



CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Point of Contact: Edward C. Zimmermann E-mail: <edz@corequarry.com> rev F: August 2026

Disclaimer

This document is intended to provide an overview of the CoreQuarry project, its objectives, architecture, and anticipated capabilities. It is a working document and may be updated as the project evolves. Descriptions of planned features, performance characteristics, and development priorities reflect current intentions and should not be interpreted as guarantees of future functionality or delivery schedules.

The information contained herein is provided for discussion, evaluation, and planning purposes. Technical details, implementation approaches, and project scope may change in response to ongoing research, testing, community feedback, and technological developments.

Invitation for Collaboration & Community Feedback

CoreQuarry is an evolving, community-informed project, and constructive feedback is essential to its development. We actively welcome comments, technical critiques, architecture discussions, and open-source contributions from developers, researchers, and system designers.

Whether you are interested in reviewing the technical design, benchmarking capabilities, reporting edge cases, or contributing directly to the codebase, we invite you to connect with us through any of the following channels:

- **GitHub Repository:** <https://github.com/re-lsearch/CoreQuarry> -- Explore the codebase, review architecture proposals, track upcoming milestones, or submit issues and pull requests.
- **Discord Community:** <https://discord.gg/wvn6K7JPg> -- Join real-time discussions on implementation details, technical strategy, and engine design with the community.
- **Direct Contact:** [\[edz@CoreQuarry.com\]](mailto:edz@CoreQuarry.com) -- For inquiries regarding research collaboration, integration discussions, or private feedback.

Technical Executive Summary

CoreQuarry is a high-performance, local-first knowledge retrieval platform that makes advanced AI-powered search accessible on consumer hardware, **ensuring complete data and infrastructure sovereignty**. It enables exploration and interaction with complex collections of documents using a unified combination of keyword search, semantic retrieval, and document structure awareness— **completely insulated from foreign legal jurisdiction, cloud lock-in, and the risk of exposing sensitive intellectual property to third-party AI models**.

Our design paradigm is especially suited to the emerging next generation of LLM and RL agents. Agents need more than keyword matching. They need to express relationships: this term occurs inside that field, these concepts appear near each other, this phrase comes before that phrase, these facts belong to the same structural element. Positional and structural operators let an agent ask about the shape of information, not merely its presence.

Unlike conventional RAG systems, which typically divide documents into arbitrary chunks before embedding them into a single semantic space, CoreQuarry exploits existing document structure. Semantic retrieval can be performed within specific fields, paths, record types, or structural regions, creating smaller and more coherent semantic spaces. This reduces ambiguity, improves retrieval precision, and preserves

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

contextual information that is often lost in conventional chunking pipelines.

Structural and positional search gives an agent a way to preserve that information before generation: retrieve the right relationship, not just the right words. For agentic search, operators such as NEAR, BEFORE, PEER, WITHIN, and field constraints are therefore not syntactic luxuries; they are tools for turning a flat language model into a much more precise information-seeking system.

The platform is designed from its very foundations to deliver accurate, explainable, and auditable results while operating on affordable hardware, reducing both infrastructure costs and energy consumption. This makes it particularly suitable for research institutions, cultural heritage organisations, archives, libraries, enterprises, and public-sector bodies seeking to deploy trustworthy AI systems while maintaining control over their data and infrastructure.

By providing the retrieval layer required for modern agentic Retrieval-Augmented Generation (RAG) applications, CoreQuarry helps organisations transform large collections of information into accessible knowledge, supporting discovery, decision-making, and new forms of human–AI collaboration. Its local-first architecture enables deployment on everything from laptops and edge devices to institutional servers, making advanced knowledge retrieval practical, sustainable, and accessible at a broad range of scales.

CoreQuarry Innovations:

- CoreQuarry employs an object-oriented retrieval architecture where fields, records, chapters, pages and collections can all serve as retrieval targets.
- CoreQuarry implements a structural retrieval architecture in which queries operate on navigable document structures rather than fixed document representations.

Unlike most search platforms, which transform documents into intermediate representations and retrieve primarily at the document or stored-field level, CoreQuarry is built around a document-handler (DOCTYPE) model. Each document type encapsulates knowledge of its structure, record boundaries, fields, encodings, presentation rules, and conversion logic. Rather than storing complete transformed copies of content, the system records the locations and structural relationships of information within the original source whenever possible. At search time, the appropriate DOCTYPE is invoked to navigate, reconstruct, normalize, and present the requested object. This allows retrieval to operate on logical entities such as fields, records, pages, chapters, speakers, scenes, or entire collections without requiring intermediate content stores or application-specific extraction layers. The result is a flexible "any-to-any" architecture in which document structure remains available throughout the retrieval process, enabling structural navigation, dynamic transformation, and query-time selection of the most appropriate unit of retrieval.

- CoreQuarry's architecture centres on a **dynamic search-time unit of retrieval**. Rather than returning only documents, queries operate over structural document models exposed by document handlers and datatype-specific indexes. Search results may therefore be individual fields, records, pages, chapters, speakers, complete documents, collections, or dynamically generated XML and JSON representations. Semantic retrieval through Schmate (building on the ggml tensor library and a vastly enhanced and turbo charged HNSW implementation) extends this framework by adding embedding-based datatypes while preserving the same retrieval and navigation model.
- **Rich Operator Support:** Full set of binary and unary operators (AND, OR, XOR, NEAR, BEFORE/AFTER, PEER, distance-based, field-specific operations) allow precise relational queries

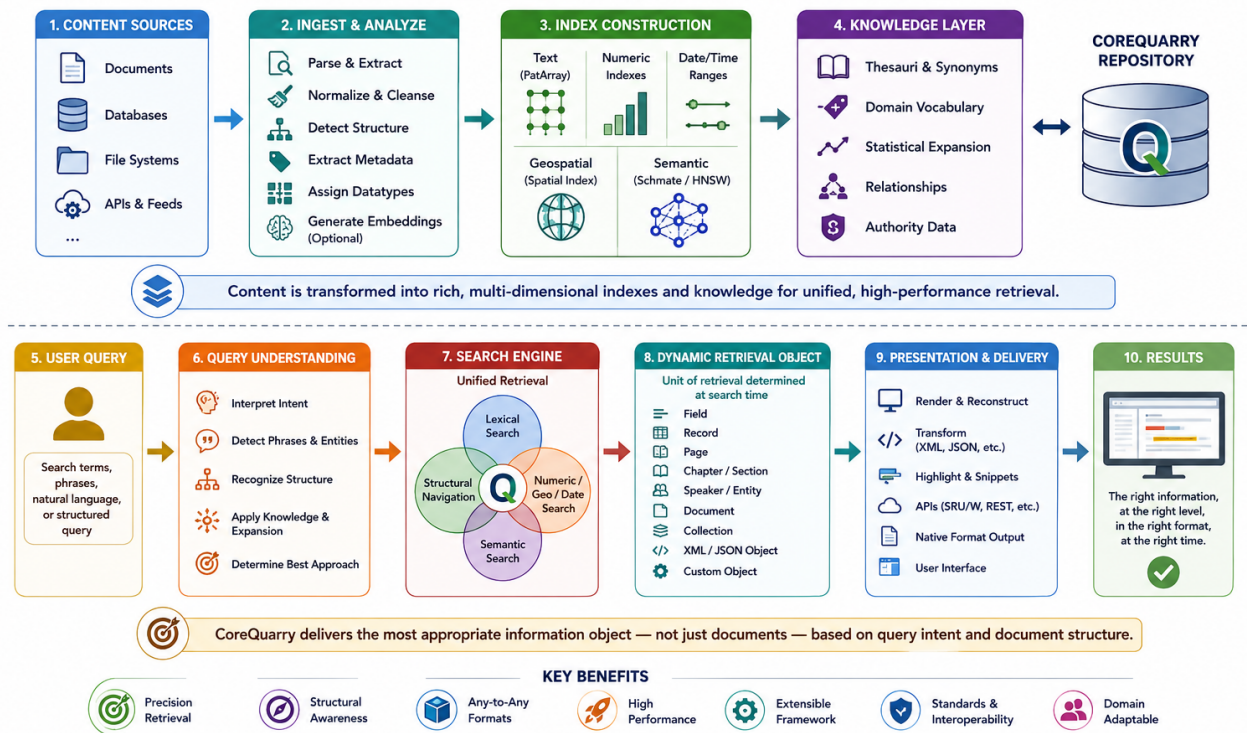
CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

beyond traditional inverted index models.

- **Positional ranking** rather than BM25-based ranking (which is just a Bag-of-Words retrieval model summing individual term contributions without concern for their position within a corpus). Because BM25 lacks positional awareness, it cannot distinguish between different meanings that depend on word order or positional context. “*Man bites dog*”—who bite whom?
- **CoreQuarry distinguishes lexical boundaries from lexical conventions.** Explicit whitespace normally separates terms; punctuation within a word, historical hyphenation, and print word-break conventions may be handled by the lexical matcher. Because those interpretations can vary, proximity is measured in persistent source coordinates rather than analyzer-generated token positions.
- **Structure-Preserving Indexing:** Unlike conventional vector databases and many RAG systems that flatten documents into chunks or JSON records, CoreQuarry preserves document structure during indexing. Sections, headings, positional relationships, metadata, object types, and document hierarchies remain available to the query engine.

CoreQuarry High-Level Process Flow

From Content Ingestion to Dynamic Retrieval and Delivery



- **Smart Query Modus:** Designed to improve the signal-to-noise ratio of lexical retrieval by identifying the results most likely to match the user’s intent, rather than returning every item containing one or more query terms. It progressively broadens the search: first looking for exact phrase matches, then for the individual words within the same content container, and finally combining the terms with OR logic. At the broadest stage, the number of matched query terms is used as a relevance signal, so results containing more of the requested concepts are prioritised. This approach aims to preserve useful recall while avoiding an excessive number of weakly related

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

results.

- Unlike existing vector databases or traditional search engines, it supports true **structure-aware queries, and typed object retrieval**, enabling precise and contextually-aware search over heterogeneous document corpora.
- **Native Document Processing:** CoreQuarry directly ingests and indexes a wide range of document formats without requiring conversion into an intermediate storage model. Structured formats such as HTML, XML, Markdown, email, and source code retain their native organisation, while rendered formats such as PDF and office documents are analysed to recover logical structure prior to indexing.
- **Typed Object Retrieval:** Documents, sections, tables, code blocks, citations, metadata records, graph nodes, and other entities can be treated as first-class searchable objects. This enables context-aware retrieval across heterogeneous collections that combine structured, semi-structured, and unstructured data.
- **Hybrid & Extensible Indexing:** Lexical, structural, numeric, geospatial, phonetic, and vector types are first-class citizens; plug-ins allow user-defined types and new algorithms without redesigning the core.
- **Efficient Incremental Indexing:** Append-only design with versioned updates, delete flags, and background garbage collection supports concurrent search and continuous ingestion, making it suitable for live AI pipelines.
- By leveraging memory-mapped, append-only indexes and a two-tier address-based caching system, the engine achieves **extremely low memory footprints** while scaling to handle complex, hybrid RAG queries **on consumer hardware, including laptops and edge devices**.

General Impact:

This engine enables privacy-preserving, local-first AI retrieval, supporting applications in legal research, medical knowledge, industrial edge AI, and large-scale document analytics. Its small footprint and high flexibility make it uniquely suited for embedded devices, laptops, and offline environments, dramatically lowering the barrier for organizations to implement robust AI-powered search without relying on cloud services.

Key Product Differentiators

- Dynamic Search Time Unit of Retrieval.
- Unified Hybrid Retrieval: Combines lexical, structural, and semantic (vector) search in a single engine.
- Native abstract model: XML, JSON, MIME, RDF, relational tables, binary formats are just expressions of underlying abstractions rather than the abstractions themselves. Documents are mapped internally into their structural essence without conversion to some baseline format (like JSON).
- Rich Query Operators: Supports advanced positional, proximity, field-specific, and relational operators beyond the features of traditional lexical indexers (such as FTS5, Lucene, Tantivy).
- Typed Object Indexes: Indexes dates, numbers, hashes, geospatial, phonetic, and vectors as first-class data types.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- Append-Only, Memory-Mapped Architecture: Enables concurrent search and continuous ingestion with minimal RAM usage (suitable even for the most basic laptops).
- Extensible & Pluggable: Supports new document types, custom fields, and alternative retrieval algorithms (HNSW, NSG, etc.).
- Local-First Deployment: Runs efficiently on laptops and edge devices, enabling privacy-preserving AI search without cloud dependencies.
- High Throughput: Demonstrated thousands of QPS on consumer hardware for complex hybrid queries.
- Extremely efficient and fast HNSW implementation with support for quantization and pass-through (optimal native support of pre-quantized models).
- We explicitly do not target raw time-series machine logging telemetry but semantic discovery, structural awareness, and AI pipeline retrieval of highly complex, high-value, unstructured and semi-structured data sets.

Figure 1. Schmate Search & Embedding Architecture

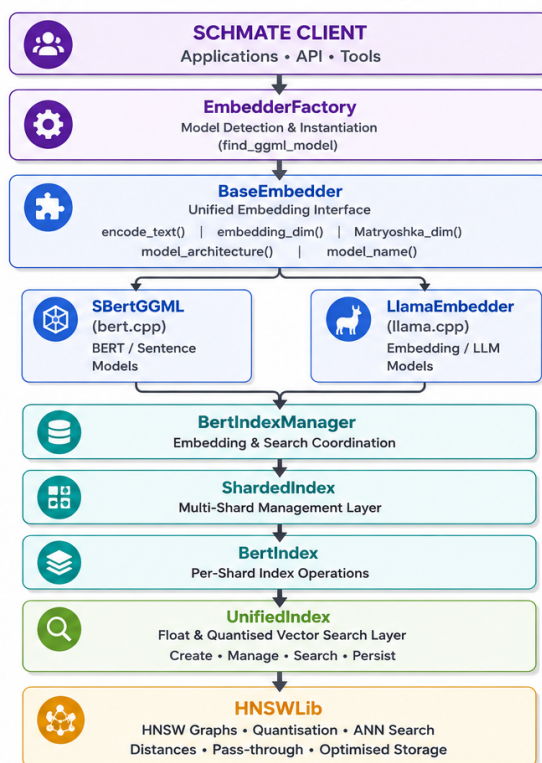
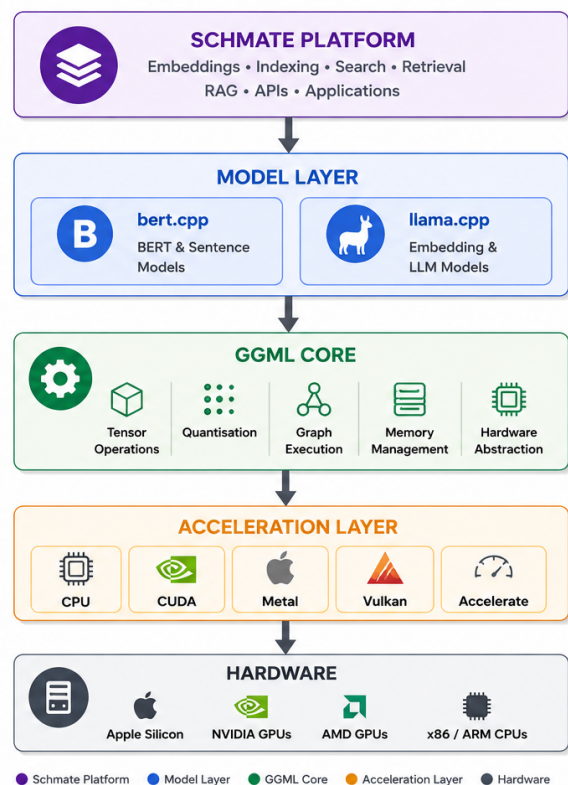


Figure 2. Schmate Runtime Stack



The system solves four major problems that current RAG and search solutions don't:

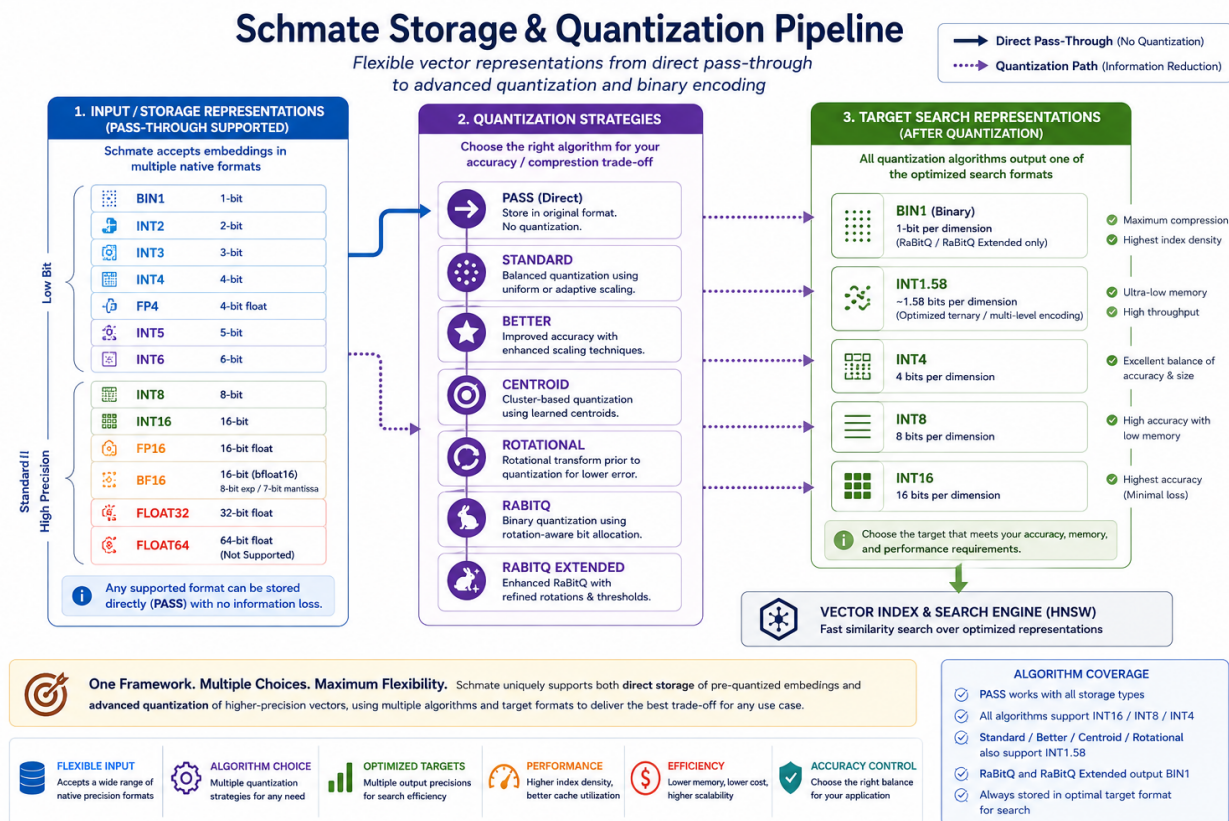
1. Hybrid Retrieval at Scale
 - Lexical + structural + semantic in a single index
 - True boolean/proximity operators and field-scoped searches

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- Optional vector embeddings for AI-enhanced relevance
- 2. Local-first & Lightweight
 - Runs fully on consumer hardware (including laptops, edge devices)
 - Lexical-only mode: <10MB RAM
 - Support for CUDA, Metal, Vulkan etc. for embeddings. Best in class performance.
 - No dependency on heavy external services (no Elasticsearch, no FAISS clusters)
- 3. Extensibility & Heterogeneous Data
 - Typed object indexes (dates, numbers, geospatial, phonetics, hashes, vectors)
 - Plug-ins for custom document formats and schemas
 - Supports structured traversal, wildcards, field/path scoping
- 4. Structure-Aware Retrieval for RAG
 - Preserves document structure rather than flattening documents into arbitrary chunks.
 - Chunking in typical systems destroys structure (paragraphs, sections, metadata, tables, speakers, chapters, records).
 - Semantic indexes can be attached to specific fields, paths, or document regions.
 - Typical systems are plagued by “Semantic collapse” because everything is embedded into one large vector space where distinctions that are obvious to humans become blurred.
 - Reduces semantic ambiguity by searching within well-defined contextual spaces
 - Improves retrieval accuracy for heterogeneous and hierarchical data
 - Enables more precise grounding for RAG applications while reducing hallucination risk.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Optimal Support of pre-Quantized Models:



Most vector databases today assume that embeddings are generated in full precision and then compressed inside the database. A typical workflow consists of:

1. **Full-Precision Embedding Generation** – An embedding model such as OpenAI's text embedding models, Cohere embeddings, or open-source Hugging Face models produces vectors in 32-bit floating-point (float32) format.
2. **Vector Ingestion** – These large float32 vectors are transmitted to the vector database and stored.
3. **In-Database Quantization** – The database applies a quantization scheme (Binary Quantization, Scalar Quantization, Product Quantization, etc.) to reduce storage requirements and accelerate similarity search.
4. **Re-ranking / Re-scoring** – Candidate results identified in the compressed space are often re-evaluated using the original float32 vectors to recover some of the accuracy lost during quantization.

