 1.1 (a) The B+ tree, an access method. (b) The buffer pool. (c) The optimizer.
- 1.2 It touches every layer: executors take locks on the keys they read and write, every page change is logged, and commit is decided at the interface. It cuts across the stack instead of sitting in it.
- 1.3 A five-way self-join of the triple table on the subject column, each pattern being a selection on that table, followed by a projection.
- 1.4 Only Metal. The stored triples say `Metal subClassOf Material`; `Alloy subClassOf Material` would have to be inferred, and OntoDB has no reasoner.

chapter 2: Disks, Files, and Pages

- 2.1 At byte $37 \times 4096 = 151,552$. Pages are fixed-size, so the offset is arithmetic on the page id.
- 2.2 The index points at the record id, $(page, slot)$. Compaction changes the offset stored in the slot, not the slot number.
- 2.3 No: it was only in the operating system's page cache. `fsync` on the file (and, for a new file, on its directory) makes it durable.
- 2.4 The kernel may already have dropped the dirty pages and cleared the error, so the second `fsync` "succeeds" with the data gone. The only safe response is to stop and recover from the log.
- 2.5 Zero is what unwritten memory and files contain. With -1 as the sentinel, a forgotten initialization shows up as an error instead of as a tree rooted at page 0, which is a legal page.
- 2.6 OntoDB refuses it: a term heap record must fit on one page (at most 4,076 bytes), so the term cannot be interned and the load fails as one transaction. PostgreSQL would compress it and, if still large, move it out of line into a TOAST table, leaving a pointer in the row.
- 2.7 Allocation grew the file, but the new length was never made durable (no `fsync` before the crash), while the log record that uses the page was forced. Redo reads the page past the end as zeros and reapplies the record; writing the page back grows the file.

chapter 3: The Buffer Pool

- 3.1 It must be written back to disk (after the log is flushed up to its pageLSN, by the write-ahead rule) before the frame is reused.
- 3.2 At $t = 11$ the backward 2-distances are A: $11 - 8 = 3$, B: $11 - 3 = 8$, C: $11 - 7 = 4$. LRU-K evicts B.
- 3.3 X. Both have fewer than three accesses, so both have infinite distance, and the tie goes to the earliest oldest access: X's is at time 2, Y's at time 5. Recency does not enter until a frame has K accesses.
- 3.4 A scan touches each page once; under LRU every scanned page becomes the most recent, so the scan pushes out the hot pages. With $K = 2$, a page touched once has infinite distance and leaves first. $K = 1$ is LRU.
- 3.5 Both steal cells. Under steal, an uncommitted change can reach the disk, and the log must be able to take it back.
- 3.6 The bytes just written: one thread runs requests in the order scheduled. With several I/O threads the read could run first and return the old page, unless requests for the same page were ordered some other way.
- 3.7 Until the first thread's read finishes: the pool's one latch is held across the I/O, and the single I/O thread would serialize the reads anyway.
- 3.8 It would write the very page the crash point was meant to keep off the disk, so the test could not observe the state a real crash leaves.
- 3.9 Once the pin is gone the pool may evict the frame and load a different page into it; a latch still held would then be a latch on someone else's page.
- 3.10 Page 7 is now resident (another page may have been evicted to make room), and its LRU-K history holds two recent accesses, so it has a finite, small distance and will be among the last to go.
- 3.11 That frame can never be evicted, so the pool is permanently one frame smaller; enough leaks and every frame is pinned and the next miss fails. Guards unpin exactly once at the end of their scope, and the tests check that nothing stays pinned.

chapter 4: B-Trees and B+ Trees

- 4.1 A 24-byte key plus a 4-byte child page id. n keys separate $n + 1$ children, so the leftmost child has no key of its own, and storing slots uniformly simply leaves that key unused.
- 4.2 $300^3 = 2.7 \times 10^7 < 10^8 \leq 300^4$, so four levels. The root and the roughly 300 pages below it are always cached, so a lookup costs about two reads.
- 4.3 The right (new) leaf, whose first key it is. Left in the left leaf, a key equal to the separator sits in the left child, which breaks the separator rule, and a search for 14 goes right.
- 4.4 Leaf keys are the data, and every key must stay in a leaf. Internal keys only route searches, so the middle one can move up and leave both halves.

