

Bitmap indexes for WooCommerce catalogs on ordinary hosting



Mika Sipilä, Retry AI Studio

linkedin.com/in/mika-sipila · Searchmaker · searchmaker.io

ABSTRACT

Product search and faceted filtering in WooCommerce are served by SQL queries over posts, postmeta and term tables, executed inside a full WordPress request. This works for small catalogs and degrades as catalogs, attributes and traffic grow. We describe the architecture of Searchmaker, a local catalog engine that replaces those queries with an immutable, file-based index of per-term bitmaps and posting lists, answered by a standalone read-only PHP endpoint that does not bootstrap WordPress. The design targets ordinary shared or VPS hosting: no external services, no daemons, bounded memory, atomic index switches, and fail-soft fallback to WooCommerce. We describe the index format, query semantics and ranking tiers, incremental maintenance by patches and delta layers, per-site synonym compilation and privacy-safe learning from zero-result searches, and report development measurements on a synthetic 100,000-product catalog. We state clearly what these measurements do not show: a head-to-head comparison on identical hardware and a measurement on a real large MariaDB-backed shop are still to be done.

1 Introduction

A WooCommerce product is a post with metadata and term relationships. A search for a word, filtered by category, brand, two attributes and stock status, is therefore a join across several large tables, preceded by the cost of loading WordPress, the active theme and every active plugin. Caching helps repeated pages but not the long tail of filter combinations, and large catalogs with variable products multiply the rows involved. Hosted search services avoid the database but introduce a subscription, a data-sharing relationship and an additional point of failure, which many shops prefer not to take on.

We ask a narrower question: how fast and small can catalog search and filtering be if the only environment assumed is what a typical WooCommerce host already provides: PHP, the filesystem and the existing database? Our answer is to treat the catalog as read-mostly data, compile it into a compact immutable index outside the database, and query it with bit operations.

2 Design constraints

1. **Ordinary hosting.** PHP and the filesystem only. Redis, SQLite, WASM, external services, shell access and daemons are never required.
2. **Bounded memory.** No product arrays in PHP on the query path. Reads are fixed-size; builders stream; the Google Merchant feed writer is a streaming writer.
3. **Immutable indexes.** A new version is built beside the active one, validated, and switched in atomically. The active index is never modified in place.
4. **WordPress is the source of truth, not the runtime.** WordPress and WooCommerce are used to build the index and for administration; public catalog queries are answered by an endpoint that does not load WordPress.
5. **Parent-oriented results.** The result unit is the parent product; variations contribute SKUs, attributes, stock and a min/max price.
6. **Fail soft.** If the index is missing, corrupt or of an unknown format, the shop falls back to ordinary WooCommerce behaviour.

3 Index

3.1 *Terms, bitmaps and posting lists*

Every searchable or filterable fact is a *term*: a category, a brand, an attribute value, a stock status, a word in a title, a SKU, a character trigram. Products receive dense integer ids in ascending source order. A term's postings are stored either as a dense bitmap (one bit per product, 12.5 KB at 100,000 products) or as a sorted list of 32-bit ids, whichever is smaller for that term, a choice made at build time. This is the classic trade-off between bitmap indexes (O'Neil and Quass, 1997) and inverted lists (Zobel and Moffat, 2006), which compressed schemes such as Roaring bitmaps (Chambi et al., 2016) refine; we use the simpler two-representation scheme because it needs no dependency and keeps the reader a few hundred lines of PHP.

Term lookup is a binary search over a sorted file of fixed 24-byte records (an 8-byte hash prefix, a count, a storage type and an offset). No dictionary is held in memory, so the cost of a request is a handful of small reads rather than a parse of the whole index.