This architecture was developed when embedding models primarily produced floating-point vectors.

However, the industry is increasingly moving toward models that generate compressed representations directly. Recent research and commercial systems now produce binary, low-bit, or otherwise quantized embeddings as a first-class output, eliminating the need for a separate quantization stage.

CoreQuarry fully supports the traditional database-centric workflow, including state-of-the-art quantization techniques and optional re-ranking. However, it also provides native support for pre-quantized embeddings, allowing these models to be used in their original representation without unnecessary conversion or recompression.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Advantages of Native Pre-Quantized Support

1. Reduced Storage Requirements

Traditional pipelines store the original float32 vectors alongside their quantized representations to enable accurate re-ranking. This can significantly increase storage consumption. With pre-quantized embeddings, **CoreQuarry** can operate directly on the compact representation, eliminating the need to retain large floating-point copies and substantially reducing index size.

2. Lower Ingestion Costs

In many vector databases, quantization occurs during indexing, consuming CPU resources and increasing ingestion latency. **CoreQuarry** can ingest pre-quantized vectors directly, avoiding an expensive preprocessing stage and enabling significantly faster indexing of large datasets.

3. Reduced Network and Data Transfer Overheads

A 1,536-dimensional float32 embedding requires approximately 6 KB of storage and transfer bandwidth. Equivalent binary or low-bit representations may require only a few hundred bytes. When embeddings are generated remotely, the reduction in bandwidth can be substantial, lowering both latency and infrastructure costs.

4. Improved Query Performance

Because the representation used for storage is already the representation intended for search, **CoreQuarry** avoids many of the conversion, decompression, and re-quantization steps found in conventional systems. This allows more vectors to reside in memory and improves cache efficiency, resulting in faster query execution.

5. Elimination of Quantization Mismatch

Traditional systems often quantize embeddings using generic algorithms that were not part of the model's training process. This introduces approximation errors and can degrade retrieval quality. Pre-quantized models are trained or optimized with their compressed representation in mind, meaning the generated vectors are inherently suited to low-bit search. **CoreQuarry** preserves this representation exactly, avoiding additional loss of information.

6. No Requirement for Full-Precision Re-ranking

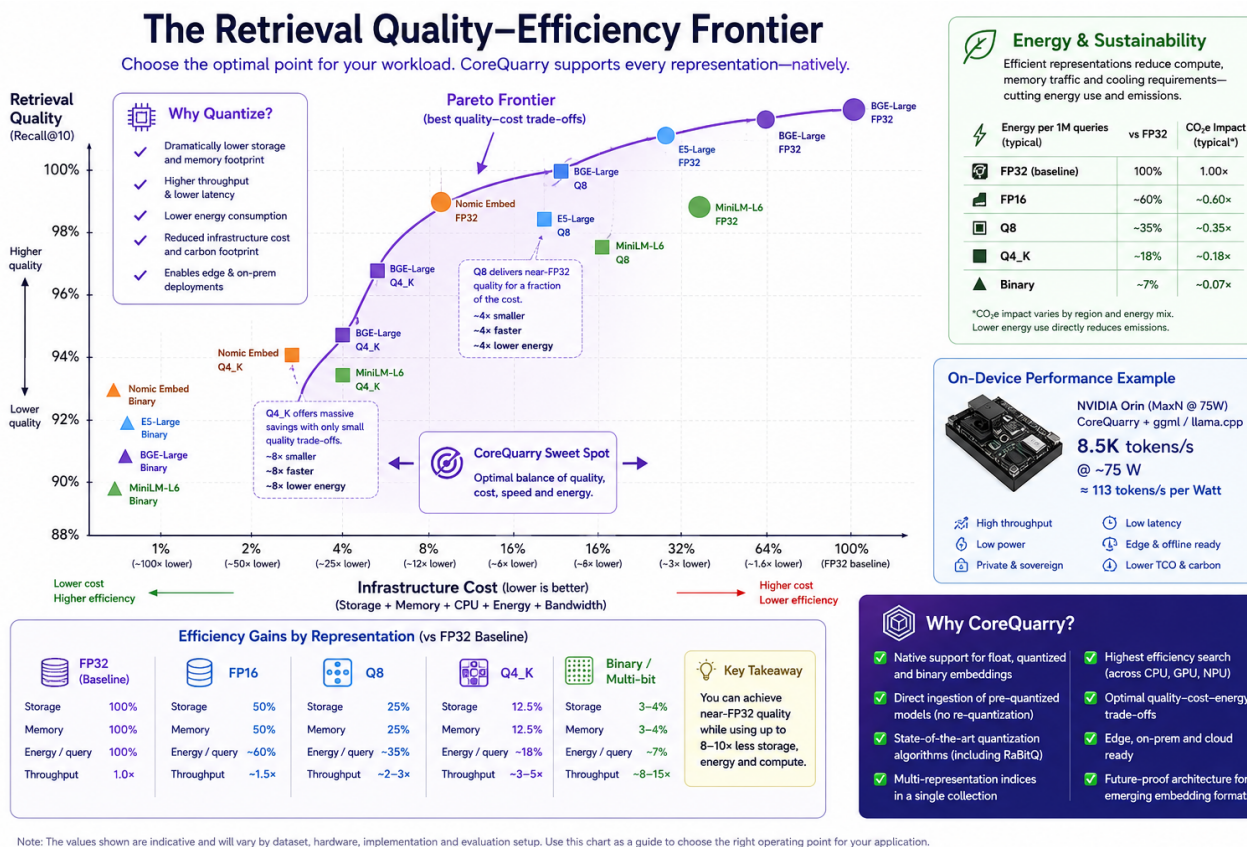
Many vector databases rely on a two-stage search process because quantization introduces enough error that final candidates must be re-scored using the original float32 vectors. When using high-quality pre-quantized models, **CoreQuarry** can often perform retrieval directly in the native representation, reducing memory consumption, storage requirements, and query latency while maintaining high retrieval quality.

7. Future-Proof Architecture

The trend in embedding research is toward increasingly compact representations, including binary embeddings, Matryoshka embeddings, low-bit embeddings, and representations specifically designed for efficient approximate nearest-neighbor search. By treating quantized vectors as first-class citizens rather than derived artefacts, **CoreQuarry** is well positioned to exploit future advances

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

without requiring a redesign of the indexing pipeline.



CoreQuarry Native Support:

- ✓ Float Embeddings
- ✓ Quantized Embeddings
- ✓ Binary Embeddings
- ✓ Pre-Quantized Models
- ✓ Future Embedding Formats

8. A More Efficient End-to-End Pipeline

Traditional vector databases follow a workflow of:

Model → Float32 Embedding → Database Quantization → Search → Float32 Re-ranking

With pre-quantized models, **CoreQuarry** can instead support:

Model → Quantized Embedding → Direct Indexing → Direct Search

This eliminates redundant computation, reduces storage and memory requirements, lowers bandwidth costs, simplifies deployment, and can improve end-to-end retrieval performance. As quantized embedding models become increasingly common, native support for pre-quantized representations provides a significant architectural advantage over systems that continue to assume float32 vectors as the primary representation.

On Premise Self-Hosting on commodity accelerator hardware versus Cloud

Many institutions operate under uncompromising mandates for data stewardship, legal compliance, and long-

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

term digital preservation. While cloud-managed database solutions offer superficial convenience, running knowledge retrieval and Retrieval-Augmented Generation (RAG) architectures wholly on-premise is the only way to protect institutional integrity. Because CoreQuarry's local pipelines are deliberately engineered to squeeze maximum throughput out of affordable, consumer off-the-shelf (COTS) systems, organizations achieve absolute data independence without the crippling capital expenditure typical of enterprise AI infrastructure.

Crucially, this hermetically sealed on-premise architecture eliminates the compliance friction inherent in a web of rigorous regulatory frameworks:

- **The General Data Protection Regulation (GDPR):** Local hosting inherently satisfies core **GDPR** mandates—such as Data Minimization (Article 5) and strict restrictions on International Data Transfers (Chapter V)—by eliminating the risk of personal data leaving sovereign borders. Cloud-hosted vector databases also routinely struggle with the **Right to be Forgotten (Article 17)** because removing data from immutable indexes, backups, and distributed cloud logs is notoriously difficult.
- **The EU Data Act:** With its strict rules on unlawful third-party access and government data requests, the EU Data Act places a heavy burden on enterprises using foreign-controlled cloud platforms. CoreQuarry keeps the entire data lifecycle within your sovereign perimeter, ensuring compliance by default.
- **The Swiss Federal Act on Data Protection (FADP):** Under the revised FADP, cross-border disclosures carry severe penalties if the destination country lacks adequate protection. By executing all semantic indexing and retrieval inside Swiss borders on local hardware, public sectors and financial institutions completely insulate themselves from foreign legal overreach like the US CLOUD Act.
- **National and Local Protection Acts:** CoreQuarry ensures strict adherence to localized statutory adaptations, such as the **German Federal Data Protection Act (BDSG)**, as well as national public record mandates like the **UK Public Records Act 1958** and the **US Federal Records Act (FRA)**. Unlike cloud-managed solutions that introduce complex multi-tenant custody chains, local execution guarantees an unbroken, audited chain of custody for public and historical records.
- **Specialized Archival and Ethical Standards:** Processing records locally natively aligns with the **International Council on Archives (ICA) Code of Ethics**, which mandates that archivists actively protect corporate and personal privacy within electronic records. By keeping semantic embeddings and document structures entirely within the institutional perimeter, organizations prevent the systemic "data leakage" that occurs when feeding proprietary metadata into external API pipelines.
- **European Security, Sovereignty, and Critical Infrastructure Requirements:** Maintaining all indexing, retrieval, and AI processing on-premise directly fulfills the escalating demands of European cybersecurity and digital sovereignty frameworks. CoreQuarry provides an immediate path to compliance with Germany's stringent **BSI IT-Grundschutz** standards and national cybersecurity frameworks across EU member states.

Crucially, for entities bound by the **NIS2 Directive** and critical infrastructure (KRITIS) regulations, reliance on foreign cloud providers introduces catastrophic supply-chain vulnerabilities. By retaining absolute control over the data, the local weights, and the execution environment, organizations drastically reduce third-party attack surfaces, radically simplify their security accreditation processes, and maintain uncompromised operational resilience during network isolation.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

By optimizing these systems for high-performance, consumer-off-the-shelf (COTS) hardware, institutions can deploy advanced local AI without requiring prohibitive capital expenditures. This approach ensures total data sovereignty and long-term digital preservation while maintaining strict adherence to public procurement budgets.

The "Hybrid-Local" Trap: Hidden Cloud Dependencies

When evaluating self-hosted knowledge retrieval systems, organizations must look beyond superficial marketing terminology to verify true data isolation. A dangerous trend has emerged where software solutions marketed as "on-premise" operate on a compromised hybrid architecture. While the core application interface or search index may reside on a local server, the system quietly relies on external cloud APIs—such as OpenAI, Cohere, or Hugging Face—for critical processing stages like text chunking, embedding generation, and vectorization.

This architecture theater is rampant even among established enterprise search platforms:

- **The Local Elasticsearch Illusion:** Organizations frequently deploy a self-managed, local instance of Elasticsearch assuming total privacy. However, to execute modern semantic or hybrid search, the standard operational path heavily promotes their **Inference API** or **Elastic Inference Service (EIS)**. Under the hood, this workflow pipes raw, local document text out to third-party cloud connect endpoints just to compute the vector representations before returning them to the local node.
- **The MongoDB Atlas Reality:** Other enterprise alternatives discard the on-premise premise entirely. **MongoDB Atlas Vector Search** is inextricably bound to a managed, cloud-hosted multi-tenant layer. The moment an organization adopts it, they are not just selecting a database; they are locking themselves into an infrastructure model that forces their proprietary data asset onto external public cloud regions subject to foreign data-custody laws.

For organizations bound by strict privacy mandates, these hidden network egress dependencies represent a critical vulnerability. Passing sensitive data to a third-party API or hosting it in a multi-tenant cloud layer completely invalidates the compliance, confidentiality, and security benefits of an on-premise deployment.

To maintain absolute data and infrastructure integrity, a system cannot rely on cloud-tethered shortcuts. The entire pipeline—document ingestion, lexical parsing, text chunking, local weight embedding generation, vector path storage, and large language model (LLM) inference—must run completely contained within the secure, local COTS perimeter.

Absolute Digital Preservation and Long-Term Stability

- **No Vendor Lock-In:** Reliance on a cloud provider risks catastrophic disruption if the vendor raises prices, alters their API, or goes out of business.
- **Immutable Infrastructure:** On-prem code bases can be frozen in time, ensuring that search and retrieval tools remain functional and identical for decades.
- **Offline Resiliency:** Libraries must function during regional internet outages or infrastructure failures; local setups ensure internal search operations never go offline.

Financial Predictability and Zero Usage Fees (FinOps)

Publicly funded institutions operate on fixed, rigid annual budgets that do not align with unpredictable cloud billing models.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- **No Per-Query API Fees:** Cloud RAG systems charge per token or per search volume, leading to volatile costs if a search tool suddenly goes viral with patrons.
- **Fixed Capital Expenditure:** Purchasing on-prem hardware is a predictable, one-time expense that fits traditional institutional grant structures and capital budgets.
- **Leveraging Existing Hardware:** Libraries can repurpose existing local server infrastructure or institutional university clusters to run highly optimized, quantized open-source models.

Uncompromising Customization and Domain Accuracy

Standard cloud search tools and commercial AI models are optimized for general web data, not specialized historical archives.

- **Niche Metadata Integration:** On-prem systems allow direct injection of highly specific library standards (like MARC21, Dublin Core, or custom taxonomies) directly into the retrieval pipeline.
- **Historical Language Tuning:** Local open-source models can be custom-tuned to read archaic spellings, dead languages, or specific regional dialects without cloud filtering mechanisms blocking the content.
- **Algorithmic Transparency:** Researchers require objective, un-biased search results; local open-source retrieval pipelines allow staff to inspect the exact retrieval math, proving how an answer was generated.

Memory Efficiency in the Era of AI Infrastructure

The rapid growth of AI workloads has placed unprecedented demands on memory capacity and memory bandwidth. While processor performance continues to improve, advances in memory density and cost reduction have slowed significantly compared to previous decades. At the same time, retrieval systems are expected to manage ever larger collections of embeddings, metadata, and graph structures.

As a result, memory has become one of the primary constraints in modern AI infrastructure. Large retrieval systems frequently require servers equipped with hundreds of gigabytes of RAM, while GPU-accelerated deployments face similar limitations in VRAM capacity. These requirements directly impact hardware costs, energy consumption, and deployment flexibility.

CoreQuarry is designed around memory efficiency as a first-class architectural objective. Native support for quantized and pre-quantized embeddings allows significantly more vectors to reside in memory, improving cache locality, reducing memory bandwidth requirements, and increasing query throughput. In many workloads, an 8-bit or 4-bit representation can reduce memory requirements by a factor of four to eight while maintaining retrieval quality close to full-precision embeddings.

The AI boom has fundamentally altered the economics of memory. DRAM prices increased dramatically during 2025–2026 as memory manufacturers shifted production toward AI-oriented products and hyperscale operators absorbed an unprecedented share of global supply. With analysts expecting tight memory markets to persist into 2027, memory efficiency is no longer merely an optimisation—it is becoming a primary architectural requirement for scalable AI systems.

By operating effectively with compact representations, CoreQuarry enables larger datasets, lower infrastructure costs, improved energy efficiency, and deployment on a wider range of hardware platforms, from embedded edge devices to enterprise-scale servers.

Sustainability and Energy Efficiency

Designed and optimized for COTS unlike enterprise data center hardware which demand three-phase

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

industrial power grids and specialized server room water chillers.

CoreQuarry is designed to enable highly capable local AI systems with a fraction of the energy consumption of conventional cloud deployments. By enabling operation at any point on the retrieval quality–efficiency frontier, CoreQuarry allows organisations to optimise not only for accuracy and cost, but also for energy consumption, deployment constraints, and digital sovereignty requirements.

Commitment to Open Infrastructure and Software Freedom

CoreQuarry is built on a foundation of genuine open-source development, distributed under the permissive **Apache License 2.0**. All core functionality—including high-performance hybrid indexing, localized vector operations, and structural document parsing—is available under this single license. **CoreQuarry** explicitly rejects the industry trends of proprietary extensions, artificial feature-gating, dual-licensing traps, or "open-core" models designed to force users into commercial clouds. The community code is the enterprise code; users are entirely free to inspect, modify, redistribute, and deploy the platform without commercial dependencies or vendor lock-in.

This approach reflects an unyielding architectural principle: critical knowledge infrastructure must remain transparent, fully auditable, and under the absolute control of its operators. For archives, libraries, national research institutions, public-sector organizations, and enterprises, this eliminates the existential risk of license shifts and ensures that deployments can be maintained independently over decades.

At a time when many platforms leverage the marketing goodwill of "open source" while relying on proprietary cloud services, restrictive "source-available" licenses, or compute-gated features, **CoreQuarry** remains fiercely committed to a fully transparent development model. Every essential capability required to achieve elite performance on local COTS hardware is delivered uncompromised to the entire community.

Potential Markets and Applications

CoreQuarry addresses a broad spectrum of retrieval and knowledge-management challenges, ranging from individual AI assistants to large-scale enterprise and public-sector infrastructure. Its combination of hybrid search, graph-aware retrieval, quantization support, and efficient deployment enables adoption across multiple market tiers.

Tier 1: Immediate Markets

These represent the most accessible and rapidly growing opportunities, where organisations are actively deploying Retrieval-Augmented Generation (RAG) systems and searching for alternatives to complex vector database stacks.

Local AI Assistants and Private Knowledge Systems

Organisations increasingly wish to deploy AI assistants over sensitive internal data without exposing information to external cloud providers. **CoreQuarry** enables local and on-premise RAG systems capable of searching contracts, technical documentation, research papers, correspondence, and internal knowledge bases.

Potential users include law firms, medical practices, consulting companies, engineering firms, researchers, and small-to-medium enterprises seeking secure AI capabilities.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Enterprise Knowledge Infrastructure

Many enterprises currently rely on a combination of search engines, vector databases, ETL pipelines, metadata stores, and orchestration frameworks. These multi-system architectures are expensive to deploy and maintain.

CoreQuarry offers a unified retrieval platform combining semantic search, keyword search, metadata filtering, graph traversal, and structured querying within a single engine, reducing complexity while improving performance and operational efficiency.

Archival, Compliance, and Records Management

Organisations responsible for large collections of historical, legal, financial, or regulatory information require retrieval systems that understand both document content and document relationships.

CoreQuarry supports semantic discovery alongside metadata, temporal information, versioning, and graph structures, making it suitable for archives, compliance systems, standards repositories, legislative collections, and corporate records management.

Tier 2: Emerging Growth Markets

These markets are experiencing rapid growth as organisations seek more efficient, private, and sovereign AI infrastructure.

Sovereign and On-Premise AI

Governments, public institutions, defence organisations, and regulated industries increasingly require AI systems that operate entirely within their own infrastructure. **CoreQuarry** enables high-performance retrieval without dependence on external cloud services, supporting digital sovereignty, security, and regulatory compliance requirements.