- 4.5 Holding two leaves, and possibly a parent, can use up a small pool; with latches, two scans moving in opposite directions can deadlock.
- 4.6 A split would have to rewrite the parent pointer in every child that moved, up to 72 pages for one internal split. Without them, a structure change re-fetches parents remembered by id, which is safe only while one thread changes the tree; latch crabbing fixes that.
- 4.7 Two leaves of 85. A leaf of 1 would be below the minimum of 84 keys, so the last two leaves share their 170 keys evenly.
- 4.8 A full leaf splits into two half-full leaves, so random inserts leave leaves about two-thirds full, and each insert is a random descent. Bulk load writes full leaves sequentially.

chapter 5: Indexing RDF

- 5.1 258 is 0x0102. Big-endian: 00 00 00 00 00 00 01 02; little-endian: 02 01 00 00 00 00 00 00. Against 3 (...00 03 and 03 00...), big-endian puts 258 above 3, correctly; little-endian puts it below.
- 5.2 (5, ?, ?): [(5, 0, 0), (6, 0, 0)]. (5, 7, ?): [(5, 7, 0), (5, 8, 0)].
- 5.3 With $n = 2$, the predicate overflows to 0 and the carry bumps the subject: (6, 0, 0). With $n = 1$ and subject $2^{64} - 1$, nothing is above the prefix, so the bound saturates to $(2^{64} - 1, 2^{64} - 1, 2^{64} - 1)$.
- 5.4 Object bound: OSP. Subject and object bound: OSP, whose prefix is object then subject.
- 5.5 Otherwise one term gets two ids, and a join between loaded data and a query constant silently matches nothing. Example: a datatype written with a prefix, "21"^^xsd:integer, and the same datatype written as a full IRI must become one spelling.
- 5.6 Every operator would read that term as "unbound": a join would overwrite the slot from the other side, a scan would treat a constant as a variable, and the saturated upper bound could exclude a real key. The answers would be wrong without any error.
- 5.7 With SPO, POS, and OSP, every pattern shape is already a prefix of some index, so it is already a range scan; another order can only offer the same range sorted differently. That matters to an operator that wants its input in a particular order, such as a merge join on the object, or a sort that an index could avoid.
- 5.8 Range scans: subject bound; subject and predicate; all three; and none (the whole tree is the range). Subject and object bound is a subject range filtered by object. Predicate only, object only, and predicate with object are full scans with a filter.
- 5.9 The prefix scan on the hash returns both ids; the dictionary reads each candidate's spelling from the heap and returns the one that is equal. A collision costs an extra read, never a wrong answer.
- 5.10 A root moves whenever it splits or collapses; the header page absorbs those changes. The catalog changes only when an index is created or dropped.

chapter 6: The Log-Structured Alternative

- 6.1 An LSM tree: writes are sequential and batched. A B+ tree pays a random page write for each insert; the LSM tree pays later in compaction, and in read amplification.
- 6.2 Older versions of the key sit in immutable SSTables that are never edited. The tombstone shadows them until compaction drops both.
- 6.3 One wasted SSTable read. A Bloom filter never says “absent” for a present key.
- 6.4 Appending to the log is the write path, and a checkpoint rewrites the whole state as one sorted image, which is a compaction into a single level that discards the old log.

chapter 7: From SPARQL to a Plan

- 7.1 ?s is slot 0, ?name slot 1, ?age slot 2, in order of first appearance.
- 7.2 Rows can differ in slots that are not projected: two students with the same name are two scan rows but one projected row.
- 7.3 Ids are assigned in load order, not value order; “10” can have a smaller id than “9”. Sorting must compare values.
- 7.4 Volcano stops as soon as ten rows reach the root, doing only the join work that produced them. Materializing computes the entire join first, then keeps ten rows.
- 7.5 Run generation writes 100 runs of 10 pages (1,000 pages). Merging 8 at a time gives 13 runs, then 2, and 2 is few enough for the final merge to stream to the consumer: two intermediate passes, each writing 1,000 pages. In all, 3,000 pages written (and 3,000 read back), plus the input scan.
- 7.6 Neither. Without ORDER BY the order of solutions is unspecified, so any two of the matching names are a correct answer. The backends produce different orders (insertion order against key order).
- 7.7 An empty pattern has exactly one solution, the empty binding, and that row is the identity for a join. Zero rows would make every join empty.

