

Forum:[Hack Forums](#)

Thread: [Lossless integer index shrinking ~10x](#)

iconia - August 17, 2026, 1:14 pm

(I had ai help me organize this info, if it seems a bit sterile, forgive that)

Shrinking an integer index ~10x with delta + varint encoding (lossless)

A practical deep-dive on making "which records contain X" lookups tiny and fast — with copy-paste code, real before/after numbers, and the gotchas nobody warns you about. No libraries, no magic. Just how the big search engines have done it for 20 years.

Part 1 of 2 — the problem, the two ideas, the code, and the numbers. Part 2 (lossless proof, boundaries, prod gotchas) is in the reply below.

Quote:Scope: a storage/encoding tutorial for anywhere you keep lists of integer IDs — an inverted index, event logs, a graph's adjacency lists, time-series row IDs, a tag->items map. If you've ever stored one row per (key, id) pair and watched your DB balloon, this is for you.

Code ROW-PER-ENTRY (fat)		ONE-BLOB-PER-KEY (thin)	
word	rid	word	rids (packed blob)
nginx	17	nginx	[11 16 01 34 ..] 1-2 B/id
nginx	39	apache	[09 02 51 ..]
nginx	40 .. 58,000 rows ..		6,000 rows (one/word)
apache	12 774 KB		~78 KB

1. The problem, concretely

Say you're building a search/inventory tool. You want to answer "which records mention nginx?" fast. The classic structure is an inverted index: for every term, a list of the record IDs that contain it.

The obvious way to store it in SQLite (or Postgres, or a KV store) is one row per pair:

```
Code CREATE TABLE postings (  
  word TEXT NOT NULL,  
  rid INTEGER NOT NULL,      -- record id that contains `word`  
  PRIMARY KEY (word, rid)  
);  
-- 'nginx' -> 4, 4 -> 17, 4 -> 39, 4 -> 40, ... one row each
```

It works. It's also enormous, and the reason is not your data — it's the per-row overhead. Every row carries a cell header, key bytes, and page slack. In a real index I measured, this table held 58,000 rows costing 774 KB — about 13.3 bytes to store what is essentially one integer. The actual integers were a rounding error; the packaging was the cost. On a realistic corpus, that one table was ~33% of the entire database.

Code one nginx -> 39 row on disk (~13 bytes):
[cell header][key "nginx" + rid][rid ~1B][page slack]
_____~12 bytes of packaging_____/ ^ the actual data
The goal: store the same information in a fraction of the bytes, still exact, still fast to read. We'll get

to ~1.35 bytes per id — roughly 10x smaller — with two ideas.

2. Baseline — measure before you optimize

Never optimize a number you haven't measured. Here's how to get the honest size of a table in SQLite (VACUUM first so free pages don't lie to you):

Code # size of just the postings table, isolated (copy DB, drop everything else, VACUUM)

```
sqlite3 index.db "VACUUM;"
python - <<'PY'
import sqlite3, os
c = sqlite3.connect("index.db")
rows = c.execute("SELECT COUNT(*) FROM postings").fetchone()[0]
words = c.execute("SELECT COUNT(DISTINCT word) FROM postings").fetchone()[0]
print(f'{rows:}, rows across {words:} words -> {rows/words:.1f} ids/word avg')
PY
```

Write that number down. Everything below is measured against it.

3. Idea #1 — Delta encoding (store the gaps, not the values)

Your id list for a word is a set of integers. Sort it. Now, instead of storing the absolute values, store the difference from the previous one:

Code absolute: 17, 39, 40, 92, 93, 95

deltas: 17, 22, 1, 52, 1, 2

Why this helps: the ids might be large (5-7 digits), but the gaps between sorted ids are usually small, especially when the ids cluster (records added around the same time share words). Small numbers are cheap to store — if you have a way to store a small number in a small space. Which is idea #2.