Scientific Research and Data Discovery

Research organisations must search across publications, datasets, software repositories, laboratory records, and experimental results. **CoreQuarry**'s ability to combine semantic retrieval with structured metadata and graph relationships makes it particularly attractive for research infrastructures and scientific knowledge systems.

Potential users include universities, digital humanities projects, pharmaceutical research organisations, and national research infrastructures.

Cultural Heritage and Digital Collections

Museums, libraries, archives, and cultural institutions are increasingly adopting AI technologies to improve access to digitised collections. **CoreQuarry** supports semantic exploration of historical corpora while preserving provenance, metadata relationships, and document structure.

Applications include manuscript collections, newspaper archives, historical records, museum catalogues, and digital heritage platforms.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Tier 3: Strategic and Future Markets

These opportunities align with long-term trends in AI deployment, efficient computing, and embedded intelligence.

Edge AI and Embedded Intelligence

The proliferation of AI-capable edge hardware creates demand for retrieval systems that can operate with limited memory, storage, bandwidth, and power consumption.

CoreQuarry's support for quantized and pre-quantized embeddings enables deployment on compact hardware platforms including industrial controllers, robotics systems, autonomous devices, field equipment, and edge AI appliances.

AI-Native Applications and Developer Platforms

Just as SQLite became the default embedded database for applications, there is an emerging need for an embedded retrieval engine that developers can integrate directly into software products.

CoreQuarry provides semantic retrieval, hybrid search, graph navigation, and RAG capabilities without requiring a separate search infrastructure, enabling a new generation of AI-native applications.

Potential applications include personal knowledge management systems, software development assistants, customer support tools, document management platforms, and intelligent productivity software.

Sustainable and Energy-Efficient AI

As AI adoption expands, energy consumption is becoming an increasingly important economic and operational consideration. **CoreQuarry**'s efficient indexing, compact representations, and support for low-power hardware reduce infrastructure requirements while maintaining retrieval quality.

This enables high-performance AI systems to run on modest hardware, reducing operational costs and extending deployment options beyond traditional cloud environments.

Next-Generation Retrieval Systems

The retrieval ecosystem is rapidly evolving toward binary embeddings, low-bit representations, multimodal search, graph-enhanced retrieval, and domain-specific embedding models.

CoreQuarry is designed as a future-proof retrieval platform capable of supporting emerging representations and search paradigms, allowing organisations to adopt new AI technologies without replacing their retrieval infrastructure.

Long-Term Vision

CoreQuarry aims to become foundational retrieval infrastructure for AI applications in the same way that SQLite became foundational infrastructure for embedded data management: lightweight, efficient, deployable anywhere, and capable of scaling from individual applications to enterprise and institutional knowledge systems.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Copyrights / Acknowledgements (a long history)

Copyright (c) BSn/Munich 1992-94

Copyright (c) MCNC, Clearinghouse for Networked Information Discovery and Retrieval, 1994-2001.

Copyright (c) BSn/Munich 1995-2011

Copyright (c) A/WWW Enterprises, Crownsville, MD, 1995-2001.

Copyright (c) NONMONOTONIC Networks 2011-2020

Copyright (c) 2020-21 Edward C. Zimmermann and the re-iSearch project.

Copyright (c) 2021-24 ExoDAO Network Association (Zurich)

Copyright (c) 2024-25 re-Isearch Org

Copyright (c) 2026 Edward C. Zimmermann (corequarry.com) and NONMONOTONIC Inc (in Formation).

This product includes software developed by Yury Malkov and the NMSLIB project contributors

(<https://github.com/nmslib/hnswlib>). Original Copyright (c) 2017 Yury Malkov licensed under the Apache License, Version 2.0.

This product also includes software (from the ggml tensor library) developed by Georgi Gerganov and the ggml-org contributors (<https://github.com/ggml-org/ggml>). Original Copyright (c) 2023-2024 Georgi Gerganov licensed under the MIT License.

The ODF support is (c) 2023 Svante Schubert (as part of a project supported by NGI-Zero/Nlnet). It is released under the MIT License.

This project implements the RaBitQ quantization algorithm based on the research by Jianyang Gao and Cheng Long: "RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search" (SIGMOD 2024).

Support:



Through the [NGI0 Discovery Fund](#), a fund established by [NLnet](#) with financial support from the European Commission's [Next Generation Internet](#) programme, under the aegis of [DG Communications Networks, Content and Technology](#) under grant agreement No [825322](#).

Through the [NGI0 Commons Fund](#), a fund established by [NLnet](#) with financial support from the European Commission's [Next Generation Internet](#) programme, under the aegis of [DG Communications Networks, Content and Technology](#) under grant agreement No [101135429](#). Additional funding is made available by the [Swiss State Secretariat for Education, Research and Innovation](#) (SERI).

Through a Grant from the Bundesministerium für Forschung, Technologie und Raumfahrt (Germany) GRANT_NUMBER: [01IS22S32](#) (exploring support of the IPFS and supporting remote indexing). Through a grant from the European Commission Coordination and Support Action (CSA) on ICT standardisation (extending support for additional post ISO-8601:2019 features)

Additional support was provided by a grant from OpenData CH/Mercator Foundation CH.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Appendix I: Technology Backgrounder / The 30 year old new kid

In the contemporary landscape of AI infrastructure, there is a pervasive illusion that cutting-edge capabilities require brand-new, hyper-complex software stacks. The enterprise market has been flooded with bloated vector databases and heavily layered search platforms that rely on thousands of brittle cloud dependencies just to move text and floating-point numbers across a network.

CoreQuarry rejects this paradigm. It is built on a simple, contrarian truth: the most effective way to solve the modern localized AI challenge is not to build more abstractions, but to return to high-performance, hardware-sympathetic systems engineering.

Architecturally, CoreQuarry is quite literally a thirty-year-old new kid. It is a hybrid knowledge retrieval engine that emerged in 2026 as the modern evolution of **Isearch**—the pioneering open-source text engine of the mid-1990s co-developed alongside the Clearinghouse for Networked Information Discovery and Retrieval (**CNIDR**). Originally engineered to solve the foundational "document type" problem of the early World Wide Web, this deep architectural pedigree was reborn as **re-Isearch** through the long-term support of **NLnet**, as well as pivotal grants from the German Federal Ministry for Research, Technology and Space (**BMFTR**), **Open Data Switzerland**, and the **Mercator Foundation**.

Through **Project Schmate** (שמחטש), this battle-tested infrastructure evolved to integrate advanced vector indexing directly alongside its historic lexical core. However, preserving the past did not mean ignoring the present. The core engine has been re-engineered to exploit the exact mechanical realities of contemporary edge computing platforms:

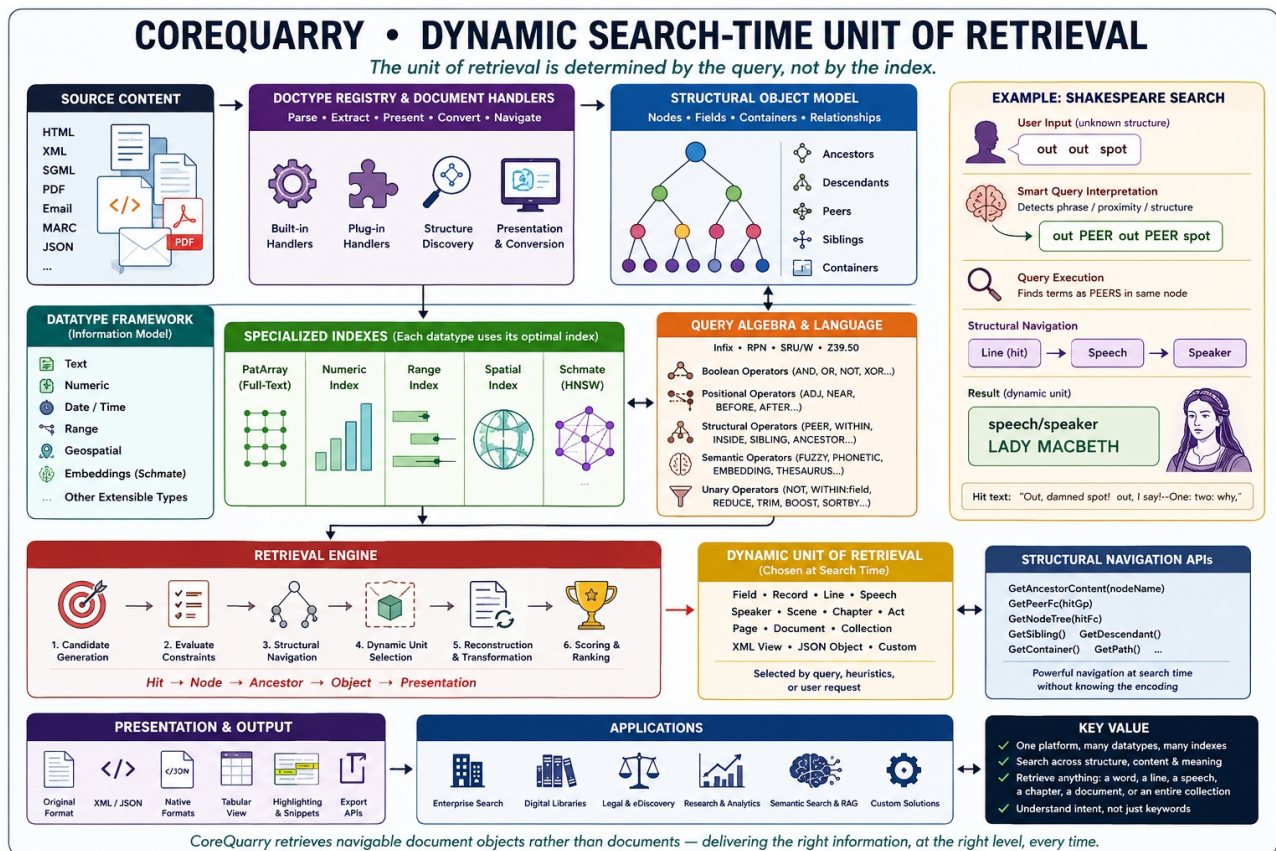
- **Hardware-Native Vectorization (SIMD):** Rather than letting the CPU sit idle while waiting for heavy GPU resource allocations, CoreQuarry utilizes deeply optimized Single Instruction, Multiple Data (SIMD) compiler pathways. By mapping data layouts directly to native CPU execution blocks—including AVX2, ARM Neon, and SVE2 architectures—the engine handles lexical term sorting and arithmetic array evaluations at the physical limits of the processor.
- **NVMe Parallelism & Memory Mapping:** Traditional cloud databases choke on the network bottlenecks of remote block storage. CoreQuarry bypasses the OS storage overhead entirely by treating local, high-speed NVMe flash drives as native extensions of system memory. Using parallelized, append-only, memory-mapped I/O (mmap), index traversal achieves near-zero latency, pulling document segments off local solid-state drives instantly without suffocating host resources.
- **Cache-Conscious Data Layouts:** Modern AI search bottlenecks are often bound by memory bandwidth, not raw processing power. CoreQuarry organizes its internal structures to optimize L3 cache lines and maximize the massive bandwidth of contemporary high-speed DDR5 and unified RAM architectures. By ensuring data density and deterministic layout patterns, it minimizes expensive CPU cache misses.

Through **Project Schmate** (שמחטש), this battle-tested infrastructure evolved to integrate advanced vector indexing directly alongside its historic lexical core. By combining three decades of production-proven document parsing and high-throughput text retrieval with modern, localized semantic routing, CoreQuarry circumvents the resource bloat of cloud-native infrastructure. Instead of hiding behind layers of virtual machines or cloud-tethered microservices, CoreQuarry's native systems layout interfaces directly with physical hardware.

By matching these dense, deterministic data structures with the precise mechanical realities of local processors and commodity accelerator hardware, a design philosophy rooted in the dawn of open-source internet infrastructure becomes the definitive architectural breakthrough for the next era of local-first digital

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

sovereignty. It is thirty years of engineering expertise, purpose-built to mine the core of human knowledge.



Our philosophical ancestor is early SGML retrieval systems (like OpenText which grew out of a University of Waterloo project from Timothy Bray, Frank Tompa, and Gaston Gonnet to digitize the Oxford English Dictionary) but designed to support beyond SGML multiple document formats and data types; and modernized for:

- embeddings
- local-first compute
- RAG

For vector search we use the The Hierarchical Navigable Small World (HNSW) algorithm. Introduced in 2016 by Yu. A. Malkov and D. A. Yashunin in their paper, "*Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*". It is the evolutionary successor to the Navigable Small World (NSW) graph, combining it with the hierarchical structure of Skip Lists.

Both our algorithms for lexical and vector structures were designed to move beyond simple linear scans. HNSW achieves logarithmic complexity $O(\log N)$ for vectors, while PatArrays provide similar efficiency for finding any substring in a massive corpus without an inverted index.

The Industrial Metamorphosis: 1995 vs. 2026

To understand why a local-first architecture is viable today, one must look at the staggering, exponential shift in the underlying economics of physical hardware. Hardware has progressed dramatically since the mid-1990s when early SGML open-source retrieval systems were first deployed.

In 1995, a high-end enterprise server (such as a Sun Ultra Enterprise, IBM RS/6000 SP, HP 9000, or a multi-socket Intel Pentium Pro server) was a massive, room-hogging capital expenditure. A fully configured system capable of indexing large corporate document pools typically featured:

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- **Volatile Memory:** 1 to 4 GB of EDO or early ECC DRAM, creeping along a 60–66 MHz memory bus with a peak bandwidth of just 200–500 MB/s and a high latency of 60–80ns.
- **Storage Arrays:** Massive external SCSI disk arrays holding a total of 50 to 500 GB (such as a flagship EMC Symmetrix holding ~400 GB), built from individual mechanical spindles carrying 2 to 18 GB each, mediated by a RAID controller with a tiny 32–256 MB write cache.

In the mid-1990s, buying this level of compute, storage, and mandatory database licensing demanded an institutional budget of \$500,000 to \$3,000,000+ (which translates to roughly \$1M to \$6M in inflation-adjusted dollars).

The Numbers Side-by-Side

Metric	Mid-1990s Enterprise	Modern 2026 Workstation	Improvement
RAM capacity	1–4 GB	128 GB–2 TB	~100–500×
Memory bandwidth	~400 MB/s	~300 GB/s	~750×
Memory latency	~70 ns	~75 ns end-to-end	Roughly flat
Disk seq. read	10–15 MB/s	12,000–14,000 MB/s	~1,000×
Disk random IOPS	100–200	1,000,000–2,000,000	~10,000×
Disk seek/latency	8–12 ms	0.02–0.1 ms	~100–500×
Storage capacity	400 GB (array)	4–8 TB (single drive)	~10–20× per drive
CPU clock	100–500 MHz	4,000–6,000 MHz	~10–60×

Today, a fast consumer desktop or a 64-core high-end workstation costs between \$2,000 and \$9,000, while outperforming a 1990s data center by multiple orders of magnitude.

Even the discontinued entry-level **Apple Mac mini (M4)**—which had cost a mere **€475 (Netto)**—delivered 38 TOPS of local AI performance, 120 GB/s of memory bandwidth, and 16 GB of unified RAM. Its being updated to the **M6 mini** with 170 GB/s bandwidth for \$900 (reflecting also the current increase in RAM prices). Performance of the M6 mini is expected to deliver around 15-35% better indexing performance than the M4 it replaces.

For specialized AI workloads the **NVIDIA DGX Spark**, powered by the **GB10 Grace Blackwell** packs a phenomenal 1 PetaFLOP of AI throughput, 128 GB of unified memory @273 GB/s , and a 4 TB NVMe SSD directly onto a desktop for roughly **€4,349 to €4,790**. The forthcoming **M5 Studio Ultra** has **1.2TB/s** of unified memory bandwidth is priced, similarly, at around **\$5000**. It is expected to deliver a good as much as 4x the performance (on CoreQuarry’s ggml backend) than the GB10 despite having only 140 TLOPS/s.

Example of an Enterprise Ready Specification (Prosumer Class):

Component	Target Specification	Institutional Rationale	Estimated Cost (USD)
Graphics Processing Units (GPUs)	2x NVIDIA GeForce RTX 5090 (32GB GDDR7 VRAM per card)	Provides a massive 64GB of combined VRAM pool over a high-bandwidth PCIe interface. This capacity natively holds a 70B parameter model at Q4_K_M quantization with enough residual VRAM to support expansive context windows and user sessions.	\$4,000 – \$4,600
Processor (CPU)	AMD Ryzen Threadripper 7960X (24 Cores / 48	Provides the raw parallel processing power required for the document ingestion, optical character recognition (OCR), text tokenization,	\$1,300 – \$1,500

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

	Threads) or high-end Intel Xeon W-2400	and vector indexing phases of a RAG pipeline.	
System Memory (RAM)	128GB to 256GB DDR5 ECC (4x 32GB or 4x 64GB Multi-Channel Kits)	Houses the runtime footprint of the vector database (e.g., Qdrant, Milvus) and provides a secure memory buffer for processing massive archive batch imports. Error-Correcting Code (ECC) RAM prevents memory leaks and bit-flips during multi-day background indexing tasks.	\$600 – \$1,000
Storage (SSD)	4TB NVMe M.2 SSD (PCIe Gen 5, e.g., Crucial T700 or Samsung PRO series)	Yields high-speed read/write performance (>12,000 MB/s). Essential for instantaneous zero-copy memory mapping (mmap) of multiple GGUF files and quick structural queries into the local vector index.	\$350 – \$500
Motherboard & Power	Workstation Motherboard (WRX90 / PCIe Gen 5 x16/x16 split) + 1600W 80+ Titanium PSU	Requires native physical spacing and dual x16 lane routing to prevent GPU bottlenecking. The industrial-grade power supply unit guarantees continuous power stability under heavy, concurrent processing loads.	\$800 – \$1,100
Cooling & Chassis	High-Airflow E-ATX Workstation Case + Premium Liquid/Air Cooling Systems	AI inference generates intense thermal output over prolonged periods. High-static pressure fans prevent hardware throttling, ensuring deterministic system speeds during heavy usage sessions.	\$300 – \$500
Total Estimated Capital Expense		Enterprise-grade on-premise infrastructure safely under budget limits.	\$7,350 – \$9,200

This specification allows the institution to host a large-scale, quantized 70B parameter model (such as Llama 3 70B or Qwen 3 70B) natively alongside an embedding model.

Form Factor: Housed in a standard standalone tower or a compact 4U rackmount chassis, avoiding the need for dedicated data-center space rental.

What Contemporary Hardware Means for Information Retrieval

The most dramatic difference in modern hardware isn't raw, aggregate clock speed—it is the obliteration of random I/O latency and the massive scaling of Input/Output Operations Per Second (IOPS). This is the exact bottleneck that traditional database queries and information retrieval (IR) engines hammer the hardest.

- **The Death of I/O Wait:** A sequential full-table or corpus scan that took up to ten minutes on a 1990s SCSI disk array now takes seconds on a local NVMe drive.
- **The Dissolution of the Seek-Time Tax:** Consider a complex database query requiring 10,000 scattered, random disk reads. On a 1990s enterprise machine, the processor sat idle for 80 to 120 seconds just trapped in I/O wait while mechanical drive heads physically moved. A modern consumer NVMe drive satisfies those exact 10,000 random reads in roughly 10 to 50 milliseconds.
- **The Vanishing Buffer Pool:** Entire operational datasets that once demanded aggressive, highly defensive buffer pool management—because volatile memory was an expensive commodity—now fit entirely within the local RAM of a low-cost workstation. For hot data, physical disk I/O has been eliminated entirely.
- **Massive Vectorized Pipeline Velocity:** Due to exponential leaps in memory bus architectures, modern in-memory analytics and columnar scanning paths can process data roughly 750 times faster than their 1990s ancestors, even when nominal latency metrics look similar.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- Memory bandwidth improvements mean in-memory analytics (columnar stores, etc.) can process data $\sim 750\times$ faster even if latency is similar

Database administrators of the 1990s spent enormous intellectual and operational capital on micro-tuning disk access—index alignment, meticulous block caching, and elaborate hardware disk striping—precisely because a single random read penalised execution by 10 milliseconds. On modern hardware, that entire problem domain has evaporated.