chapter 8: Joins and Query Optimization

- 8.1 Nested loop: $10^3 \times 10^6 = 10^9$ comparisons. Index nested loop: 1,000 prefix probes, each a root-to-leaf descent (mostly cached) plus the few matching triples.
- 8.2 When both inputs are large, no index makes a probe selective, and the build side fits in memory.
- 8.3 The exact number of triples with that predicate and object, read from the pair counts. Errors return at joins, where the estimator must assume how join values are distributed, and after updates, until the statistics are collected again.
- 8.4 $(10^4 \times 10^3)/10^6 = 10$. The q-error is $\max(10/1000, 1000/10) = 100$.

- 8.5 It keeps the 2,000 course rows, keyed by ?c, and makes 50,000 lookups, one per probe row. Swapped, it would keep 50,000 rows to answer 2,000 lookups: the same answer for 25 times the memory, which is why the build side should be the smaller one once estimates exist.
- 8.6 With rebinding, each of the 1,000 outer rows binds ?s, so the inner scan is a prefix probe returning that student's names: about 1,000 probes reading a few triples each. Without it, every outer row rescans all 1,000,000 ub:name triples: 10^9 triples read.
- 8.7 A join's estimate is built from its inputs' estimates, so their errors multiply: two inputs each off by 10 give an output off by up to 100.
- 8.8 \explain prints the optimizer's plan. If the executor swapped algorithms at run time, the printed plan would not be the one that ran.
- 8.9 It joins three type patterns that share no variable: about 1,900 graduate students times 1,000 universities times 15 departments, tens of millions of rows of cross product, before the first pattern that connects them. Any operator must produce those rows; only putting a connected pattern first avoids them.

chapter 9: Beyond Basic Graph Patterns

- 9.1 (Steel, unbound), (Copper, 1085), (Glass, unbound). Steel's 1370 fails the join filter, so Steel survives padded; the filter decides only which right rows may join.
- 9.2 Inner join is symmetric, so either side can be built. A left join must pad the preserved side's unmatched rows; probing with it pads at probe time, while building on it needs a matched flag per entry and a final pass to emit the unmatched ones, a different algorithm.
- 9.3 Both ?a and ?b. Rows from the left side have ?b unbound, and rows from the right side have ?a unbound.
- 9.4 It keeps the set of distinct values per group, so its memory grows with the data rather than with the number of groups; a hash table sized for the groups can overflow because of those sets.
- 9.5 Every left row survives a left join, so its output has at least as many rows as its left input. An estimate of 40 below a left input of 100 is impossible, a bug in the estimator.
- 9.6 The scan can meet triples the statement itself inserted, so which matches it processes depends on where new triples land relative to the cursor, and templates that create new matches can keep generating work. Materializing the solutions of WHERE first fixes them before anything changes.

chapter 10: Materialized Inference

- 10.1 Rewriting wins when the hierarchy changes often and is queried rarely, or the closure is huge and queries touch little of it. Materializing wins for read-heavy workloads that ask for classes with their subclasses, the usual case for ontologies.