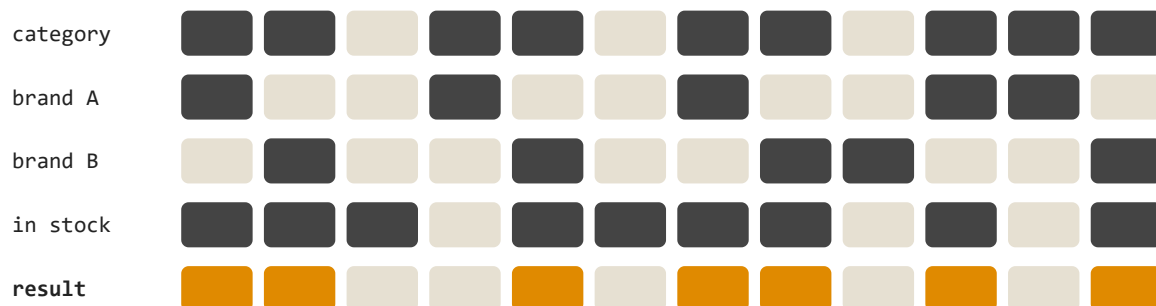


Figure 1. Twelve products, four filter bitmaps and the result of `category AND (brand A OR brand B) AND in stock`. At catalog scale the same operation runs over 12.5 KB per term.

3.2 Files and versions

An index lives in a versioned directory per language and consists of a manifest with byte lengths and checksums, the term table, the bitmap and posting files, a sorted vocabulary with offsets for prefix lookup, one JSON line per product card with a random-access offset file, the id map, and four small *volatile* files holding prices (min and max) and stock. A pointer file names the active version. A build writes a temporary directory, verifies sizes and checksums, renames it to the next version number, and replaces the pointer atomically. The previous version is retained for one-click rollback; anything that a retained version shares files with is retained too. Any exception during a build removes the temporary directory and leaves the pointer untouched.

An index format change (for example after a plugin update) makes older indexes unreadable by design; the plugin detects this, serves WooCommerce results, and queues a rebuild.

4 Query execution

The query language is deliberately small: OR within a facet group, AND across groups, an optional scope that must also match (for example the category of an archive page), a price range, and text. Facet counts for a group ignore that group's own selection but respect everything else. The candidate bitmap computed once for the result set is reused for all counts, so five facet groups cost roughly five bitmap intersections rather than five database queries.

Text search is lexical and tiered. A query token matches exactly; the last token (and any token without an exact hit) also matches by prefix; tokens of four or more characters also match inside longer compound words by trigram overlap, which is important for compounding languages (a search for *saha* finds *moottorisaha*); a token with no match falls back to fuzzy trigram similarity for typos. Results are partitioned into disjoint bitmaps by rank: an exact SKU or model match; all tokens exact; prefix; and everything else. All text is lowercased and accent-folded on both the indexing and query side.

Sorting by price uses a single top-k pass over the volatile price file with memory proportional to k, falling back to a full sort for small candidate sets and very deep pages.

5 Keeping the index fresh

Prices and stock change constantly (orders, ERP synchronisation); titles and categories rarely do. We therefore separate *volatile* from *structural* change. WooCommerce reports exactly which properties changed on a save; if only price, sale dates or stock fields changed, the product is queued and, after a short debounce, a **patch** is written: a new immutable version containing only the four volatile files, referencing every other file in the version that holds it (references never chain, so a patch of a patch still points at the base).



Structural edits (a new title, SKU or category; new, trashed or deleted products) change the dense id space and cannot be patched into the base. They are applied as **delta layers**: a tombstone bitmap hides the base versions of edited or removed products and a small complete index holds the current version of every edited and new product. A layered reader composes both behind the same interface the query engine already uses, so combined bitmaps are simply the base AND NOT tombstones, concatenated with the delta. The plugin falls back to a full rebuild when the delta would exceed three percent of the catalog or on any error. Term-level changes (a renamed category, a new brand) and the first build always rebuild.

6 Language: synonyms and learning

Synonym packs. Language-specific vocabulary is data, not code. Packs are plain JSON lists of folded term pairs with a direction and a confidence. At configuration time a pack is *compiled against the site's own index*: every alternative whose words do not occur as tokens in the catalog is dropped, which keeps

typically only 15 to 25 percent of a pack and removes rules that could never match. At query time, one extra query is run for each rule that matches a one- or two-word window of the query, bounded to eight variants per search; matches reachable only through a rule rank below every match of what the shopper typed.

Learning. When a search finds nothing and the shopper then searches for something else and clicks a result, an anonymous event pair is recorded by a separate append-only endpoint. A daily job promotes a reformulation to a rule only if several distinct visitors made it, at least one clicked, they are a meaningful share of everyone who ended on the first query, the first query still finds nothing, the second finds something, both are one or two words without digits, and neither is a protected brand term. Rules are capped and recomputed from the last thirty days on every run, so unsupported rules disappear without a separate demotion mechanism. No IP address, user agent or user id is stored; the visitor is a day-salted hash of a random per-tab identifier, personal-looking queries are dropped, and raw events are deleted after 35 days.