The primary structural bottleneck has migrated completely from the local storage interface to the **CPU cache line** and **coherency in memory access patterns**.

This technological leap exposes why standard text-retrieval designs—most notably the classic, pure inverted index popularized by legacy search architectures—are no longer appropriate as a sole engine foundation. The inverted index is obsolete! Their historical advantage (accepting their significant limitations) makes no sense in 2026.

The inverted index was an engineering response to hardware constraints, never an ideal data structure:

- Disk was slow and expensive— you couldn't afford to scan raw text
- RAM was tiny— you couldn't hold documents in memory
- IOPS were precious— every random read cost 10–12ms
- So you precomputed posting lists to minimize disk seeks at query time

The entire architecture mantra of Lucene, early Google, and most 1990s IR systems was essentially: *"disk is our enemy, precompute everything possible to avoid touching it."*

Inverted indexes were explicitly designed to avoid reading documents. They create highly fragmented, sparse term-postings lists scattered across storage to allow a sluggish 1990s CPU to jump directly to match coordinates. However, when these sparse, highly fractured pointer chains are executed on modern multi-core processors, they result in catastrophic CPU cache misses and choke the pipeline with memory latency.

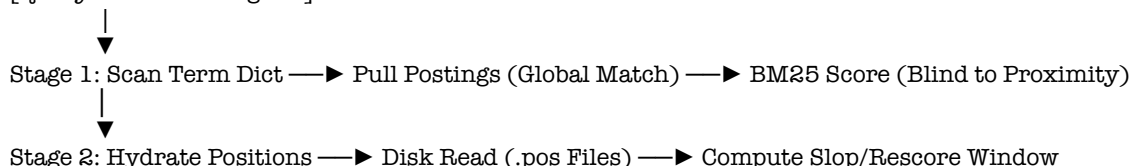
While this model historically delivered high performance for basic keyword lookups, it achieved speed through also a destructive process: collapsing multi-dimensional text hierarchy into flat arrays, completely stripping away data geometry. Document topology becomes invisible and structurally blind. To extract some minimal context architects must rely on lossy, manual workarounds:

- **Position Increment Gaps:** Artificial token offsets (e.g., jumping +100 positions at a newline) are injected to simulate paragraph boundaries.
- **Inline Payloads:** Arbitrary byte metadata arrays are baked directly into individual token postings lists to tag structural origins at the word level.

This approach creates a severe structural deficiency. The search engine does not actually understand hierarchy; it merely calculates numerical distances ("slop") between pointer addresses. It cannot naturally distinguish whether two terms reside within the same sentence, the same speech bubble, or entirely separate paragraphs without calculating complex token arithmetic at query time.

Because an inverted index reduces text retrieval to a flattened vocabulary, it is fundamentally bound to a Bag-of-Words scoring model (like BM25) during its primary execution phase. This design creates an expensive, multi-stage retrieval bottleneck when handling complex queries.

[Query: "database engine"]



CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Distributed full-text engines frequently market themselves as "schema-less NoSQL document stores." In practice, this is an abstraction. The underlying inverted index forces a highly rigid Schema-on-Write lifecycle. The moment a document is ingested by an inverted index, its tokenization pattern, stemming rules, position offsets, and payload compressions are permanently baked into immutable segments on disk. If an organization's querying requirements evolve—such as transitioning from broad keyword lookups to strict paragraph-bounded constraints—the query layer cannot adapt dynamically. The engineering team must refactor the write-time index analyzers, alter the global schema mapping, and execute a computationally expensive, time-consuming full re-index of the entire dataset.

PatArrays, by contrast, enable true operational agility by supporting Schema-on-Read structural execution. Because the document's geometric topology remains intact inside the engine (just addresses), users can dynamically apply, modify, or pivot structural constraints at query time. A search can be restricted to specific element paths, localized zones, or hierarchical depths on the fly, without modifying the underlying data on disk or triggering index rebuilds.

Despite their advantages, however, Patricia Trees and Pat Arrays fell Out of Favor in the 1990s because of hardware:

- Pointer chasing is catastrophically cache-unfriendly— every node traversal is a potential cache miss
- On 1990s hardware, a cache miss meant going to main memory at 60–80ns, and with tiny caches (L1 was 8–16KB) misses were constant
- The inverted index, despite its bulk, had more predictable, sequential access patterns that could be optimized with careful disk layout
- Patricia trees also consumed significant RAM for pointers — precious in an era of \$50,000/GB RAM
- Worst case pointer density: a tree over a million terms might have millions of pointer pairs, each potentially a cache miss during traversal

So Patricia trees were theoretically elegant but practically punished by memory hierarchies of the era.

Cache hierarchies have transformed things

Cache	1995 Size	2026 Size	Improvement
L1	8–16 KB	32–64 KB	~4×
L2	256 KB (rare)	512 KB–1 MB per core	~4–8×
L3	Nonexistent/tiny	64–256 MB (Threadripper)	∞ → huge
RAM	256MB–4GB	128GB–2TB	~500×

A Patricia tree over a large IR vocabulary (~1–10 million terms) occupies perhaps 500MB–2GB of RAM. On a modern system this fits entirely in L3 cache on high-end CPUs, or at worst in RAM with DDR5's enormous bandwidth.

When the whole tree lives in L3, pointer chasing is no longer going to slow DRAM — it's hitting ~10 cycle L3 latency vs the ~200 cycle DRAM penalty that killed Patricia trees in the 1990s.

PAT Array Problem (1992)	2026 Solution
Index too large for RAM	128GB–2TB RAM — fits comfortably
O(log n) random accesses slow	DDR5 random access ~75ns, L3 ~10ns — negligible
Construction too slow	Modern O(n) suffix array construction (SA-IS algorithm) + fast CPUs

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Binary search cache-unfriendly

Entire index in L3/RAM — cache misses collapse

No word-level ranking (BM25 etc.)

Can layer neural/embedding ranking on top

Our indexing architecture builds on ideas related to Patricia trees and Pat arrays, but extends them to address the I/O and scalability constraints that have historically limited their use in large-scale information retrieval. A Patricia tree provides an efficient compressed trie structure for prefix searches, but it requires pointer-heavy navigation and does not map well to modern memory-mapped storage or high-throughput disk access. Pat arrays, by contrast, store lexicographically ordered suffix or term references in contiguous arrays, improving locality but traditionally requiring the underlying corpus text to be stored alongside the index and incurring expensive disk reads during lookup. Our approach separates corpus storage from the index while maintaining compact addresses.

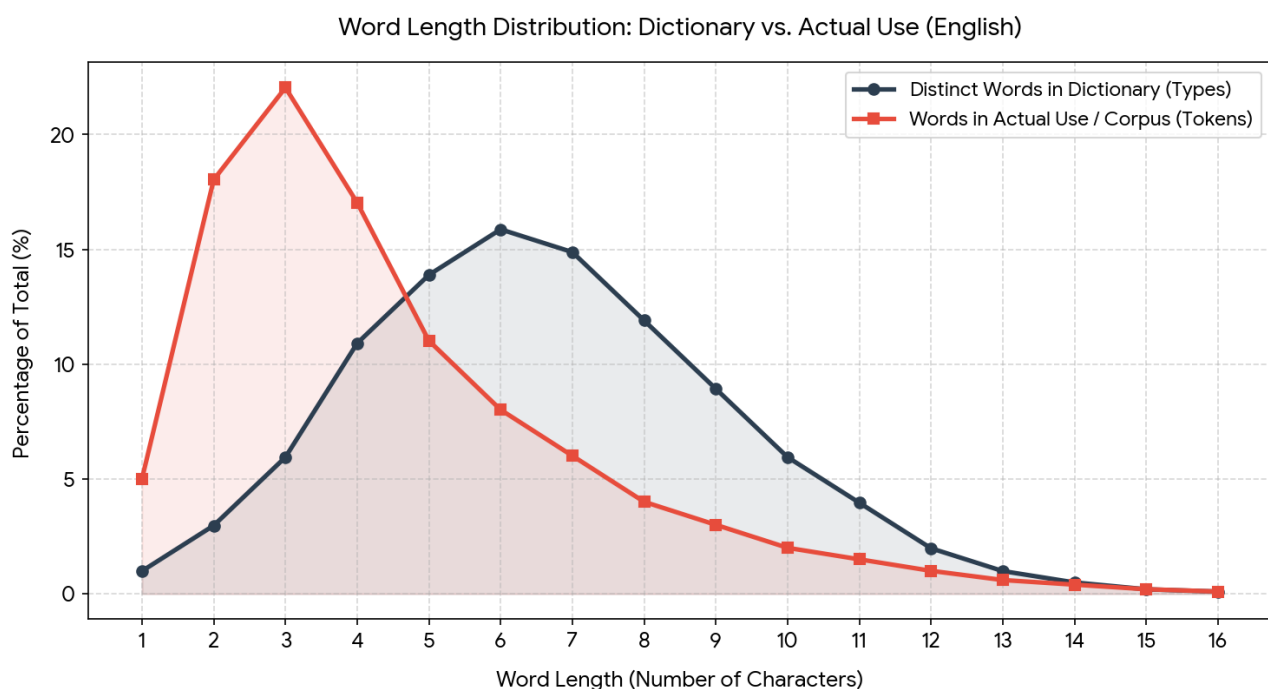
Caching Strategy based on Word Length Distribution

Information Efficiency: Human communication is optimized for speed. According to **Zipf's Law of Abbreviation**, the words we repeat most frequently are compressed by evolution into the shortest possible forms

Distinct Words (Types) follow a **unimodal distribution** (a hump-shaped curve). The vast majority of dictionary words are moderately long (typically 6–9 characters in English) because a language needs permutations of letters to create a unique vocabulary. Very short words are rare because there are only so many combinations you can make with 1 to 3 letters

Words in Use (Tokens) follow a **monotonic decrease or highly right-skewed distribution**. When analyzing a real book or corpus, massive frequency is dominated by short grammatical words ("the", "of", "and", "to"). This causes a huge spike at lengths 2–4, dropping sharply as words get longer.

Grammar vs. Content: Short words (2–4 letters) act as structural or grammatical "bridges" (e.g., pronouns, prepositions, articles). We reuse them constantly in every sentence, whereas long words (8+ letters) carry specific technical or descriptive content and are only used when strictly necessary



Our 2-tier cache, originally introduced years ago in our proprietary fork, allows the advantages of the separate corpus but removes the need to read it for words less than some pre-set prefix length.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

This works remarkably well especially in English

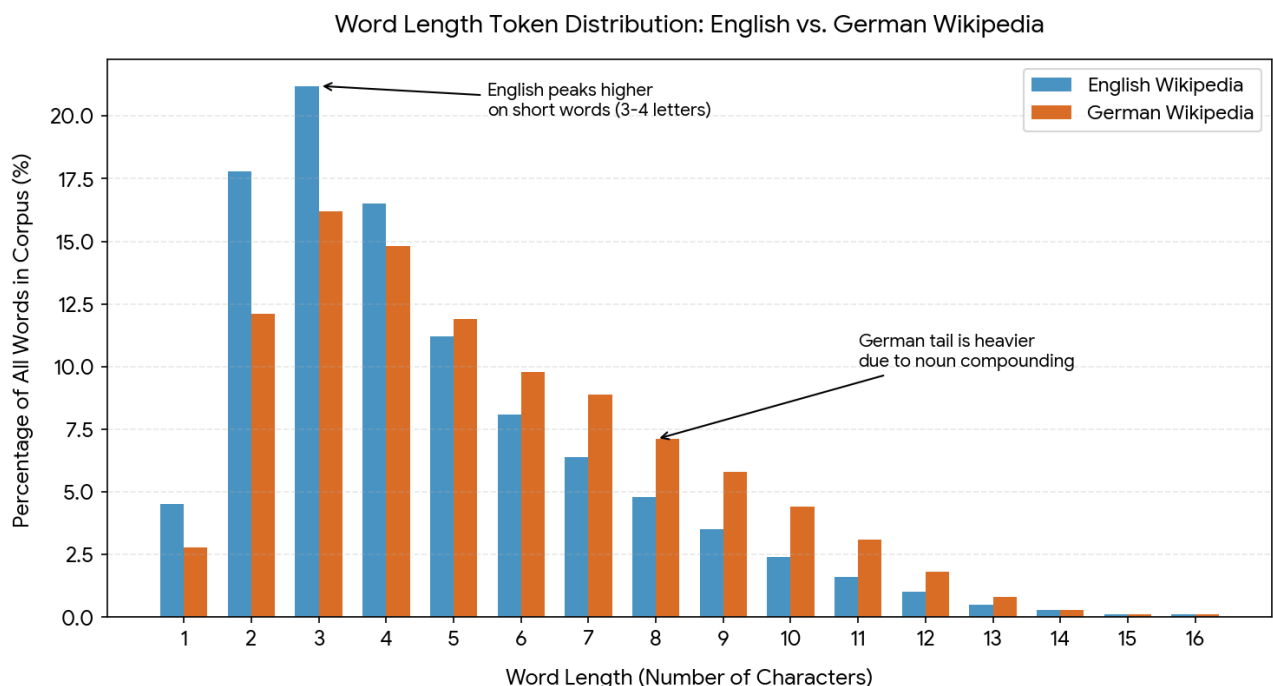
- 1–7 letters: ~82.97%
- 8–10 letters: ~13.46%
- 11–14 letters: ~3.46%
- 15 letters: ~0.076%
- ≥16 letters: ~0.034% combined

≈ **99.9% of words are under 15 characters long** in typical English text. **10 characters or less: ~96.4%**. A 10-character prefix is long enough that collisions are uncommon, so the resulting count will be close to the original dictionary size.

So even with a length set to 10, 96% of all words won't demand a corpus lookup. This cache is also relatively small since it would require only 26 bytes per unique word (for 64-bit indexes).

Source	Count
Words ≤10 chars	~118k
Unique prefixes from longer words	~44k
Total unique entries	~162k

≈150,000 – 170,000 max unique entries so we can safely assume that the maximum size of the cache will be under 4.5 MB.



German behaves very differently from English because of productive compounding (e.g., *Krankenhausverwaltungsgesetz*). So truncating to 10 characters collapses many more words.

Length	Approx. share
≤10 letters	~40–50%
11–15 letters	~30–35%

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

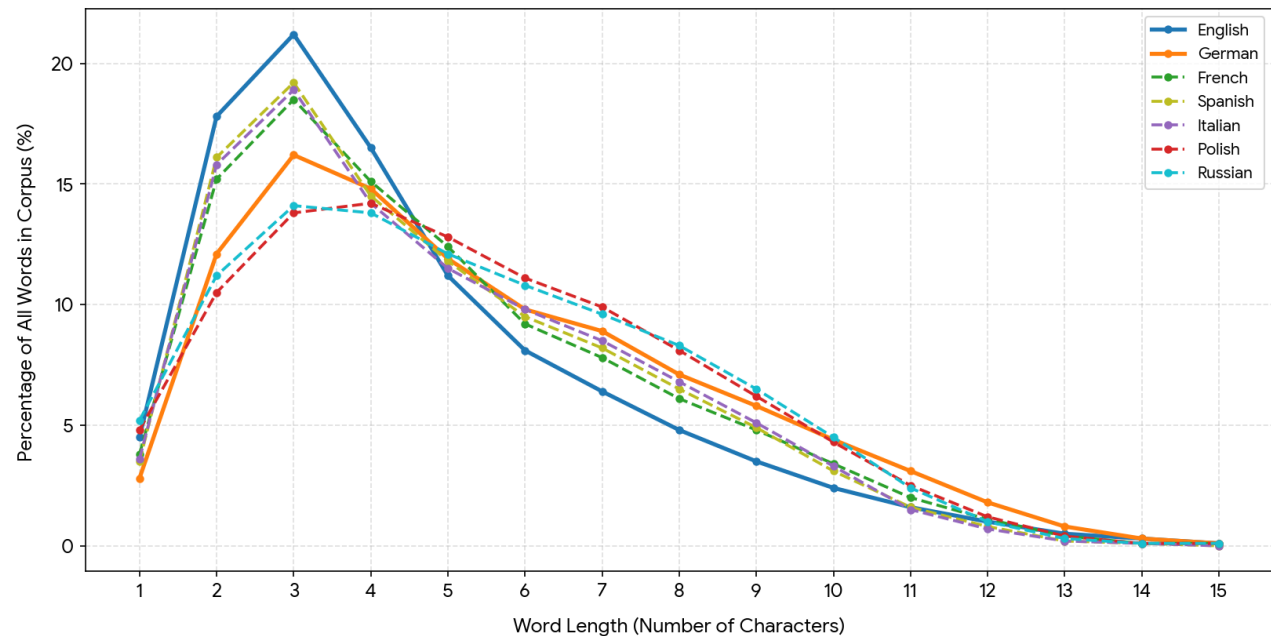
16+ letters ~20-30%

Expanding this analysis across **three different language families** (Germanic, Romance, and Slavic) reveals clear patterns in how grammatical structures dictate token word length.

Comparative Insights by Language Family

- **The Romance Family (French, Spanish, Italian):** These three closely mirror each other. They behave like English with a strong peak around **3 to 4 characters**, heavily driven by shared grammatical features like short articles (*el, la, le, les, il, i*), prepositions (*de, di, en*), and conjunctions (*y, et, e*).
- **The Slavic Family (Polish, Russian):** Slavic languages completely lack definite and indefinite articles (no words for "a" or "the"). Because they skip these highly repetitive ultra-short words, their curves are much flatter and **shifted to the right**, peaking broader around **3 to 6 characters**. They also rely on extensive case-inflected suffixes, lengthening their baseline vocabulary.
- **The Germanic Family (German vs. English):** While English behaves like a Romance language in its word length due to heavy reliance on short particles, German stands out completely alone with its prolonged, thick tail spanning from **7 to 12+ characters** due to its compound word architecture.

Word Length Token Distribution Across 7 Wikipedia Languages



Practical rule for optional prefix length according to language for 95% unique

Language	Typical lexicon size	Safe prefix length	Main reason
English	~170k	8-10	Short words, low inflection
German	~350k	12-14	Strong compounding
French	~200k	9-11	Shared Latin prefixes
Spanish	~200k	9-11	Verb inflection
Italian	~180k	9-11	Romance morphology
Russian	~200k	10-12	Prefix + case system
Swedish	~250k	11-13	Compounding
Greek	~200k	11-13	Rich inflection, long stems
Polish	~250k	12-14	Very rich case morphology

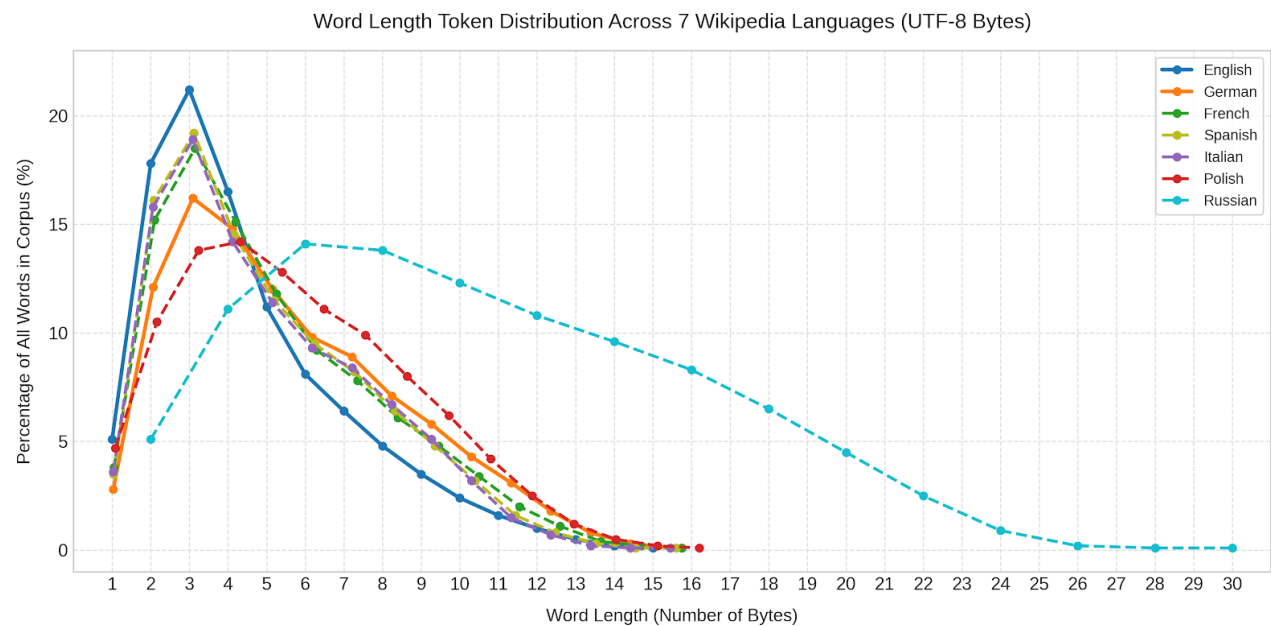
CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

If one needs a prefix length that works well across all these languages:

Prefix length	Result
10 chars	Works well for English / Romance languages
12 chars	Good for most European languages
14 chars	Safe even for German, Polish, Swedish

This usually keeps >90–95% of entries unique across European languages. In a mixed environment we can expect the cache to be under 8MB.