- 10.2 Round 1: $A \text{ subClassOf } C$ (rdfs11) and $x \text{ a } B$ (rdfs9). Round 2: $x \text{ a } C$. Round 3: nothing new, stop.
- 10.3 A new triple can match either body pattern of a rule. Joining it only as the first premise misses every derivation in which it is the second.
- 10.4 In a cycle the links support each other's derivations, so after the asserted support is deleted each still has a count from the cycle, though nothing outside the cycle derives it. DRed overdeletes everything the deleted triple could have supported and puts back only what follows from what remains, so self-support cannot keep a triple alive.
- 10.5 It should still hold the triple, now as an inferred one. The store must record which triples were asserted, separately from whether they are entailed, so the delete removes the assertion and rederivation keeps the entailment.

chapter 11: Semantic Search

- 11.1 Ids are given out in the order terms are first seen in the file, so two classes are close in id only if their IRIs first appeared close together. It would have to list each class right after its parent (a depth-first listing of the hierarchy), and even then it would order structure, not meaning.
- 11.2 The grid has $n = 100$ and diameter 18, so some two neighbors are at least $99/18 = 5.5$, that is 6, positions apart in any order. (The true bandwidth of the 10×10 grid is 10.)
- 11.3 With the tree $A(B(D), C)$ numbered in postorder, $D = 1, B = 2, C = 3, A = 4$. The sets are $D : \{[1]\}$, $B : \{[1, 2]\}$, $C : \{[1], [3]\}$ (its own [3] plus the interval of its child D), and $A : \{[1, 4]\}$.
- 11.4 At most 6: one for its own tree interval plus one for each non-tree edge below it. The bound is tight only if every non-tree head lies outside the others' intervals.
- 11.5 Pre-filtering reads only the members, so it wins when the class is small, and it is exact. It loses when the class holds most of the terms, because it then scores nearly all of them while an index would score a few. Post-filtering is the reverse.
- 11.6 The names of a class's descendants are close to the class and to one another, so a query from inside the class has most of its nearest neighbors inside it too. The independence estimate assumes the filter is unrelated to the similarity.
- 11.7 It cannot decide from the node whether the subtree is below or above the bound, because the skipped bytes are not stored. It must load a key from the subtree, compare it with the bound, and only then choose a side.

chapter 12: Transactions and Isolation

- 12.1 $r_1(A)$ before $w_2(A)$ gives $T_1 \rightarrow T_2$; $r_2(B)$ before $w_1(B)$ gives $T_2 \rightarrow T_1$. The graph has a cycle, so the schedule is not conflict-serializable.

- 12.2 $r_1(A) w_2(A) c_2 r_1(A)$: T_1 reads two different values. Repeatable read is the first level that forbids it.
- 12.3 Validity is defined by the application's invariants, such as "every student has a name". The engine cannot know them; it provides atomicity, isolation, and durability, so that the application can keep them.
- 12.4 Nothing from the statement: the insert operator deletes the two triples it had added before the error propagates. That holds even with no log.
- 12.5 Both read N and both insert a student they believe is number $N + 1$: an anomaly built on a phantom (write skew). It is possible at read uncommitted, read committed, and repeatable read. Serializable's range locks make one insert wait; if both hold the range in S and both want to insert, it is a deadlock, and the younger transaction is aborted.

chapter 13: Concurrency Control

- 13.1 Plain 2PL may release an X lock in its shrinking phase, before commit; another transaction reads the value and must abort if the writer later aborts. Strict 2PL holds X locks until commit or abort, so no one reads an uncommitted write.
- 13.2 IS on the database and S on the "spo" index, with no key locks. Any writer below must hold IX on the index, and IX conflicts with S, so the one check on the index covers every key.
- 13.3 T_5 , the younger (larger id).
- 13.4 A latch is a short mutex on a physical page for one operation; it protects a page's structure during a split. A lock is held for a transaction on a logical item, such as a triple key, and provides isolation.
- 13.5 For an insert, a child is safe if it is not full, so it cannot split. For a delete, it is safe if it is more than half full, so it cannot underflow.

chapter 14: Logging and the Write-Ahead Rule

- 14.1 Rule 1: a page is written only after the log records for its changes are durable; this makes steal safe. Rule 2: a commit is acknowledged only after its commit record is durable; this makes no force safe.
- 14.2 Skip it: $\text{pageLSN } 40 \geq 35$, so the change is already on the page.
- 14.3 Without group commit, ten. With it, as few as one, or two if a flush was already running when they arrived.
- 14.4 Short: a torn write. Full length with a bad checksum: corruption. In point-in-time mode, either one ends the valid prefix there.
- 14.5 A checkpoint of the memory database empties the log file. Byte offsets would restart at zero; a counter keeps increasing, so comparisons with the snapshot's LSN stay correct.