4. Idea #2 — Varint / LEB128 (small number = few bytes)

A normal integer takes a fixed width — 4 or 8 bytes — no matter how small it is. A variable-length integer (varint) uses only as many bytes as the number needs: 1 costs 1 byte, 200 costs 1 byte, 50,000 costs 3 bytes.

The scheme (LEB128, the same one Protocol Buffers and DWARF use): chop the number into 7-bit groups, little-endian. Use the high bit of each byte as a "continue" flag — 1 means "more bytes follow," 0 means "this is the last byte."

Code 14 = 0001110 -> 1 byte: 0000_1110

300 = 10_0101100 -> groups 0101100 | 0000010

byte0 = 1_0101100 (cont) byte1 = 0_0000010 (stop) -> 2 bytes

Combine the two ideas — delta then varint — and each id in a clustered list collapses to 1-2 bytes instead of 13.

5. The code (copy-paste, ~25 lines, no dependencies)

Code def encode_ids(ids) -> bytes:

```
"""Sorted-set of ints -> delta+varint blob. Lossless. Input order irrelevant."""
```

```

out = bytearray()
prev = 0
for n in sorted(ids):
    d = n - prev          # delta from previous (ascending, so d >= 1 for a set)
    prev = n
    while d >= 0x80:      # emit 7 bits at a time, high bit = "more follow"
        out.append((d & 0x7F) | 0x80)
        d >>= 7
    out.append(d)         # final byte, high bit clear = "stop"
return bytes(out)

```

```

def decode_ids(blob) -> set:
    """delta+varint blob -> the original set of ints. Exact inverse of encode_ids."""
    out, run, val, shift = set(), 0, 0, 0
    for b in blob:
        val |= (b & 0x7F) << shift
        if b & 0x80:      # continue bit set -> accumulate more
            shift += 7
        else:             # stop bit -> we have a full delta
            run += val     # undo the delta (running sum)
            out.add(run)
            val, shift = 0, 0
    return out

```

New schema — one row per word, the whole id list in a single BLOB:

```

Code CREATE TABLE postings (
    word TEXT PRIMARY KEY,
    rids BLOB NOT NULL      -- encode_ids(the id set)
) WITHOUT ROWID;

```

Read path is a single-row fetch + a tight decode loop:

```

Code def get_ids(conn, word):
    row = conn.execute("SELECT rids FROM postings WHERE word=?", (word,)).fetchone()
    return decode_ids(row[0]) if row else set()

```

6. Before / after — the actual numbers

Same data, same queries, only the encoding changed. Measured on a real corpus, VACUUMed:

	rows	bytesper id	row-per-entry	58,000	774 KB	13.3 B	delta+varint blob	6,000 (one/word)	~78 KB	~1.35 B

That's ~10x smaller on the index, and because the index was ~a third of the DB, the whole database dropped ~20-29% (the effect grows with data — more ids per word = more deltas that pack to 1 byte). And reads got faster, not slower: one blob fetch + a decode loop touches far fewer pages than scanning hundreds of individual rows.

Verify the win is real (don't trust a cached WAL — checkpoint + VACUUM):

```

Code sqlite3 index.db "PRAGMA wal_checkpoint(TRUNCATE); VACUUM;"
ls -l index.db      # <- the honest number
-> Continued in Part 2 below: proving it's lossless, when NOT to use it, and the four gotchas that bite you in production.

```

Part 2 of 2 — losslessness, boundaries, and prod gotchas

(Continued from Part 1 above, where we built the delta+varint encoder and got the index ~10x smaller. This half is the part that keeps you out of trouble.)

7. Losslessness — prove it, don't assume it

The single most important test for any compression: round-trip fuzzing. Encode random sets, decode them back, assert equality. If this passes thousands of times, you're exact.

```
Code import random
for _ in range(10000):
    s = set(random.sample(range(1, 5_000_000), random.randint(0, 400)))
    assert decode_ids(encode_ids(s)) == s      # exact inverse, always
print("round-trip: lossless over 10k random sets")
```

Edge cases this covers for free: empty set (-> empty blob), single element, huge gaps, duplicate inputs (a set dedups them). Run it in CI. Compression bugs are silent and awful; this is your seatbelt.