7 Runtime and security

The public query endpoint is a few dozen lines that load an autoloader and one small configuration file written by the plugin; it never loads WordPress. It accepts only GET, bounds query-string length, query length, filter groups and keys, page and page size, takes the language and enabled request modes from configuration rather than the request, and returns a generic error without detail when the index is unavailable. It cannot write. Learning events use a separate script that accepts only small JSON bodies, writes size-capped JSON lines into a directory whose name includes a random key, and answers identically whether an event was stored or dropped. Fake events are possible (pages are cached, so browsers cannot be authenticated); the defence is in promotion thresholds, caps and automatic demotion rather than in the endpoint, and the worst case is a harmless extra rule for a word that found nothing anyway.

Licenses are Ed25519-signed JSON payloads verified offline, with no license server and no network call from the plugin.

8 Evaluation

Setup. PHP 8.3 / FPM / 128 MB, shared-style host. 100,000 products, 165,264 variations, ~38.8k terms. Averages of repeated runs; engine time only (no HTTP, no WordPress). These are development measurements by the authors, not an independent benchmark.

Table 1. Engine time per query, synthetic catalog of 100,000 products.

Query	Time
-------	------

Query	Time
Category + in stock	0.49 ms
Category + brand + 2 facets + stock	0.39 ms
Multiselect (OR within, AND across groups)	0.42 ms
Price range + category + stock	0.47 ms
Search + filters + first 24 product cards	0.64 ms
Counts for five facet groups	4.42 ms
Naive full scan in PHP (baseline)	53.6 ms

Table 2. Maintaining the index, 100,000 products.

Operation	Result
Full rebuild	6.1 s (peak memory about 57 MB)
Price/stock patch, 2,000 changes	178 ms (writes 0.9 MB instead of 41 MB)
Edit delta, 300 edited + 100 new products	93 ms (layered queries stay at 14 to 27 ms)
Google Merchant feed, 100,000 items	0.78 s (about 128,000 items per second, bounded memory)

Two observations. First, queries that would each be a multi-table join in SQL complete in well under a millisecond because they are bitwise operations over a few kilobytes; the naive full scan used as a baseline is two orders of magnitude slower. Second, maintenance cost scales with what changed: a patch of two thousand price changes wrote 0.9 MB in 178 ms, against 41 MB and about ten seconds for a rebuild, which is what makes ERP-driven shops practical. On a real, slow store of roughly 6,500 products, engine times were in the single-digit milliseconds while round trips through WordPress's REST path were commonly 0.45 to 0.55 seconds and the theme's built-in live search (an admin-ajax request) averaged about 1.4 seconds, which motivated the standalone endpoint.

Threats to validity. The catalog is synthetic, and the benchmark index stores several representations at once, so its 75.5 MB size overstates a production index. Engine time excludes the web server and network. Cold-request and concurrency measurements are still to be added. Most importantly, we have not yet published a head-to-head comparison with WordPress's default search or with other plugins on

identical hardware and the same catalog; [the benchmarks page](#) marks those rows as placeholders and describes the protocol.

9 Limitations and future work

- Search is lexical. There are no embeddings; semantic matching is approximated by open synonym packs and learned reformulations.
- Standard simple and variable products, categories, one configured brand taxonomy and global attributes are supported. Arbitrary custom-field filters are out of scope.
- Compatibility has been verified on a block theme, a classic theme and Salient with WPBakery. WPML was tested through a simulation and Polylang is untested on a real installation; TranslatePress is unsupported.
- Synonym packs are machine-generated and not yet reviewed by native speakers.
- Planned: a head-to-head benchmark protocol run on identical MariaDB hosts, a real-catalog measurement at 100,000 products, learned-rule outcome tracking, and events from the results page and Fast Grid.

References

1. P. O'Neil and D. Quass. Improved query performance with variant indexes. *SIGMOD*, 1997.
2. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys* 38(2), 2006.
3. S. Chambi, D. Lemire, O. Kaser and R. Godin. Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience* 46(5), 2016.

© Searchmaker. Measurements are development measurements under the stated conditions. The synonym data discussed here is open under CC BY 4.0: github.com/searchmaker/searchmaker-language-packs.