- 14.6 Redo rebuilds both leaves, and the old leaf's sibling link points to the new one, but the parent has no separator or pointer for it. A scan along the leaf chain still finds every key. A point lookup for a key that moved to the new leaf descends to the old one and misses it, and an insert of that key would add a duplicate.
- 14.7 Writing it requires the log to be durable up to its pageLSN, and that LSN belongs to an open page group, so the flush would wait for the group to end. If the thread choosing a victim is the one running that group, it would wait for itself. An ordinary flush always finishes.
- 14.8 Simpler: redo is one copy, the same for every kind of page, and needs no code that understands the page. More expensive: every change logs about 4 KB per page, so the log grows by hundreds of times more than with records that describe the change.

chapter 15: ARIES

- 15.1 Analysis is unchanged: the dirty page table still has P1 at recLSN 10, because nothing logged the flush. Redo fetches P1, sees pageLSN 40 $\geq$ each record's LSN, and skips all three. Undo is unchanged: undo 40 (CLR 50), undo 20 (CLR 60), end.
- 15.2 Repeating history puts the database in exactly its state at the crash, so undo can use the same logical operations as a normal abort, and CLRs from an interrupted earlier undo are replayed, not repeated.
- 15.3 Analysis finds T_2 a loser with lastLSN 60. Redo reapplies CLRs 50 and 60 if the pages lack them. Undo reads CLR 60, which has no undoNextLSN, and writes the end record. Nothing is undone twice.
- 15.4 The split is a structural change, valid whoever caused it, and other transactions may since have inserted into the new leaf. Only the logical insert belongs to the transaction, and its undo is a logical delete.
- 15.5 Records for pages that were not dirty at the crash, and records older than a page's recLSN, without reading the page.
- 15.6 The CLR for that undo is written only after the delete finishes, so the next recovery undoes the same insert again. OSP still holds t , so the delete must remove it there. A Delete that asked SPO first would find t gone, report nothing to do, and leave OSP holding a triple the other indexes lack, so a query that scans OSP returns it and one that scans SPO does not.
- 15.7 Redo may be what rebuilds page 0, the catalog: if the crash came after its record was durable but before the page was written, the page on disk is stale or zeros. Reading the index roots before redo could use the wrong roots.
- 15.8 That the harness fails when the store is broken: break redo, the page groups, and the idempotent undo in turn, and watch seeds fail. The workload must also reach the cases that matter, here splits, merges, and crashes inside them.

chapter 16: Durability in Practice

- 16.1 No steal: uncommitted changes never reach the snapshot, and replay simply does not apply an uncommitted transaction's updates.
- 16.2 The new snapshot. Every record in the old log is at or below the snapshot's LSN, so none is applied.
- 16.3 Ids are dense, and each term record carries its id. Skipping 19 would put the next term on id 19, so committed triples naming 20 would mean a different term; OntoDB's replay detects this and refuses.
- 16.4 When damage could come from something other than a crash (a failing disk, copied or edited files) and nothing should be truncated before someone looks.
- 16.5 Checksums show each record is intact as written. Replay shows the files actually reproduce the live database, which also catches logic bugs, such as a change that was never logged, that leave perfectly valid checksums.

chapter 17: From Memory to Pages

- 17.1 It cannot know what the newer layout means. Reading it could make recovery "repair" good pages into garbage and write them back. Refusing loses nothing.
- 17.2 Redo is physical and needs only page ids and bytes. Undo of an index change is logical, because splits may have moved the key, so it needs the trees, whose roots the catalog names, and the catalog's pages are correct only after redo.
- 17.3 In the log: commit forced it. Redo replays the page records onto the pages that were never written.
- 17.4 When the trees it builds start empty, or are new trees swapped in at commit. Undo drops those trees and frees their pages.
- 17.5 That the next open has nothing to redo or undo. A test reopens and checks that the recovery summary says so.
- 17.6 `SELECT ?s WHERE { ?s ?p :Material } scans OSP, while SELECT ?o WHERE { :Steel ?p ?o } scans SPO. If OSP lacks a triple SPO holds, the first query misses it and the second finds it.`

chapter 18: Keeping the Log Small

- 18.1 From LSN 120: the smaller of the smallest recLSN (140) and the smallest first LSN of an active transaction (120).
- 18.2 pageLSNs on disk and LSNs inside records refer to each other; renumbering would break the comparisons redo depends on. The file records the LSN of its first record instead.