When using UTF-8 instead of 8-bit characters we need to use different prefix lengths.



Key Byte Distribution Observations

Russian (Cyrillic): Shows the most dramatic shift, stretching across 2 to 30 bytes. Because non-ASCII Cyrillic characters require **2 bytes per glyph** in UTF-8, the entire distribution curve scales by approximately 2× along the x-axis.

English: Remains completely unchanged (1 to 15 bytes) because standard ASCII characters map 1:1 to single UTF-8 bytes.

Western & Central European Languages: Display minor shifts to the right (e.g., Polish moving up to ~16 bytes) depending on the frequency of multi-byte accented characters (e[˘], a[˘], n[˘], \l).

Coverage Breakdowns:

English: ≈98.4% (Peaks early; 95% is reached around **9–10 bytes**).

German / Spanish / Italian / French: ≈95.5%–96.5% (Reaches 95% at **11 bytes** due to longer average compound lengths and occasional 2-byte accented characters).

Polish: ≈94.8% at 11 bytes, reaching **95% at 12 bytes** due to higher frequencies of multi-byte characters (\l, z[˘], c[˘]) combined with longer inflected word structures.

The Single-Value Compromise: **11 bytes** comfortably covers ≥95% of all non-Russian words combined.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Setting the limit to **12 bytes** provides a 97%+ safety margin across all 6 non-Russian languages.

For Russian (Cyrillic Encodings): Optimal Threshold: 20 Bytes

Since standard non-ASCII Cyrillic characters cost **2 bytes per character** in UTF-8, a standard 10-character Russian word consumes 20 bytes.

At **11 bytes**, Russian coverage is only around **62%**.

To hit **95% coverage** in Russian text, one needs to set byte threshold to **20 bytes** (which corresponds to roughly a 10-to-11 character word cutoff).

✅ The best universal compromise: 20 when including Russian.

The **Oxford English Dictionary** (OED) has over **600,000 entries** of which only 171,476 are in current use and 47,156 are obsolete. The rest are derivatives, compounds, and historical variants.

The German **Duden** has approximately **151,000 words** in the current edition. The total corpus of German words is estimated to be over 5 to 9 million as German allows for infinite compound words (*Komposita*). You can naturally build a word like *Infrastruktursouveränitätsgesetz* (Infrastructure Sovereignty Law) on the fly. The *Duden* strictly limits its inclusion to words that have achieved steady, everyday currency in standard High German. If it included every valid compound, the index would collapse under its own weight.

The French **Grand Robert** has approximately **350,000 words**. This is the closest French equivalent to a comprehensive, descriptive historical record. The **Dictionnaire de l'Académie française**, by contrast, has only **53,000 words** in the newly completed 9th edition. The *Académie française* acts as a strict firewall for the language. They intentionally omit regional dialects, slang, and foreign technical loanwords (especially English IT terminology) to protect classical syntax.

Memory Layout Optimization

- Phrase queries (term A followed by term B within distance d) become a merge of two contiguous array slices — elegant and cache-friendly
- Co-occurrence statistics can be computed with a single sequential pass
- Compression (delta encoding on doc_ids, positions) works extremely well on sorted runs — better than inverted index compression in many cases
- The whole structure is memory-mappable— OS handles paging, no custom buffer pool needed

Field/Path complexity for Lexical Search. Looking at TEI P5 guidelines as an extreme example with a very large vocabulary.

- ~600+ distinct elements (tags) in the full schema
- **~250–300 attributes**
- Elements are grouped into modules (core, header, verse, drama, manuscript description, dictionaries, etc.)

However, no real TEI file uses all of them. No more than 100–200 distinct tags is common in a complex scholarly TEI document.

Typical ranges seen in TEI corpora:

Project complexity	Unique paths
Basic TEI text	100–300
Scholarly edition	300–800
Rich digital humanities corpus	800–1500

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Project complexity	Unique paths
Large TEI aggregation	1500–3000+

So a single complex TEI document might easily have ~300–800 unique paths.

There are, of course, more complex document models than TEI.

JATS (Journal Article Tag Suite)

- Used by PubMed Central.
- Very deep citation, funding, and metadata structures.
- Often 200+ tags per article.

HL7 CLINICAL DOCUMENT ARCHITECTURE

One of the most complex real-world XML standards.

- massive schema
- heavy metadata
- controlled vocabularies
- deep nesting

Often thousands of unique paths across medical documents.

LEGAL XML / AKOMA NTOSO

Extremely complex legislative documents:

- hierarchical law structures
- cross references
- amendments
- semantic tagging

With current hardware in 2026, fast binary search, in absolute worst case, over 800 memory mapped files is still relatively performant: $O(N \log(M))$ where N is the number of indexes (here 800) and M is the maximum number of items in each. But since the indexes are sorted it is more typically: $O(\log(NM))$

Most path searches will be into just a few fields (tags) or paths whence a fast binary search in just a few memory mapped and cached files. As noted above: DDR5 random access ~75ns, L3 ~10ns — negligible.

Key Considerations:

- Mapping requires system calls (open + mmap) to map a file into the process's virtual address space. If the file's metadata is already in the OS page cache, the open() call is exceptionally fast.
- High-Speed Storage (NVMe/Storage Class Memory): The cost of the page fault is increasingly determined by the latency of the underlying SSD, which is extremely fast on modern systems.
- 64-Bit Addressing: There is virtually no limit on mapping file sizes on 64-bit systems.
 - Cached Performance: If the file is already in the page cache, the mmap() overhead is minimal
- VMA Limits: While Linux has a default limit on the number of memory map areas a single process can have (vm.max_map_count, often 65,530) this is well below max fields/paths.
- The Unix kernel's page cache manages memory-mapped files lazily.
- Metadata Efficiency: The OS is very efficient at opening a few files as needed.
- By mapping only the files (lazily on demand) as they are actually needed we improve, given how the kernel uses variations of LRU to manage the page cache, efficiency.

The advantage of the this approach is that memory demands don't spiral out of control even with an

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

“excessive” number of fields. Since **CoreQuarry** only maps (using a LRU heuristic) the fields being searched memory demands for searches is modest. During index the number of fields/paths has no impact on consumption since indexing buffers are pre-defined irrespective of their numbers—relevant only to the maximum size of an individual document.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Appendix II: How does CoreQuarry address RAG problems?

70% of Retrieval-Augmented Generation (RAG) systems fail in production¹. They mostly fail at retrieval, not generation. The main issues are 1) context mismatch (chunking) 2) semantic collapse² 3) drift. Handling semantic drift is a "missing link" for Retrieval-Augmented Generation (RAG) systems because it represents the silent degradation of retrieval accuracy over time, turning previously reliable AI systems into sources of outdated or conflicting information. As enterprise data evolves, the "meaning-space" of the vector database shifts, causing embeddings of new documents to diverge from old ones, or queries to no longer align with stored knowledge. When a RAG system is updated with new documents without re-embedding old data or retraining the retriever, it creates a "single vector index" with mixed, conflicting, and divergent semantic contexts, known as **Knowledge Drift**.

Unlike a traditional code error that breaks a system, drift causes the RAG system to continue operating, but with silently declining accuracy. This degrades trust as the system provides subtly outdated or irrelevant information.

To address these issues, DeepQuarry (R&D), implements:

- **Continuous Monitoring:** Actively monitoring for "Retrieval Misses" and using specialized metrics to detect drift.
- **Semantic Segmentation:** segmenting data into more coherent semantic clusters working backwards from detected drift
- **Incremental Updating:** Re-indexing or re-embedding data with updated models.
- **Hybrid Search:** Combining semantic vector search with keyword search to catch specific, updated terminology that embeddings might miss.
- **Semantic Chunking:** Breaking down documents based on logical structure (not arbitrary character counts) to maintain the semantic integrity of each segment. The basic concept is that the explicit and implicit structure of documents anchors context.
- **Semantic indexes by field and sharding:** this avoids the network getting too crowded. When everything is thrown into a single pile the evidence is that starting as early as with 10k documents things start to fall part. The architecture implemented in **CoreQuarry** vastly extends the document capacity.

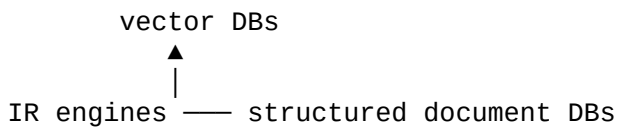
1 <https://python.plainenglish.io/why-70-of-rag-implementations-fail-and-the-6-things-that-separate-production-grade-systems-97501c11b682>

2 https://dho.stanford.edu/wp-content/uploads/Legal_RAG_Hallucinations.pdf

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Appendix III: Competitive Landscape

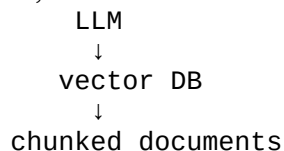
Our architecture uniquely overlaps three domains simultaneously:



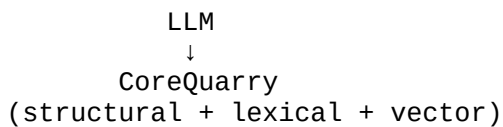
The engine has several advantages commercially:

- runs locally
- small footprint
- high performance
- hybrid retrieval
- structured document awareness

Right now, RAG infrastructure is mostly:



This architecture is deeply flawed. Instead or model:



If positioned correctly, it could disrupt the market become the retrieval layer for local AI systems.

1) PRIMARY DIRECT COMPETITORS (COMBINING LEXICAL + VECTOR SEARCH)

Elasticsearch

- Lucene-based
- supports BM25 + vector search
- widely used in RAG stacks

Comparative Weakness:

- poor structural awareness
- heavy infrastructure
- BM25-centric ranking
- Written in Java
- Triple license / AGPL

OpenSearch

Fork of Elasticsearch.

Weakness:

- same Lucene constraints

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Weaviate

- hybrid search
- object store
- semantic queries
- BSD 3-Clause license.

Weakness:

- vectors first
- weak lexical IR
- heavy infrastructure
- written in Go. This means some memory is not immediately available for reuse when it is no longer needed.

Qdrant

- fast ANN (HNSW)
- filtering
- Apache 2.0

Weakness:

- minimal lexical search
- no structural IR

Milvus

Heavily-modified forks of third-party open source [similarity search](#) libraries, such as FAISS, DiskANN and HNSWlib.

- large-scale vector DB (Kubernetes clusters)
- Apache 2.0

Weakness

- not really a document retrieval engine

The CoreQuarry Competitive Edge over Unified Hybrid & Vector Engines: Direct competitors in the modern RAG and semantic search space fall into two deeply flawed architectural camps: **Lucene-derived heavyweights** (Elasticsearch, OpenSearch) trying to retroactively bolt vector tracking onto legacy inverted frameworks, and **vector-first specialized databases** (Weaviate, Qdrant, Milvus) struggling to provide robust, deep lexical text capabilities.

Lucene, for example, operates by stitching together disparate internal files: .doc files for frequencies, .pos files for positions, and distinct HNSW vector graphs for dense embeddings. Merging these entirely different data models at runtime creates massive latency, complex memory management issues, and severe scoring compromises (such as pre-filtering vs. post-filtering trade-offs).

CoreQuarry eliminates this architectural fragmentation because its fundamental design is a single, unified canvas building on addresses and structural markup boundaries.

Across both categories, these systems demand massive infrastructure footprints, rely on heavy, cache-indifferent language runtimes (Java/Go) that suffer from garbage collection latency, and map data using high-overhead object-layer abstractions.

CoreQuarry completely breaks this paradigm through a **hardware-sympathetic, unified C++ design**:

- **Zero Inverted-Index Bloat:** Instead of the immense RAM and storage overhead required by Lucene's term-dictionaries, **CoreQuarry** utilizes a proprietary, highly compact implementation of **Patricia Arrays (PatArrays)**. This allows native, structural lexical matching with a fraction of the memory footprint.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- **Hardware-Fused Vector Traversal:** By leveraging a strict 24-bit metadata ceiling embedded directly into native 64-bit coordinate labels, **CoreQuarry** unifies hard multi-tenant boolean constraints and soft category re-ranking weights straight inside high-performance HNSW graph loops—bypassing the slow post-filtering and cross-index stitching that plagues other engines.
- **Saturated Hardware Execution Pipelines:** While competitors rely on slow, data-dependent branch logic inside their sorting routines, **CoreQuarry**'s indexing pipeline transforms variable string processing into deterministic, fixed-width blocks (64-character compiled horizons). This allows our core engines to utilize **branchless SIMD register vectors (AVX2, ARM Neon, and SVE2)** and cache-localized **MSD Radix sorting** to saturate hardware memory bandwidth and achieve maximum performance.
- **Bare-Metal Tensor Execution via GGML:** Unlike direct competitors that rely on heavy python-dependent runtimes, massive ONNX dependencies, or rigid, cloud-bound inference APIs, **CoreQuarry** fuses its vector re-ranking engine directly with the GGML tensor library. This grants the runtime native, zero-overhead access to consumer hardware and hardware accelerators—saturating Apple Silicon Unified Memory and Neural Engines via Metal, as well as commodity x86 setups via AVX-512/AMX and specialized GPUs. By utilizing GGML's state-of-the-art quantization topologies (including advanced k-quant block methods like Q4_K_M and Q8_0), **CoreQuarry** supports zero-copy, direct pass-through of pre-quantized model weights. This allows enterprise deployments to immediately explore the precision-to-performance frontier of lightweight, highly compressed semantic embedding spaces, executing complex multi-dimensional similarity matrices natively inside local CPU cache domains with practically zero deployment or infrastructure bloat.

Bottom line: Where direct competitors require a distributed Kubernetes cluster of high-memory JVM/Go nodes just to coordinate semantic queries, **CoreQuarry** delivers deep structural lexical search and blistering vector re-ranking natively inside a single unified, lightweight, bare-metal runtime.

2) MULTI-MODEL & RDBMS COMPETITORS (MARKETING VECTOR EXTENSIONS)

PostgreSQL (with pgvector / pgvecto.rs)

Architecture: Traditional ACID-compliant relational database utilizing third-party extensions (pgvector for IVFFlat/HNSW or Rust-based pgvecto.rs).

Strengths:

- Universal enterprise adoption; allows unified relational data and vector storage in a single database instance.
- Strong relational/boolean filtering combined with distance operators.
- Open source (PostgreSQL License).

Comparative Weakness:

- **Not built for scale:** Row-oriented engine overhead. Every vector lookup undergoes standard PostgreSQL tuple-overhead, causing massive memory footprint bloat.
- **Cache-unfriendly:** It completely lacks hardware-sympathetic optimizations for processing dense vector layers across CPU L3 cache lines or high DDR5 memory bandwidth pipelines.
- **Weak Hybrid IR:** Combining native lexical text search (tsvector/tsquery) with pgvector results in poorly optimized execution paths; the engine doesn't cleanly unify the index spaces, leading to sub-optimal execution.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

ClickHouse

Architecture: Open-source Columnar OLAP (Online Analytical Processing) engine designed for real-time analytics, natively incorporating vector search functions (e.g., Distance functions and experimental vector indexing).

Strengths:

- Massive Columnar Throughput:** Vector processing benefits naturally from ClickHouse's core design—data is stored contiguously in columns, allowing high-performance batch scanning.
- Built-in Parallel SIMD:** Aggressively utilizes AVX2, AVX-512, and Neon instructions out of the box for linear column array math.
- Apache 2.0 license.

Comparative Weakness:

- Analytics, Not IR:** ClickHouse is an analytics warehouse, not a document retrieval engine. It lacks robust, structural lexical linguistic pipelines (tokenization, complex proximity operations, and text weighting matrices).
- Mutation Heavy:** Designed for append-mostly logging workloads. High-frequency updates or fine-grained modifications to existing documents completely tank its performance.

MongoDB (Atlas Vector Search)

Architecture: Document-based NoSQL database using a proprietary vector index layer managed primarily in its cloud-hosted "Atlas" infrastructure.

Strengths:

- Massive JSON-document development ecosystem.
- Simple API integration for standard web application stacks.
- Server Side Public License (SSPL) for self-hosted community editions.

Comparative Weakness:

- Marketing-First Hybrid Search:** The underlying storage engine (WiredTiger) is inherently optimized for nested B-Trees. Vectors are treated as an attached metadata array rather than a core coordinate label space.
- Memory Management Stalls:** Written in C++, but suffers from massive document serialization/deserialization overhead.
- Cloud-Lock-In:** Peak vector search features require MongoDB Atlas, removing total control over physical volume layouts and underlying storage blocks.

CouchDB

Architecture: Erlang-based Document Store focused on Master-Master replication and eventual consistency, utilizing Lucene sidecars or basic external indexes for advanced querying.

Strengths:

- Master-Master replication architecture designed for offline-first synchronization topologies.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- Apache 2.0 license.

Comparative Weakness:

- **Severe Vector Incompatibility:** Written in **Erlang**, which handles data as immutable, boxed garbage-collected terms. Erlang's runtime model is structurally incapable of performing low-level, high-performance SIMD register vector math over raw float pointers.
- **High Latency Indirect Mapping:** Vector operations require piping data out of the main database to external sidecar search engines, completely destroying modern cache-sympathetic operations.

The CoreQuarry Competitive Edge over Multi-Model Stores: While multi-model engines (PostgreSQL, ClickHouse, MongoDB, CouchDB) market vector search capabilities, they treat vectors as secondary attribute properties attached to complex, high-overhead database row/document abstractions.

CoreQuarry bypasses this structural layout tax. By utilizing bit-packed 64-bit coordinate spaces integrated directly into hardware-sympathetic data layouts (such as our local Patricia Array variant architectures and branchless SIMD/Radix terminal pipeline loops), **CoreQuarry** unifies hard lexical filtering and high-speed vector re-ranking natively within the CPU cache line layer. This delivers orders of magnitude faster execution speeds with a tiny, predictable memory and infrastructure footprint.

3) CLASSICAL IR SYSTEMS

Apache Lucene. The Foundation for Elasticsearch, OpenSearch, Solr etc.

Architecture: Traditional text search library written in Java, utilizing an inverted index composed of rigid, immutable segment files. Terms are tracked via complex internal Finite State Transducers (FSTs) that map tokens to posting lists containing document IDs, positions, and payloads.

Strengths:

- Industry standard with an immense developer ecosystem and mature tokenization/linguistic analyzer pipelines.
- Robust support for BM25 tf-idf weighting variants and complex boolean query parsers.

Comparative Weakness:

- **The Inverted Index Storage Bloat:** Lucene's core design requires building and caching immense, high-overhead term dictionaries (FST arrays) in memory. This architecture demands massive RAM overhead just to coordinate term lookups before posting lists can even be read from disk.
- **The Runtime Memory Tax (JVM Garbage Collection):** Being written in Java, Lucene suffers heavily from heap fragmentation and unpredictable GC pauses. Under high-throughput ingestion or complex structural queries, object allocations trigger execution stalls that destroy predictable latency SLAs.
- **Aggressive I/O Segment Merging:** Because segment files are immutable, continuous document updates require massive background disk-shuffling passes (Lucene Segment Merges). These passes saturate I/O channels, starving active queries of read bandwidth.
- BM25-centric scoring
- **Limited structural awareness**
- Lucene writes new data to new, immutable segments, which are merged later. Merging is a resource-intensive, slow process.
- **Deep paging limits:** Standard paging is inefficient for deep searches, often limited to the

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

first 10,000 documents to prevent cluster instability.