8. When it helps, and when it doesn't (be honest)

Great fit: Sorted lists of ids/keys/timestamps — inverted indexes, adjacency lists, event streams. Clustered ids (things added near each other share terms -> small deltas -> 1-byte each). Read-heavy: you fetch the list, iterate, done.

Poor fit (know the boundary): Random, sparse 64-bit ids with no clustering — deltas are huge, varints hit 8-9 bytes, you save little. If you can, intern your ids to small dense integers first (map the big external id -> a compact internal rowid) and pack those. Write-hot single lists — see the gotcha below.

The general principle worth internalizing: compression is just removing the redundancy you can predict. Sorted ids are predictable (they only go up, usually by a little), so you store the surprise (the gap), not the value. That framing carries to text, audio, anything.

9. Gotchas (the stuff that bites you in prod)

1. Insert = read-modify-write the blob. To add one id you must read the blob, decode, add, re-encode, write. That's fine if a word's list is bounded, but if a single key can grow to millions of ids, that RMW gets expensive. Two fixes: Cap the list (drop ultra-common terms entirely — see #2), so no blob is huge.

Overflow table: keep a small (word, rid) scratch table for recent inserts, and periodically merge it into the packed blobs in a batch. Read = blob + overflow.

2. Skip the ultra-common terms (the "stop-word" move). A word that appears in most records is useless for narrowing a search (it matches everything) yet its posting list is the biggest. Don't store it. Pick a cap — e.g. "appears in more than N records" — and drop those lists entirely; a query for a common word gathers nothing distinctive anyway. In my index this alone removed ~40% of the postings and made queries faster. The rarest terms are the ones that actually discriminate; spend your bytes there.

Code records
with word

```
^
| #          <- a few ultra-common words: DROP
| #          (match everything, biggest lists)
| # #
| # # # #
| # # # # # # # # <- the long tail of rare words: KEEP
+-----> words, most-common -> rarest
the discriminating signal lives out here --^
```

3. Rebuild from a temp table, don't hoard in RAM. When you (re)build the whole index, don't accumulate every id list in memory. Flush (word, rid) pairs to a temp table, then SELECT ... ORDER BY word, rid and stream-aggregate one word at a time into blobs. Bounded memory, any corpus size.

4. WITHOUT ROWID on the blob table means the row is the b-tree entry keyed by word — one copy, not two. Small win, free.

10. Wrap-up

Two textbook ideas — delta encoding (store the gap) and varint/LEB128 (small number, small space) — turn a bloated (key, id) table into a compact one-blob-per-key index that's ~10x smaller, lossless, and faster to read. Add the "drop the ultra-common terms" move and you shave another big chunk while improving query quality.

None of this is exotic — it's how Lucene, Elasticsearch, and every serious search index have worked for two decades. But writing it yourself, measuring the before/after, and knowing exactly why it works is the difference between using a black box and owning your storage.

Full code above is dependency-free and public-domain — take it, measure your own before/after, and post your numbers.

Questions / corrections welcome — happy to go deeper on the overflow-merge scheme or the common-term cap if there's interest.

flyingduck - August 17, 2026, 1:23 pm

super HQ shit bro! :) keep it up! :)

itcorp - August 17, 2026, 2:43 pm

Nice share bud! Keep it up! :)

iconia - August 17, 2026, 2:46 pm

(Aug 17, 2026 01:23 PM)flyingduck Wrote: [\[CITE\]](#) super HQ shit bro! :) keep it up! :) thanks!!

(Aug 17, 2026 02:43 PM)itcorp Wrote: [\[CITE\]](#) Nice share bud! Keep it up! :) thanks man!