- 18.3 A diff assumes the bytes around it are the old version, but a torn page mixes old and new sectors, so the result is neither. The first change after a checkpoint logs the full image, so redo rebuilds each page whole from the log before any diff applies.
- 18.4 So that it applies only to the exact version it was computed against. Redo checks that the page's pageLSN equals that LSN.
- 18.5 A begin record, then an end record with the active transactions and each dirty page's recLSN; no pages. The tables are copied while transactions run, so changes between begin and end may be missing, and analysis replays the records from the begin record to catch up.
- 18.6 With no page writes at checkpoint time, recLSNs advance only when pages are written in the background. The cleaner keeps the redo start, and so the part of the log that must be kept, recent.

Bibliography

- [1] A. Petrov. *Database Internals: A Deep Dive into How Distributed Data Systems Work*. O'Reilly, 2019.
- [2] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.
- [3] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [4] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, 3rd edition. McGraw-Hill, 2003.
- [5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.
- [6] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
- [7] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.
- [8] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.
- [9] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD*, 1993.
- [10] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, 1984.
- [11] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
- [12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [13] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.
- [14] T. Neumann and G. Weikum. RDF-3X: A RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
- [15] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.
- [16] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

- [17] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.
- [18] S. Harris and A. Seaborne, editors. SPARQL 1.1 Query Language. W3C Recommendation, 2013.
- [19] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.
- [20] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Modelling in Data Base Management Systems*, 1976.
- [21] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.
- [22] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *OSDI*, 2014.
- [23] PostgreSQL Global Development Group. Fsync errors (“fsyncgate”). PostgreSQL wiki, 2018. https://wiki.postgresql.org/wiki/Fsync_Errors.
- [24] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.
- [25] RocksDB. WAL recovery modes. RocksDB wiki. <https://github.com/facebook/rocksdb/wiki/WAL-Recovery-Modes>.
- [26] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):16:1–16:45, 2009.
- [27] P. J. Hayes and P. F. Patel-Schneider, editors. RDF 1.1 Semantics. W3C Recommendation, 2014.
- [28] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [29] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.
- [30] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel materialisation of Datalog programs in centralised, main-memory RDF systems. In *AAAI*, 2014.
- [31] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2–3):158–182, 2005.
- [32] Microsoft. The transaction log: operations that can be minimally logged. SQL Server documentation. <https://learn.microsoft.com/en-us/sql/relational-databases/logs/the-transaction-log-sql-server>.
- [33] PostgreSQL Global Development Group. Write-ahead logging: reliability and full_page_writes. PostgreSQL documentation, chapter “Reliability and the Write-Ahead Log”. <https://www.postgresql.org/docs/current/wal.html>.

- [34] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.
- [35] T. Grust. Accelerating XPath location steps. In *SIGMOD*, pages 109–120, 2002.
- [36] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.
- [37] Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.
- [38] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia. ACORN: Performant and predicate-agnostic search over vector embeddings and structured data. *Proceedings of the ACM on Management of Data*, 2(3):120, 2024 (SIGMOD).
- [39] J. J. Pan, J. Wang, and G. Li. Survey of vector database management systems. *The VLDB Journal*, 33:1591–1615, 2024.
- [40] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013.
- [41] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *SODA*, pages 1027–1035, 2007.
- [42] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.
- [43] N. Reimers and I. Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *EMNLP-IJCNLP*, 2019.
- [44] M. Ashburner et al. Gene Ontology: tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.
- [45] P. Holanda. Persistent storage of adaptive radix trees (ART) in DuckDB. DuckDB blog, 2022. <https://duckdb.org/2022/07/27/art-storage>.