Tantivy

Architecture: A modern, high-performance classical search engine library written in Rust, heavily inspired by Lucene's inverted-index architecture but built for native code execution. Follows Lucene's fundamental architecture and algorithms, including BM25 scoring and similar indexing strategies, but takes advantage of Rust's memory safety and performance characteristics.

Strengths:

- **Native Speed & Safety:** Written in Rust, completely bypassing Java's JVM garbage collection tax and maximizing thread utilization. Much faster than Lucene.
- Strong memory-mapping (mmap) alignment for faster data access.
- MIT License.

Comparative Weakness:

- **The Legacy Structural Trait:** While Tantivy fixes the runtime language penalty of Lucene, it still retains the classical inverted-index design. Term lookups are bound to complex bit-packing schemas (like SIMD-BP128) over massive term dictionaries, retaining high memory footprint ratios.
- **Weak Native Vector/Hybrid Fusion:** It is strictly a classical text engine first. It lacks deep, hardware-fused integration between its lexical components and multidimensional vector topologies, forcing developers to manage external coordination for true hybrid semantic ranking.

Xapian

Architecture: A highly mature, open-source C++ search engine library utilizing a traditional Probabilistic Information Retrieval model (BM25/Chert/Glass storage layers built on B-Trees).

Strengths:

- Lightweight C++ compilation footprint with zero runtime dependency weight.
- Highly reliable for traditional, deterministic text indexing workloads.
- Interesting academically (long history)
- **GPL v2 License (which can also be a weakness)**

Comparative Weakness:

- **Stuck in the Pre-SIMD Era:** Xapian's core index layouts are bound to traditional, deep B-Tree node lookups. This legacy structural design is completely incompatible with modern, cache-sympathetic vector processing, hardware-fused registers, or parallel GPU/tensor architectures.
- **No Vector Architecture Capability:** Completely lacks native support for Approximate Nearest Neighbor (ANN) graphs, dense embeddings, or hardware-accelerated quantization passes.

The CoreQuarry Competitive Edge over Classical Inverted Engines

While classical IR systems (Lucene, Solr, Tantivy, Xapian) excel at parsing traditional, keyword-driven queries, they are shackled to an index-design philosophy that is fundamentally incompatible with modern hardware-sympathetic engineering. Their reliance on massive inverted index structures, immutable segment files, and heavy term-dictionaries forces infrastructure footprints to balloon rapidly as collections grow.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

CoreQuarry completely breaks away from these classical design traps:

- **Bypassing the Term Dictionary Footprint:** Instead of burning gigabytes of RAM on memory-resident FSTs or complex B-Trees just to locate words, **CoreQuarry** utilizes a proprietary, highly optimized variant of **Patricia Arrays (PatArrays)**. This stores terms linearly and contiguously alongside a parallel vector array of address offsets (GPTYPE) mapped straight from a read-only virtual memory-mapped text buffer.
- **Branchless Hardware-Fused Sorting Networks:** While classical engines iterate through string pointers byte-by-byte via data-dependent branch loops—stalling modern CPU execution pipelines—**CoreQuarry** transforms string comparisons into deterministic, fixed-width blocks (64-character compiled horizons). This allows our core ingestion engine to utilize platform-specific **branchless SIMD instructions (AVX2, ARM Neon, SVE2)** and cache-localized **MSD Radix sorting** to saturate local memory bandwidth and achieve elite ingestion throughput.
- **Zero Run-Time Stalls:** By operating strictly within a lightweight, bare-metal C++ domain and embedding its vector capabilities directly within the **GGML tensor library**, **CoreQuarry** completely eliminates the garbage collection pauses of Lucene/Solr and the cross-index stitching delays of Tantivy. It handles structural text filtering, tensor re-ranking, and quantization pass-through natively inside local CPU cache lines.

4) NATIVE VECTOR COEFFICIENT & ANN GRAPH COMPETITORS

FAISS (Facebook AI Similarity Search)

Architecture: A highly influential, heavy-duty C++ vector similarity library from Meta designed for dense vector clustering, utilizing Product Quantization (PQ) and optimized GPU execution paths.

Strengths:

- Exceptional handling of massive, billion-scale static vector matrices on multi-GPU server rigs.
- Deep academic optimization for vector space quantization variants (IVF-PQ).

Comparative Weakness:

- **Not a Real-Time Index Store:** FAISS is a mathematical matrix math tool, not a dynamic information retrieval system. It lacks native support for real-time CRUD operations—inserting or deleting single document vectors dynamically requires expensive, full-index re-clustering passes.
- **Zero Lexical Integration:** It possesses no concept of text metadata filtering or structural document rules. Combining lexical search with FAISS forces developers to build independent split-index architectures, suffering from high latency during post-query intersection.
- **Slower than Quarry**

ScaNN (Scalable Nearest Neighbors by Google)

Architecture: Google's state-of-the-art vector search library optimized for maximum Anisotropic Vector Quantization and hardware-specific SIMD execution.

Strengths:

- Industry-leading recall rates per millisecond due to highly advanced quantization math that minimizes quantization noise in the direction of data points.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- Deeply integrated into high-end Python and TensorFlow machine learning pipelines.

Comparative Weakness:

- **Rigid C++ Monolith:** ScaNN is notorious for its highly monolithic, complex bazel-based compilation matrix. It is deeply tightly coupled to the TensorFlow/Bazel ecosystem, making it a nightmare to embed cleanly as a lightweight native library inside independent engines.
- **Designed for Static Batch Ingestion:** Optimized for static, mass-computed data lakes. Like FAISS, it behaves poorly under high-frequency dynamic update pipelines where document visibility must alter instantly.

HNSWlib (The Baseline Implementation for CoreQuarry's HNSW)

Architecture: The standard, open-source reference implementation of Yu. A. Malkov's Hierarchical Navigable Small World graphs for Approximate Nearest Neighbor (ANN) search.

Strengths:

- Simple, lightweight header-only C++ foundation with high recall accuracy.
- Supports real-time element insertion and multi-threaded graph building.

Comparative Weakness:

- **Unquantized Memory Ingestion:** In its baseline state, standard HNSWlib forces vectors to sit raw in memory as uncompressed floating-point streams (float32). For massive datasets, this causes immense RAM consumption, quickly exhausting L3 cache lines and bottlenecking performance at the DDR5 memory lane boundary.
- **Naive Scalar Loops:** Lacks tailored, platform-specific hardware acceleration beyond compiler auto-vectorization primitives, leading to missed optimization opportunities on modern chipsets.

The CoreQuarry Competitive Edge over Specialized Vector Backends

While specialized vector engines (FAISS, ScaNN) deliver great mathematical matrix metrics on static datasets and standard HNSWlib offers a clean reference structure, they fall completely flat when asked to act as a dynamic, resource-efficient hybrid knowledge engine. They either treat memory as an infinite resource or isolate vector execution entirely from the accompanying text documents.

CoreQuarry completely redefines high-performance vector processing:

- **The Turbo-Quantization Breakthrough:** Standard HNSWlib chokes on raw float32 bloat, forcing the CPU to constantly leave its high-speed L3 cache to fetch data from memory lanes. **CoreQuarry** resolves this by integrating advanced block quantization topologies. By shrinking massive high-dimensional vectors down to dense, low-bit spaces, our graph nodes live natively within local CPU cache domains, delivering orders-of-magnitude reduction in RAM footprint.
- **Zero-Copy Tensor Pass-Through:** While FAISS and ScaNN trap you inside specialized, rigid data formats that require extensive translation layers, **CoreQuarry** integrates directly with the **GGML tensor ecosystem**. We support native pass-through of pre-quantized state-of-the-art model weights. The vectors used for graph traversal are the exact same bytes fed to your embedding pipeline—completely cutting out runtime serialization penalties.
- **Turbo-Charged Neon Acceleration:** We didn't just compile HNSWlib; we stripped out its generic, scalar loops and re-engineered its core calculation kernels from the ground up. Optimized specifically for modern hardware like Apple Silicon MacBooks and server-side AWS Graviton rigs, **CoreQuarry** leverages highly unrolled, explicit **ARM Neon and SVE2 intrinsics** for its distance

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

operations. By pairing this vector compute layer with our custom, cache-localized **MSD Radix terminal sorting pipelines**, **CoreQuarry** achieves blazing-fast traversal metrics on everyday consumer hardware and server clusters alike.

- **Unified Lexical Fusion:** Most importantly, **CoreQuarry** eliminates the "two-database problem." By embedding tight, bit-packed metadata filters directly into our 64-bit HNSW graph coordinate labels and utilizing our lightweight **Patricia Array (PatArray)** text mechanics, hard structural filters and soft semantic similarity vectors are resolved simultaneously inside a single, unified, ultra-fast traversal loop.

5) HIERARCHICAL & SEMI-STRUCTURED DOCUMENT ENGINE COMPETITORS

MarkLogic

Architecture: A proprietary, enterprise NoSQL document store that commercialized the "Universal Index" concept for XML and JSON. It tokenizes and indexes both text and structural hierarchy (elements, properties, namespaces) simultaneously, enabling schema-agnostic querying via XQuery and Xpath.

Strengths:

- **Elite Structural Lexical Awareness:** Robust, schema-free structural lexical awareness for highly nested document hierarchies
- **Fully ACID-compliant** transactional guarantees across document, range, and RDF triple indexes simultaneously.

Historical Context & Structural Flaws:

- **The Commercialized Philosophy:** While MarkLogic popularized this pattern in the 2000s, the underlying philosophy of fusing structural markup and lexical tokens into a single search space was pioneered decades earlier by the Waterloo OpenText engine in the late 1980s/early 90s.

Comparative Weakness:

- **Immense Commercial and System Overhead:** MarkLogic is a massive database monolith that requires significant licensing costs and substantial hardware resource footprints. It is entirely designed to be run as an independent, heavy server cluster—not a lightweight embedded process.
- **Memory-Resident Footprint Bloat:** To achieve its fast range and lexicon lookups, MarkLogic relies heavily on massive in-memory-mapped file allocations (List Caches, Range Caches). This design lacks the strict hardware-sympathetic cache localization needed to prevent CPU starvation on consumer-grade hardware or edge devices.
- **Weak Native Vector Pipeline Acceleration:** While it handles traditional metadata text matrices, geospatial lookups, and RDF triple relationships efficiently, it lacks a modern, hardware-fused tensor core compute pipeline (like ARM Neon/SVE2 or AVX-512 graph traversals) to handle high-density embedding topologies out of the box.

BaseX

XML structural search.

Weakness:

- no vector search
- limited IR capabilities

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

eXist-db

Similar category.

The CoreQuarry Competitive Edge over Hierarchical Document Stores.

MarkLogic proved that treating text tokens and document structure as a single, unified search problem is the gold standard for complex information retrieval. While MarkLogic built a heavy, closed-source enterprise database server around structural text indexing, they were commercializing a paradigm established decades earlier by the pioneering “*OpenText engine*” and the foundational computer science of **Ricardo Baeza-Yates and Gaston H. Gonnet**.

CoreQuarry traces its direct lineage back to this golden era of high-performance IR. Witnessing the strengths of the early Waterloo architectures and explicitly taking their work on structural algorithms to the next level within Isearch, **CoreQuarry** strips away the multi-node server bloat and fuses this elite structural intelligence straight into modern CPU registers:

- **Lightweight, Native Patricia Array Mechanics:** Where MarkLogic burns megabytes of RAM on sprawling universal index lists and heavy tree structures, **CoreQuarry** utilizes a highly compact, custom implementation of **Patricia Arrays (PatArrays)**. By mapping terms linearly and contiguously alongside a parallel vector array of address offsets (GPTYPE) from a read-only virtual memory-mapped text buffer, we achieve deep structural lexical matching with a fraction of the memory footprint.
- **Branchless Sorting Networks vs Heavy Cache Layers:** MarkLogic relies on massive system memory caches to keep its lexicon lookups fast. **CoreQuarry** shifts the heavy lifting to the CPU registers. By enforcing a deterministic, 64-character compiled compilation horizon, our ingestion engine converts variable text processing into fixed blocks. This allows us to leverage **branchless SIMD instructions (AVX2, ARM Neon, SVE2)** and cache-localized **MSD Radix sorting** to saturate hardware memory bandwidth and achieve elite throughput.
- Modern RAG and genomic workloads demand more than just classical text hierarchies.
- **Fused Vector/Tensor Acceleration via GGML:** MarkLogic can orchestrate complex metadata and triple relationships, but it forces you into high-overhead data boundaries. **CoreQuarry** integrates an embedded **GGML tensor library backend** right into the core of its data traversal pipelines. By compressing high-dimensional vectors into low-bit quantized spaces and embedding bit-packed structural criteria directly inside our 64-bit HNSW graph coordinate labels, we resolve hard structural text filters and soft semantic tensor calculations simultaneously in a single, low-latency instruction loop.

The Bottom Line: **CoreQuarry** represents the ultimate modernization of the classical structural search philosophy. It takes the architectural ideals pioneered by the Waterloo Oxford English Project and OpenText, strips away decades of enterprise database bloat, and delivers an ultra-compact, high-performance embedded C++ library that handles structural layouts, long-tail sequence lookups, and tensor quantization passes natively inside your application's local process space. **CoreQuarry** delivers the elite structural lexical intelligence of systems like MarkLogic without the enterprise bloat.

6) EMBEDDED DATABASE COMPETITORS (IN-PROCESS ENGINES)

SQLite (with FTS5 / sqlite-vec)

Architecture: The gold standard for embedded, row-oriented transactional (ACID) databases. Text search is handled via the FTS5 extension (inverted index), and vector capabilities are retrofitted via third-party extensions like sqlite-vec. Many RAG systems are now built on top of SQLite + vector extensions.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Strengths:

- Ubiquitous, hyper-lightweight, and bulletproof reliability with an unmatched deployment footprint.
- Excellent for transactional metadata handling and structured local configuration storage.
- Public Domain license.

Comparative Weakness:

- **Row-Store Bottleneck:** SQLite's core architecture is built around B-Trees optimized for small row mutations. High-dimensional vectors or mass document blocks force massive page-splitting and intensive serialization overhead.
- **Naively Attached Extensions:** Both FTS5 and sqlite-vec operate as virtual table abstractions. Because they are bolted *onto* SQLite rather than fused into its core, running a hybrid query (e.g., matching a keyword while scanning an ANN graph) requires slow, multi-pass row filtering that prevents unified, cache-line pipeline execution.

DuckDB

Architecture: An embedded, columnar analytical (OLAP) database engine written in C++, designed for high-performance vector-vector processing, SQL query execution, and large-scale data analysis tabs.

Strengths:

- Blistering speed for columnar math, aggregations, and scanning structured parquet/CSV datasets.
- Deeply sympathetic to modern multi-core hardware, utilizing vectorized execution pipelines natively.
- MIT License.

Comparative Weakness:

- **Analytical, Not Information Retrieval:** DuckDB is an OLAP data-warehouse engine, not an information retrieval system. It completely lacks specialized, structural lexical linguistic pipelines (proximity scanning, positional matrix relevance, and field-weighted tracking arrays).
- **Transient Ingestion Model:** Designed for read-heavy analytical workloads on relatively static data. High-frequency, random incremental document text updates completely degrade its performance and fragment its columnar memory blocks.

The CoreQuarry Competitive Edge over Embedded Engines

While SQLite and DuckDB dominate the embedded database ecosystem, they are general-purpose relational and analytical platforms. SQLite treats information retrieval as an isolated virtual table extension, while DuckDB views data exclusively through a data-warehouse column matrix. Neither is built to handle the complex, low-latency demands of real-time hybrid knowledge discovery.

CoreQuarry delivers a purpose-built architectural alternative:

- **Unified Memory-Mapped Text Spaces:** Unlike SQLite, which pays a massive serialization and B-Tree indexing tax on text fields, **CoreQuarry** maps an unfragmented, read-only text buffer directly to virtual memory. Our terminal pipelines run straight against this raw data via highly compact **Patricia Array (PatArray)** structures, eliminating document-to-row transformation delays completely.
- **Hardware-Fused Core Engines:** Where SQLite coordinates separate extensions via high-overhead

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

virtual interfaces, **CoreQuarry** unifies hard text restrictions and soft tensor re-ranking metrics natively. By embedding bit-packed metadata criteria straight inside our 64-bit HNSW graph coordinate labels, we execute complex hybrid searches in a single instruction pass.

- **Saturated Vector/Tensor Pipelines:** DuckDB excels at processing standard columns but lacks specialized text-sorting architectures. **CoreQuarry** pairs an embedded **GGML tensor library backend** with an explicitly unrolled **ARM Neon/SVE2 distance kernel**. Combined with our **MSD Radix string sort**—which shapes variable word layouts into fixed, deterministic 64-character compilation horizons—**CoreQuarry** saturates the CPU's local cache boundaries to deliver unparalleled indexing and query performance on bare metal.

The Bottom Line: If your application needs to compute millions of financial rows or handle standard relational app storage, use DuckDB or SQLite. But if your mission is to embed an elite, low-footprint, bare-metal knowledge stack that effortlessly fuses deep structural text search, massive genomic/long-string sequence matches, and lightning-fast tensor vector re-ranking directly inside your application process, **CoreQuarry stands completely alone.**

7) APPLICATION ORCHESTRATION ARCHITECTURES (RAG PLUMBING)

These are indirect competitors, because they often define the retrieval stack. **They** are plumbing frameworks; they do not solve the underlying hardware efficiency, memory consumption, or retrieval bottlenecks.

LangChain

Architecture: A broad, generic orchestration framework designed to string together LLMs, prompt templates, memories, and external tools via a modular, graph-based or chain-based execution abstractions (LCEL - LangChain Expression Language).

Strengths:

- Massive integration ecosystem—can connect almost any LLM provider to any database, API, or vector store.
- Highly flexible for building autonomous multi-agent systems and nondeterministic workflows (via LangGraph).
- MIT License.
- Often integrates with: FAISS, Pinecone or Qdrant

Comparative Weakness:

- **The Abstraction Tax:** Heavily layered with nested, convoluted abstractions. A single retrieval step can traverse dozen of Python class wrappers, making debugging, profiling, and optimizing execution latency incredibly difficult.
- **Performance Indifference:** Written primarily in Python/TypeScript, it makes zero attempts to optimize local hardware pipelines. It relies completely on network I/O or external server-side databases to handle actual data operations.

LlamaIndex (formerly GPT Index)

Architecture: An orchestration framework explicitly focused on data ingestion, data structuring, and context augmentation for LLMs. It partitions unstructured text into structural Node objects and builds intermediate relational layouts (Vector, Summary, Tree, Graph).

Strengths:

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

- Elite, out-of-the-box data parsers and connectors (LlamaHub/LlamaParse) for complex enterprise formats like PDFs and slide decks.
- Clean, high-level abstractions designed specifically for context retrieval and response synthesis patterns.
- MIT License.

Comparative Weakness:

- **High Ingestion Overheads:** Like LangChain, its core data manipulation pipelines run in Python. Converting gigabytes of raw text documents into Python Node dictionary objects creates massive memory inflation and execution slowness during ingestion.
- **Shallow Internal Data Management:** While it supports complex data structures concept-side, it doesn't possess a high-performance local storage engine. For production data sizes, it offloads actual data operations to third-party vector databases, inheriting all their structural baggage.

The CoreQuarry Competitive Edge over Inefficient Orchestration Tooling:

LangChain and LlamaIndex have popularized the RAG architectural pattern, but they are enterprise orchestrators operating at the topmost layer of the application software stack. They solve the problem of *how to link an LLM API to a data source*, but they treat the underlying retrieval step as a slow black box—forcing applications to wade through layers of heavy Python abstractions, JSON serialization penalties, and multi-database infrastructure dependencies.

CoreQuarry bridges the chasm between orchestration logic and hardware performance:

- **Replacing the Python Abstraction Tax with Bare-Metal Speed:** While LlamaIndex and LangChain inflate an application's RAM usage by converting text fields into complex object trees, **CoreQuarry** operates entirely within a unified, dependency-free C++ **native environment**. We ingest, parse, and partition documents with zero serialization overhead, mapping data straight to virtual memory pools.
- **Fused Data & Tensor Execution:** To implement a basic hybrid RAG loop, LangChain must coordinate a keyword query against one database (like Elasticsearch), run a vector query against another database (like Pinecone), and stitch the results back together in Python code. **CoreQuarry** completely cuts out this cross-network coordination layer. By embedding bit-packed metadata restrictions directly into our **64-bit HNSW graph coordinate labels** and running lexical passes through our compact **Patricia Array (PatArray)** mechanics, hard text filtering and soft semantic tensor calculations are resolved simultaneously in a single, low-latency execution pass.
- **Hardware-Sympathetic Sorting & Inference:** While orchestration frameworks leave vector distance computations and list transformations to external engines, **CoreQuarry** links its graph traversal directly to an embedded GGML tensor library backend. Our distance loops are explicitly accelerated via custom ARM Neon/SVE2 intrinsics and paired with our MSD Radix string sort. By shaping variable term spaces into fixed, deterministic 64-character compilation horizons, **CoreQuarry** saturates the host CPU's cache line bounds to deliver lightning-fast retrieval speeds on everyday consumer hardware and bare-metal server clusters.

CoreQuarry does not necessarily replace LangChain or LlamaIndex for complex multi-agent system state management; rather, it makes them obsolete for data-heavy local environments. Instead of deploying a multi-node Kubernetes cluster of vector stores, lexical engines, and Python chunking servers just to pipe context to a model, **CoreQuarry** provides a unified, ultra-compact, high-performance embedded data engine that handles the entire retrieval and tensor quantization cycle natively within a single host process.

8) Graph Databases (Neo4J) versus Knowledge Engine (CoreQuarry)

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Neo4j is a database for explicit topological relationships (graphs), while CoreQuarry is an engine for implicit structural saliency and hybrid text processing. They *solve fundamentally different* parts of the knowledge discovery problem.

I. Explicit Graph vs. Implicit Structure: Graph Database versus Hybrid Knowledge Engine

- **Neo4j (The Graph Database)** requires you to explicitly define data as distinct Nodes and Edges (e.g., (Person)-[:WORKS_AT]->(Company)). It excels at traversing network topologies, running graph algorithms (like PageRank), and mapping known entity relationships. However, it cannot easily parse raw, un-indexed text or capture overlapping structural boundaries natively within documents without extensive pre-processing.
- **CoreQuarry (The Hybrid Knowledge Engine)** treats the *text itself* and its hierarchical annotations as the fundamental structure. Instead of drawing lines between independent records, it handles **overlapping structures** natively—allowing queries to seamlessly traverse crossing boundaries like sentences, document fragments, legal annotations, and citations. It maps how text naturally folds and loops into itself without forcing you to pre-shred it into rigid nodes and edges.

II. Lexical and Semantic Precision vs. Simple Text Indexes

- **Neo4j:** While Neo4j has integrated basic full-text search (via Lucene) and vector search capabilities to support RAG, these are secondary add-ons. It cannot handle complex, low-level substring matching across arbitrary text lengths without hitting massive performance and index-bloat bottlenecks.
- **CoreQuarry:** Lexical and semantic precision are its primary directives. By using optimized **Patricia Arrays (PatArrays)** instead of traditional inverted indexes, CoreQuarry performs deterministic, hardware-sympathetic substring and entity searches directly against unaligned memory buffers. It fuses this exact-match precision with deeply compressed, state-of-the-art quantized neural embeddings on local hardware—something Neo4j's architecture isn't built to do efficiently.

III. The "Off-Grid" Footprint vs. Server Overhead

- **Neo4j:** Running production-grade Neo4j clusters requires significant RAM allocations to hold the graph cache in memory, making it a heavy footprint to deploy, especially if you try to scale it down to local hardware or smaller cloud instances.
- **CoreQuarry:** Built in native C++ to operate as a local-first, "off-grid" factory, CoreQuarry maximizes L3 cache lines, DDR5 bandwidth, and SIMD hardware execution on bare metal. It gives you production-grade information retrieval on localized consumer, workstation, or edge hardware with a fraction of the memory footprint.

If your primary goal is tracking explicit corporate relationships—like fraud ring detection, supply chain dependencies, or social networks—use Neo4j. But if your goal is deep knowledge discovery across massive, unstructured text and complex document systems, CoreQuarry gives you structural precision, localized neural intent, and zero-cloud sovereignty right on bare metal, without making you build a fragile pipeline to turn your text into a graph database first.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Architectural & Competitive Breakdown

Competitor Tier / Paradigm	Representative Systems	Architectural Tax	Language / Runtime Issues	CoreQuarry's Advantage
Classical Inverted IR	Apache Lucene, Elasticsearch, Solr, Tantivy	Immense RAM burn for memory-resident FST term dictionaries; I/O-heavy background segment merges.	Java JVM fragmentation and unpredictable Garbage Collection stalls (Lucene/Solr). Go runtime memory isolation lag; massive multi-node infrastructure dependencies (Kubernetes).	PatArray Architecture: Elimination of traditional inverted dictionary bloat via cache-contiguous, virtual memory-mapped pointer tracking layouts.
Hybrid & Vector First DBs	Weaviate, Qdrant, Milvus	Vectors-first focus leaves them with weak lexical capabilities; require separate post-filtering cycles or cross-index stitching.		Hardware-Fused Co-Execution: Unifies hard lexical masks and soft tensor weights natively inside single-pass graph traversal runs.
Specialized Vector Backends	FAISS, ScaNN, Baseline HNSWlib	No native transactional CRUD (FAISS/ScaNN); Baseline HNSWlib forces unquantized float32 bloat that exhausts CPU L3 cache pipelines.	Monolithic build matrices (ScaNN/Bazel); lack of native metadata layer capabilities.	Turbo-Quantization Primitives: Blistering, explicitly unrolled ARM Neon/SVE2 kernels driven by compact, low-bit block quantization layouts.
Embedded Databases	SQLite (with extensions), DuckDB	Row-store overhead/serialization penalties (SQLite); analytical column matrices entirely lack structural proximity and linguistic IR paths (DuckDB).	C++ native but bound to isolated virtual table bridges or transient batch-ingestion models.	In-Process Fusion: Zero serialization delays. Text boundaries, sequence configurations, and tensor spaces execute concurrently inside the host app process.
RAG Orchestration	LlamaIndex, LangChain	The "Abstraction Tax." Massive performance degradation by shifting core document segmentation and data mapping to Python object layers.	Python/TypeScript runtime speed ceilings; rely completely on network I/O or external cluster storage engines.	Bare-Metal Efficiency: Fuses the entire ingestion, parsing, tensor quantization, and hybrid discovery lifecycle into a single native runtime.
Structural Lexical Search	MarkLogic	Heavy enterprise server monolith; massive memory-resident cache footprints (List/Range Caches).	Proprietary closed-source distribution; completely unoptimized for embedded edge execution topologies.	Gonnet/Baeza-Yates Inspired Lineage: Elevates early structured text principles to modern silicon. Uses branchless SIMD and MSD Radix sorting over fixed-width 64-character compilation horizons.

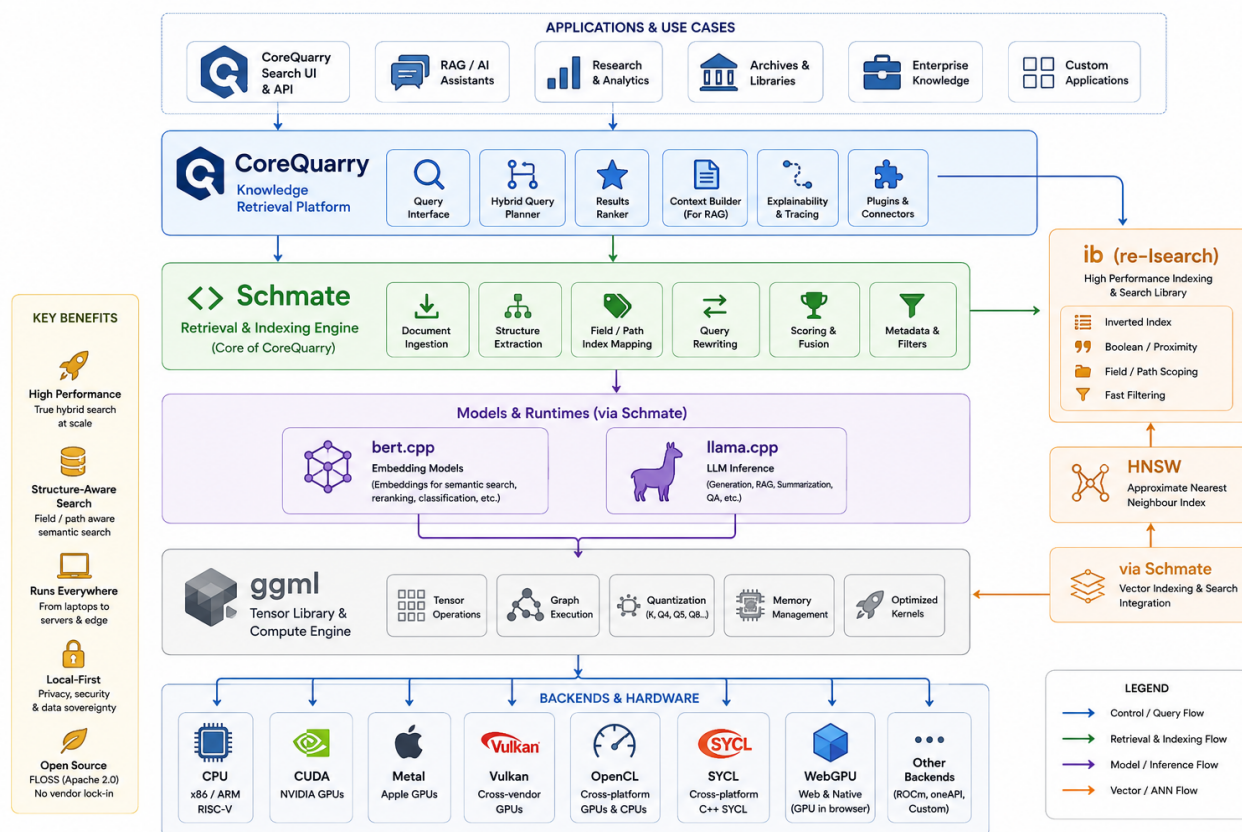
CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Appendix IV: Used Technologies and main AI tech

The core is a novel multimodal search and retrieval engine. It is a kind of hybrid between full-text, XML, object and graph noSQL-db that natively ingests a wide range of document types and data formats. Lexical search is based not on inverted indexes but on Pat Arrays with a two-tier address-based caching system that solved its main limitations. Dense vector search is handled by HNSW. The implementation is based on the HNSWlib reference but heavily enhanced and turbo-charged (using SIMD instructions).

Our HNSWlib fork supports a myriad of features including Binary, Ternary, INT4, INT8 and INT16 quantization using a number of algorithms including RaBitQ and RaBitQ-extended for binary. We also support pass-through to optimally use pre-quantized models (BIN1, INT2, INT3, INT4, FP4, INT5, INT6, INT8, INT16, FP16, BF16, FLOAT32). We naturally also provide an optional memory mapped re-scoring.

Behind our embeddings is our fork of bert.cpp and the mainlined llama.cpp. Behind these is the GGML tensor library. Unlike mainstream vendor libraries (like PyTorch, TensorFlow, or TensorRT) built for data-center GPUs, GGML runs large transformer models efficiently on commodity hardware.



1. Superior CPU and Edge Device Execution.

Mainstream libraries rely heavily on massive GPU VRAM to hold model weights during inference. GGML is written in plain C and C++, making it highly portable.

- Zero Dependencies:** It requires no heavy third-party stacks (such as CUDA or ROCm) just to run on a standard machine.
- Hardware Interoperability:** It is optimized for x86 architectures (AVX, AVX2, AVX-512) and Apple Silicon (via native ARM NEON and Metal).
- System RAM Utilization:** Because most modern consumer computers feature far more system RAM than GPU VRAM, GGML natively bridges the gap by enabling models to run efficiently on standard CPUs or via CPU-GPU offloading.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

2. **Accelerators:** GGML features a highly modular backend architecture. This layout allows it to offload computation graph execution to a variety of hardware accelerators via specific hardware drivers, completely bypassing heavy Python stacks.

3. **Advanced, Fine-Grained Quantization:** One of GGML's most compelling features is its pioneering approach to model quantization (compressing floating-point numbers into lower bit-widths, such as 4-bit or 5-bit).

- *K-Quants:* GGML utilizes highly engineered, variable bit-width quantization strategies (like Q4_K_M) that minimize the traditional loss in output accuracy.
- *Memory Compression:* It shrinks the memory footprint of massive models by up to 70-80% without heavily compromising the model's perplexity or reasoning capabilities.

4. **Streamlined Portability** (The GGUF Standard). GGML's associated file format, GGUF, was specifically designed to solve the fragmentation and loading issues of older ML formats.

5. **No Python Overhead**

6. **Specialized Text-Generation Ecosystem**

While mainstream libraries like TensorRT or PyTorch remain undisputed for model training and high-throughput server farms, GGML democratizes access to AI by allowing us to run powerful models locally on mainstream hardware.

While we build upon a heavily refactored bert.cpp we also support llama.cpp for more modern sophisticated models that don't use the BERT architecture such as EuroBERT.

Appendix V: Accelerators

Our lexical algorithms and HNSW implementation is, by design, CPU bound (the later using heavily optimized SIMD instructions) to leave the accelerators free to do the heavy lifting of running the models. Even on an Apple M1 (Pro) we've clocked 13k QPS-- on an M5 Pro we expect 30,000 to 39,000+ QPS-- so we found ample state-of-the-art performance already on CPUs.

1. NVIDIA CUDA

- *Quality:* This is the most mature, heavily optimized backend in the GGML ecosystem. It receives immediate day-one updates for new architectures.
- *Performance:* Offers the highest token-per-second generation rates. It handles massive context windows and batched requests effortlessly.
- *Efficiency:* While desktop NVIDIA cards draw significant wattage, the work-per-watt remains high because the GPU completes the generation sequence rapidly and returns to idle.

2. Apple Metal

- *Quality:* Flawless integration natively recognized by macOS and iOS.
- *Performance:* Extremely rapid. Because Apple Silicon uses a Unified Memory Architecture, the CPU and GPU share the same physical RAM pool. GGML leverages this via mmap, allowing a model to load instantly without a slow transfer over a PCIe bus.
- *Efficiency:* Unmatched. Users can run quantized 70B models locally on a MacBook with minimal heat production, preserving battery life during background operations.

3. Vulkan (The Compatibility Champion)

- *Quality:* Highly robust and serves as the best open-source, vendor-agnostic fallback.
- *Performance:* While it slightly lags behind native CUDA or ROCm tweaks, it delivers impressive matrix-multiplication speeds across diverse hardware, including Steam Decks, Raspberry Pis, and mixed-GPU setups.
- *Efficiency:* Balanced. Vulkan provides hardware acceleration on low-power devices that lack dedicated AI chips.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Appendix VI: Vector search performance

Speed matters because RAG pipelines are dominated by retrieval latency. Faster retrieval can dramatically improve RAG pipelines

Benchmarking (as a baseline for a minimal probable use) on:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro18,1
Model Number:	MK183D/A
Chip:	Apple M1 Pro
Total Number of Cores:	10 (8 performance and 2 efficiency)
Memory:	16 GB
System Firmware Version:	11881.140.96
OS Loader Version:	11881.140.96

System	Approx QPS (768d)	Notes
Qdrant	~1k–3k	depends heavily on HNSW tuning
Milvus	~2k–5k	heavier infra
FAISS	~5k–10k	optimized C++ but limited system features
Elasticsearch	usually <2k	heavy overhead
Quarry HNSW	13k	C++.

Comparing to FAISS and single thread:

Index type	Dataset size	QPS (1 thread)	Notes
Flat (exact search)	1M vectors	40–120 QPS	brute-force dot products
IVFFlat	1M vectors	150–400 QPS	typical ANN config
IVFPQ	1M vectors	300–900 QPS	quantized vectors
HNSW (FAISS)	1M vectors	200–600 QPS	depends heavily on efSearch
Quarry HNSW	1M vectors	3000 QPS	Faster for quantized vectors

The C++ HNSWlib was already one of the fastest HNSW implementations on the market. We kept to CPU only (to allow better utilization of the accelerators for the embedding models) but turbo-charged it to be 4-6x faster with Fp32 on ARM hardware and expanded its support for larger datasets through the reduction of memory demands through quantization. Comparing our optimized version with a straight compile (Clang without -O) the difference was 20x on an Apple M1. We also added support for pre-quantised models. These require not just much less memory but are also significantly faster than Fp32 models—since we implemented pass-through our HNSW graph networks also demand far less memory than solutions that pad to Fp32.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Let's look at some of our quantization benchmarks (again on the M1Pro)

=== Quantization Benchmark ===

Dimension: 384

Data points: 10000

Queries: 1000

k: 10

=== Results ===

Mode	Build (ms)	Query (ms)	Recall	Bytes/Vec	Memory (KB)
INT8-STANDARD	4499.7	93.75	0.8684	384	3750
INT8-CENTROID	4503.7	92.65	0.8266	384	3750
INT8-ROTATIONAL	4990.0	123.18	0.8644	384	3750
INT4-STANDARD	6123.0	136.67	0.8919	192	1875
INT4-ROTATIONAL	7007.3	159.96	0.8837	192	1875
BIN1-STANDARD	676.3	6.79	0.8637	48	468
BIN1-BETTER	582.8	5.05	0.7379	48	468
BIN1-CENTROID	649.6	7.14	0.8508	48	468
BIN1-ROTATIONAL	1482.0	36.56	0.7328	48	468
BIN1-RABITQ	783.6	8.68	0.9900	112	1093
BIN1-RABITQ-EXT	973.7	17.25	0.9944	304	2968

Appendix VII: Vectorization performance

We are developing and benchmarking using the M1 as a baseline upon which to base performance expectations. Instead of demanding the latest and greatest (or an array of H200s) our design goals and constraints are to provide useful performance on literally "the smallest machine possible"—and not to demand some hyperscalar cluster.

model: sentence-transformers_all-MiniLM-L12-v2

main: token time = 0.03 ms / 0.01 ms per token

main: eval time = 15.07 ms / 3.77 ms per token

model: sentence-transformers_all-MiniLM-L12-v2-f32.gguf

main: token time = 0.02 ms / 0.00 ms per token

main: eval time = 21.52 ms / 5.38 ms per token

model: bge-large-en-v1.5-q4_k_m.gguf

main: token time = 0.03 ms / 0.01 ms per token

main: eval time = 30.91 ms / 7.73 ms per token

Turning now to a 335-million parameter model and more exhaustive production-like benchmarks:

Model: bge-large-en-v1.5 arch=bert n_embd=1024 load=150 ms Warmup=5 iters, measure=50 iters

== latency by input length (batch=1) ==

very-short tok=11 bs= 1 | mean= 17.47 ms med= 17.47 p95= 18.01 min= 16.93 max= 18.38 | 630

tok/s 57.3 seq/s

short tok=38 bs= 1 | mean= 23.97 ms med= 23.83 p95= 24.92 min= 22.97 max= 25.14 | 1586

tok/s 41.7 seq/s

medium tok=137 bs= 1 | mean= 44.67 ms med= 44.41 p95= 46.24 min= 43.39 max= 46.92 | 3067

tok/s 22.4 seq/s

long tok=263 bs= 1 | mean= 78.83 ms med= 78.07 p95= 81.44 min= 76.53 max= 82.25 | 3336

tok/s 12.7 seq/s

max tok=512 bs= 1 | mean= 132.63 ms med= 132.67 p95= 133.81 min= 130.45 max= 134.25 | 3860

tok/s 7.5 seq/s

== throughput by batch size (tok=128) ==

bs=1 tok=137 bs= 1 | mean= 44.20 ms med= 43.95 p95= 45.58 min= 43.41 max= 46.02 | 3100

tok/s 22.6 seq/s

bs=4 tok=137 bs= 4 | mean= 161.03 ms med= 160.01 p95= 166.50 min= 156.61 max= 167.15 | 3403

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

```
tok/s 24.8 seq/s
bs=8 tok=137 bs= 8 | mean= 314.57 ms med= 313.94 p95= 324.07 min=291.38 max=328.69 | 3484
tok/s 25.4 seq/s
```

Looking at the 8-bit model we can see its even faster:

Loading /Users/edz/Downloads/bge-large-en-v1.5-q8_0.gguf on accelerated backend (threads=6)...

Model: bge-large-en-v1.5 arch=bert n_embd=1024 load=220 ms

Warmup=5 iters, measure=50 iters

== latency by input length (batch=1) ==

```
very-short tok= 11 bs= 1 | mean= 14.82 ms med= 14.82 p95= 14.94 min= 14.60 max= 15.05 | 742
```

```
tok/s 67.5 seq/s
```

```
short tok= 38 bs= 1 | mean= 20.79 ms med= 20.71 p95= 20.98 min= 20.47 max= 22.99 | 1828
```

```
tok/s 48.1 seq/s
```

```
medium tok=137 bs= 1 | mean= 38.63 ms med= 38.62 p95= 38.74 min= 38.54 max= 38.86 | 3546
```

```
tok/s 25.9 seq/s
```

```
long tok=263 bs= 1 | mean= 67.24 ms med= 67.21 p95= 67.38 min= 67.12 max= 67.55 | 3911 tok/s
```

```
14.9 seq/s
```

```
max tok=512 bs= 1 | mean= 116.45 ms med= 116.62 p95= 117.12 min= 115.65 max= 117.53 | 4397
```

```
tok/s 8.6 seq/s
```

== throughput by batch size (tok=128) ==

```
bs=1 tok=137 bs= 1 | mean= 38.63 ms med= 38.61 p95= 38.80 min= 38.49 max= 39.08 | 3546
```

```
tok/s 25.9 seq/s
```

```
bs=4 tok=137 bs= 4 | mean= 135.44 ms med= 135.41 p95= 135.74 min= 135.21 max= 136.05 | 4046
```

```
tok/s 29.5 seq/s
```

```
bs=8 tok=137 bs= 8 | mean= 266.41 ms med= 266.36 p95= 266.81 min= 266.07 max= 266.91 | 4114
```

```
tok/s 30.0 seq/s
```

```
bs=16 tok=137 bs= 16 | mean= 530.99 ms med= 529.86 p95= 536.17 min= 529.20 max= 539.27 |
```

```
4128 tok/s 30.1 seq/s
```

```
bs=32 tok=137 bs= 32 | mean= 1069.06 ms med= 1067.57 p95= 1074.02 min= 1066.46 max= 1085.44 |
```

```
4101 tok/s 29.9 seq/s
```

Now to the FP16 m3 model

Loading ../models/bge-m3-FP16.gguf on accelerated backend (threads=6)...

Model: arch=bert n_embd=1024 load=698 ms

Warmup=5 iters, measure=50 iters

== latency by input length (batch=1) ==

```
very-short tok= 11 bs= 1 | mean= 15.24 ms med= 15.25 p95= 15.31 min= 14.91 max= 15.33 | 722
```

```
tok/s 65.6 seq/s
```

```
short tok= 38 bs= 1 | mean= 20.60 ms med= 20.62 p95= 20.74 min= 20.04 max= 20.89 | 1845
```

```
tok/s 48.5 seq/s
```

```
medium tok=137 bs= 1 | mean= 33.48 ms med= 33.52 p95= 33.58 min= 33.03 max= 33.69 | 4091
```

```
tok/s 29.9 seq/s
```

```
long tok=263 bs= 1 | mean= 55.91 ms med= 55.96 p95= 56.07 min= 55.32 max= 56.10 | 4704
```

```
tok/s 17.9 seq/s
```

```
max tok=524 bs= 1 | mean= 103.53 ms med= 103.62 p95= 103.76 min= 102.61 max= 104.00 | 5061
```

```
tok/s 9.7 seq/s
```

== throughput by batch size (tok=128) ==

```
bs=1 tok=137 bs= 1 | mean= 33.48 ms med= 33.52 p95= 33.57 min= 32.81 max= 33.60 | 4092
```

```
tok/s 29.9 seq/s
```

```
bs=4 tok=137 bs= 4 | mean= 115.73 ms med= 115.77 p95= 115.89 min= 115.17 max= 115.97 | 4735
```

```
tok/s 34.6 seq/s
```

```
bs=8 tok=137 bs= 8 | mean= 223.99 ms med= 224.31 p95= 224.56 min= 222.96 max= 224.72 | 4893
```

```
tok/s 35.7 seq/s
```

```
bs=16 tok=137 bs= 16 | mean= 439.74 ms med= 439.56 p95= 439.90 min= 438.38 max= 458.65 |
```

```
4985 tok/s 36.4 seq/s
```

```
bs=32 tok=137 bs= 32 | mean= 930.68 ms med= 926.28 p95= 952.28 min= 915.63 max= 985.46 |
```

```
4711 tok/s 34.4 seq/s
```

Because throughput peaks in Q4 at around 3,484 tokens/sec (0.29 ms per token) for Q4K we use these

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

metrics to calculate some real-world document ingestion limits. If you block your raw text files into standard overlapping paragraph windows of roughly 256 tokens each: A single passage will compute in ~78 ms.

Our test M1-Pro chews through roughly 45 passages per second per instance (3484 tok/s ÷ 78 ms). That means one can expect to process 2,700 fully dense semantic records per minute on a baseline Apple Silicon chip. Our tests on M3Pro and M4Pro showed even significantly higher throughputs (80k tokens/s or as much as 20x).

- M1 Pro (ARMv8): Uses an older iteration of Apple's proprietary AMX (Apple Matrix Coprocessor). It is highly optimized for accelerating single-sequence matrix-vector math (bs=1).
- M4 Family (ARMv9 + SME): The M4 introduced SME (Scalable Matrix Extension). SME is explicitly designed to handle outer-product matrix-matrix tiles natively in hardware. When you increase the batch size, the mathematical operation changes from a series of matrix-vector multiplies into a massive matrix-matrix multiplication (GEMM). The M4's SME hardware can swallow those batched matrix tiles concurrently, executing them multiple times faster per clock cycle than the M1 Pro's AMX.
- An M4 Pro has up to 20 next-generation GPU cores, and an M4 Max has up to 40. Combined with a memory bandwidth of up to 546 GB/s, the M4 Max has a massive parallel execution pool.

With a base M4 Mac mini, we can expect benchmarking metrics something like this:

- Single Query Latency (bs=1, tok=38): Will drop from your current ~24 ms down to under 8 ms, making live UI search-as-you-type operations completely instantaneous.
- Peak Batch Throughput: Will scale up from your current 3,484 tokens/second to a sustained bracket of 12,000 to 15,000 tokens/second.

The Projected M5 Performance Matrix:

- Running bge-large-en-v1.5 on a cheap M5 Mac mini will yield metrics that look more like a dedicated server data center than a compact desktop:
- Interactive Query Latency (bs=1, short query): ~3 to 5 ms (making search-as-you-type feel physically instantaneous).
- Peak Batch Throughput: Easily hitting 45,000 to 60,000+ tokens/second.

At 60,000 tokens per second, a single entry-level M5 mini will let you index up to 14,000 full paragraphs every single minute.

As mentioned we support also Vulkan and CUDA. As a reference machine for CUDA we've chosen the NVIDIA Jetson AGX Orin. It is an edge AI module delivering up to 275 TOPS of performance. It features configurable power consumption ranging from **15W to 75W** depending on the model, and comes in 32GB and 64GB System-on-Module (SOM) configurations.

```
ggml_cuda_init: found 1 CUDA devices (Total VRAM: 62796 MiB):
```

```
Device 0: Orin, compute capability 8.7, VMM: yes, VRAM: 62796 MiB
```

```
CUDA Device 0: Orin (65.85 GB VRAM)
```

```
bert: CUDA backend does not support get_rows for q4_K – embedding lookups will run on CPU, all other ops on CUDA
```

```
Model: bge-large-en-v1.5 arch=bert n_embd=1024 load=817 ms
```

```
Warmup=5 iters, measure=50 iters
```

```
== latency by input length (batch=1) ==
```

```
very-short tok=11 bs=1 | mean= 9.32 ms med= 8.36 p95= 12.36 min= 7.04 max= 13.72 | 1181 tok/s 107.3 seq/s
```

```
short tok=38 bs=1 | mean= 10.31 ms med= 10.21 p95= 10.93 min= 9.85 max= 11.13 | 3686 tok/s 97.0 seq/s
```

```
medium tok=137 bs=1 | mean= 19.40 ms med= 19.35 p95= 19.90 min= 18.88 max= 20.64 | 7061 tok/s 51.5 seq/s
```

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

```
long      tok=263 bs= 1 | mean= 35.19 ms med= 35.15 p95= 35.73 min= 33.71 max= 36.83 | 7474
tok/s    28.4 seq/s
max      tok=512 bs= 1 | mean= 59.82 ms med= 59.79 p95= 60.74 min= 58.65 max= 61.31 | 8559
tok/s    16.7 seq/s
```

== throughput by batch size (tok=128) ==

```
bs=1      tok=137 bs= 1 | mean= 19.36 ms med= 19.32 p95= 19.82 min= 18.86 max= 20.60 | 7076
tok/s    51.6 seq/s
bs=4      tok=137 bs= 4 | mean= 67.49 ms med= 67.50 p95= 68.38 min= 65.81 max= 68.95 | 8119
tok/s    59.3 seq/s
bs=8      tok=137 bs= 8 | mean= 135.10 ms med= 134.98 p95= 136.11 min=133.79 max=137.25 | 8113
tok/s    59.2 seq/s
bs=16     tok=137 bs=16 | mean= 262.59 ms med= 262.67 p95= 263.98 min=260.84 max=264.76 |
8348 tok/s 60.9 seq/s
bs=32     tok=137 bs=32 | mean= 520.58 ms med= 520.57 p95= 521.90 min=518.51 max=522.87 |
8421 tok/s 61.5 seq/s
```

Notice the bge-large-en-v1.5 q4_K is currently not wholly supported by the ggml CUDA backend. We still despite letting the embedding lookup run on the CPU we still get ~8k tok/s. That said it still chews through on this model more than 2x the speed of the M1Pro or more than 150 passages per second per instance.

Looking at the Q8 model on the Orin:

```
ggml_cuda_init: found 1 CUDA devices (Total VRAM: 62796 MiB):
Device 0: Orin, compute capability 8.7, VMM: yes, VRAM: 62796 MiB
CUDA Device 0: Orin (65.85 GB VRAM)
bert: using CUDA backend (full get_rows support)
Model: bge-large-en-v1.5 arch=bert n_embd=1024 load=1014 ms
Warmup=5 iters, measure=50 iters
```

== latency by input length (batch=1) ==

```
very-short tok= 11 bs= 1 | mean= 9.97 ms med= 8.59 p95= 14.95 min= 7.27 max= 20.27 | 1103
tok/s    100.3 seq/s
short     tok= 38 bs= 1 | mean= 9.72 ms med= 9.81 p95= 10.02 min= 9.21 max= 10.31 | 3911 tok/s
102.9 seq/s
medium    tok=137 bs= 1 | mean= 16.37 ms med= 16.33 p95= 16.53 min= 15.66 max= 17.54 | 8368
tok/s    61.1 seq/s
long      tok=263 bs= 1 | mean= 27.97 ms med= 28.00 p95= 28.27 min= 27.18 max= 29.45 | 9402
tok/s    35.8 seq/s
max      tok=512 bs= 1 | mean= 46.53 ms med= 46.43 p95= 47.57 min= 45.86 max= 48.14 | 11004
tok/s    21.5 seq/s
```

== throughput by batch size (tok=128) ==

```
bs=1      tok=137 bs= 1 | mean= 16.16 ms med= 16.24 p95= 16.64 min= 15.59 max= 18.18 | 8477
tok/s    61.9 seq/s
bs=4      tok=137 bs= 4 | mean= 55.13 ms med= 55.10 p95= 55.39 min= 54.70 max= 56.46 | 9941
tok/s    72.6 seq/s
bs=8      tok=137 bs= 8 | mean= 113.15 ms med= 112.92 p95= 114.58 min=112.46 max=115.63 | 9686
tok/s    70.7 seq/s
bs=16     tok=137 bs=16 | mean= 215.03 ms med= 214.71 p95= 216.90 min=214.31 max=218.90 | 10194
tok/s    74.4 seq/s
bs=32     tok=137 bs=32 | mean= 427.06 ms med= 427.35 p95= 428.06 min=425.14 max=428.74 |
10265 tok/s 74.9 seq/s
```

Appendix VII: Lexical Search Performance (including non-vector objects)

As a test we'll index some Wikipedia Cirrus (ES Bulk) dumps (a kind of JSON)

Index [Notice]: Added 753423571 words (5871536 unique) in 2065536 records in 1439 sec. [indexed 32866k words/min, 92267 records/min (CPU), 4096MB Memory]

Main Job Statistics:

```
Elapsed "real" time: 1439 seconds
CPU time:          1343.43 seconds
User time:         1160.39 seconds
```

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

System time: 183.037 seconds
CPU/Real: 93.3584%
Max resident size: 10950288k
Page reclaims: 32038440
faults: 1586471
Swaps: 0
Context switches Vol.: 11969
Invol.: 2569882
Total records added: 2065536
Total words added: 753423571

edz@MacBookPro build % ../bin/Isearch -d /Volumes/D50/indexes/WIKI_ALL covid vaccine AND:text
Process time: 24 ms. (2) Search: 22 ms. (10.9 ms/term)
Query: "covid" "vaccine" AND:text

39 records of 2065536 matched your query, 39 records displayed.

Score File

1. 100 /Volumes/D50/Wikidata/enwikisource_content-20260621-00000.json
'National Strategy for the COVID-19 Response and Pandemic Preparedness'

edz@MacBookPro build % ../bin/Isearch -d /Volumes/D50/indexes/WIKI_ALL hamlet
Process time: 10 ms. (3) Search: 6 ms.
Query: "hamlet"

1831 records of 2065536 matched your query

edz@MacBookPro build % ../bin/Isearch -d /tmp/WIKI william shakespeare
Process time: 4 ms. (1) Search: 3 ms.
Query: "william shakespeare"

59 records of 609088 matched your query, 59 records displayed.

edz@MacBookPro build % ../bin/Isearch -d /Volumes/D50/indexes/WIKI_ALL quantum physics
Process time: 10 ms. (3) Search: 7 ms.
Query: "quantum physics"

3 records of 2065536 matched your query

edz@MacBookPro build % ../bin/Isearch -d /tmp/WIKI quantum physics
Process time: 3 ms. (1) Search: 2 ms.
Query: "quantum physics"

1 record of 609088 matched your query, 1 record displayed.

edz@MacBookPro build % ../bin/Isearch -d /tmp/WIKI -show quantum computer
Process time: 7 ms. (1) Search: 5 ms. (3.1 ms/term)
Query: "quantum" "computer" PEER

4 records of 609088 matched your query, 4 records displayed.

edz@MacBookPro build % ../bin/Isearch -d /tmp/WIKI -max 2 great composers from austria
Process time: 12 ms. (1) Search: 11 ms. (4.8 ms/term)
Query: "great" "composers" || "from" || "austria" || FOCUS:4

214 records of 609088 matched your query, 2 records displayed.

edz@MacBookPro build % ../bin/Isearch -d /tmp/WIKI -max 2 willian shakespeare is a
Process time: 197 ms. (1) Search: 196 ms. (84.2 ms/term)
Query: "willian" "shakespeare" || "is" || "a" || FOCUS:4

58808 records of 609088 matched your query, 2 records displayed.

edz@MacBookPro build % ../bin/Isearch -d /tmp/WIKI -max 2 -show "*dent" "US?"
Process time: 182 ms. (1) Search: 181 ms. (90.3 ms/term)
Query: "*dent" "US?" PEER

194 records of 609088 matched your query, 2 records displayed.

edz@MacBookPro build % ../bin/Isearch -d /tmp/WIKI -max 2 -show "*dent" "vi?e"
Process time: 146 ms. (2) Search: 144 ms. (72.0 ms/term)
Query: "*dent" "vi?e" PEER

51 records of 609088 matched your query, 2 records displayed.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

```
edz@MacBookPro build % ../bin/Isearch -d /tmp/WIKI -max 2 -show vice president
Process time: 4 ms. (1) Search: 3 ms.
Query: "vice president"
```

9 records of 609088 matched your query, 2 records displayed.

NOTE: The above numbers are just a snapshot (progress) and changing.

CoreQuarry: Local-First Knowledge Retrieval Kernel for Humans and AI Agents.

Appendix VIII: Future directions: CoreQuarry For Minority Languages (Language Machine)

In future work we are looking at also developing embedding models for under-represented languages, primarily European or classical humanist (Hebrew, Aramaic, Arabic) . This is imperative as we live in a world of Beige coloured by English.

Let's separate the problem into layers, because each one needs a different fix:

Layer 1: tokenizer vocabulary allocation. BPE/SentencePiece vocabularies are trained on web-scraped corpora, so merge-rule budget gets allocated roughly proportional to how much of each language is on the web — which tracks colonial economic history more than speaker count. Petrov et al. (2023) quantified this as up to 15× sequence-length inflation across languages for the same content. A minority language gets segmented down to near-byte level by a vocabulary that never learned its morphology, while English gets efficient whole-word tokens — so the model sees far less signal per training token for that language, uses more compute per unit of meaning at inference, and (this is the part that compounds) gets worse embeddings because there's less structure for the model to latch onto.

Layer 2: raw training-corpus imbalance. This is the one people conflate with Layer 1 but it's separate — and it's the uncomfortable finding in that same Petrov work: the length-inflation effect persists even for byte- and character-level tokenizers. Going byte-level removes the vocabulary-allocation unfairness, but a byte-level model trained on the same web-scraped mix still sees overwhelmingly more English bytes, so it still learns English-shaped patterns best. Tokenizer-free doesn't buy you fairness for free — it just moves the bottleneck from "vocabulary budget" to "actual token count in training," which you can only fix with corpus composition, not architecture.

Layer 3 (specific to embeddings): cross-lingual alignment needs parallel data. LaBSE, LASER, BGE-M3 — the whole multilingual sentence-embedding lineage — align embedding spaces across languages using translation pairs, not just monolingual text. That's a second, independent resource bottleneck: a language can have decent monolingual text and still be embedding-poor if nobody's translated much into or out of it. This is exactly why Bible corpora keep showing up as an evaluation/training resource in this literature — for a lot of genuinely small languages, systematic parallel translation essentially only exists because of Bible translation efforts. That's its own uncomfortable fact about the field.

We are building a retrieval kernel, not a generator, and retrieval has a much lower bar than fluent generation — you need a good enough semantic neighborhood, not correct output text. That's mechanically why embedding models have covered "100+ languages" more credibly than generative LLMs have. And we have already done the Layer-1 work conceptually, just at the classical-IR layer instead of the neural layer — ParseWords has zero vocabulary to be unfair, by construction. The current research response to Layer 1 for neural models (Meta's Byte Latent Transformer, and more specifically MYTE, which is explicitly framed as "morphology-driven byte encoding aimed at multilingual fairness") is the same instinct you've been applying by hand to Greek/Hebrew/Arabic/Syriac case-folding — work at the byte level so there's no learned vocabulary sitting there encoding whose morphology got more training budget.

Given our actual script cluster (Latin/Greek/Cyrillic/Armenian/Georgian + Hebrew/Aramaic/Arabic), the natural framing isn't "build one more 100-language model" — it's a small, curated, byte-or-character-level encoder trained specifically on that cluster, where Layer 2's data-imbalance problem becomes something you can actually control the composition of, rather than inheriting from whatever a web crawl happened to contain.