ChatGPT

Feasibility of a Two-Tier Vector Search System

The idea of decoupling storage (cheap, slow) from compute (fast, memory-heavy) is well established for tabular/analytics data, and AWS is now extending it to vectors. Amazon S3 Vectors is a new cloud service that lets you store **billions** of vectors at very low cost and still perform *sub-second* similarity queries [1]. However, S3 Vectors alone cannot meet tight real-time latency needs (e.g. <100 ms for chatbots or fraud detection). In practice, you **layer** a fast in-memory index/cache on top of S3 storage. As one recent article explains, "S3 Vectors lets you store huge volumes of vector data at low cost… but you'll still need a lightweight layer on top to meet real-time latency requirements" [2]. In that architecture, a stateless *router* or query service holds an **in-memory index** (e.g. FAISS/JECQ) to quickly find candidate IDs, then retrieves the actual vectors from a RAM cache (e.g. Redis) or falls back to S3 if needed [3]. This multi-tier design is feasible and has been proposed by AWS and others as a cost-effective compromise: most data lives in cheap S3, while a "hot subset" is kept in fast RAM [2] [3].

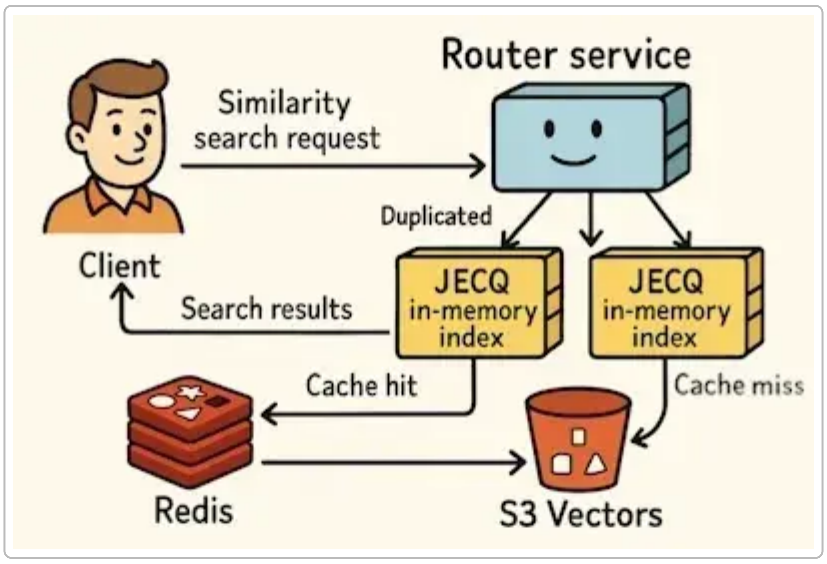

Figure: Example architecture from a recent blog – a query router uses an in-memory vector index (JECQ/FAISS) to pick candidate IDs, then checks a Redis cache for their raw vectors (cache hit) or fetches from S3 Vectors on a miss [3].

A layered cache **makes sense** for performance/scalability. Numerous sources confirm this pattern: for example, a Janea Systems post explicitly describes using S3 Vectors as a "durable, budget-friendly foundation" and keeping a few percent of vectors in RAM (e.g. 1% of 10B vectors = 300 GB) to speed up searches [2] [4]. In research on disk-based vector search (e.g. DiskANN), cache hit rates are crucial for latency, and authors propose grouping queries and prefetching to boost cache efficiency [5] [6]. In short, a fast in-memory "cache" layer over a large S3 store is a known and practical approach to balance low cost and low latency.

Choosing an Open-Source Vector Engine

You'll want an **open-source, in-memory, high-performance** vector store or index that you can modify. Good candidates include:

- **Redis/Valkey with vector search**. Redis (and its new fork Valkey) is a mature in-memory key-value store. Redis now supports vector similarity queries via modules (Redisearch) or the new Valkey-Search plugin. Valkey-Search is a BSD-licensed Redis module that "allows you to efficiently create indexes and search through billions of vectors" [7]. Redis/Valkey is written in C for speed, easily runs locally or on AWS, and is cache-oriented by design. You can customize eviction policies or integrate with your code. Because you're comfortable with Python, you can use `redis-py` or similar clients to interact with it. The tradeoff is that Redis isn't *designed* for very large datasets (you must provision enough RAM), but that's fine since most data stays in S3.

- **FAISS or JECQ (C++ libraries)**. Facebook's FAISS is a high-performance ANN library (C++) with Python bindings [8]. It isn't a full database, but you can use it as the in-memory index. FAISS supports many index types (IVF, HNSW, PQ, etc.), runs queries very fast, and scales to billions of vectors. (A newer variant, **JECQ**, is a "drop-in replacement with 6× lower memory usage" that we use in the Janea example [9].) You would then store the raw vectors separately (e.g. in Redis or a simple Python dict) for re-ranking. This approach gives fine-grained control but requires glue code to tie FAISS+cache together.

- **Dedicated vector databases** (Milvus, Qdrant, Weaviate, Chroma, etc.). These are full-featured engines (often using disk storage, with optional in-memory caches). For example, Qdrant (Rust) and Milvus (C++) are fast and open-source, but they're complex to modify on short notice. They may also already manage their own persistence (disk or S3). Weaviate (Go) and Chroma (Python) are easier to get started but may lack the ultra-high performance of FAISS or Redis. Given the weekend/POC scope, a **lighter solution like Redis+FAISS** might be simpler. If you're open to learning Rust, Qdrant is production-ready and has a REST/GRPC API, but customizing its internals would take time.

In summary, a sensible stack for a prototype is: use **Redis/Valkey** as the in-memory cache (leveraging its new vector-search module [7]), together with a fast in-memory ANN index library (FAISS or JECQ) for nearest-neighbor lookup. This covers the "really fast, in-memory" requirement. You can implement this in Python (using FAISS's Python API and a Redis client) or in Go/Rust if you prefer. Even in Python, the heavy lifting (ANN search, vector math) happens in C/C++ anyway.

Cache Eviction and Retrieval Logic

The core algorithms you need are basically **cache eviction** for which vectors to keep, and **query logic** for choosing memory vs S3. Here are guiding principles:

- **Eviction policy**: Since the cache is finite, you'll likely use a standard policy like LRU (evict least-recently-used) or LFU (least-frequently-used) on your in-memory store. Redis natively supports LRU/LFU eviction if configured. The Janea blog even suggests "advanced cache management: store frequent queries, support MFU/LFU/LRU caching strategies, and pre-load data based on user behavior" [10]. For a simple POC, you could just let Redis expire old keys or evict by LRU. Alternatively, you could proactively "pin" certain vectors (e.g. cluster centroids or the most popular items) if you identify heavy hitters from logs.

- **Query routing logic**: On each similarity query, first use your in-memory index to get candidate vector IDs. Then do something like:

- **Cache lookup**: Check the in-memory cache (Redis) for those IDs. If a vector is present, retrieve it directly. (This is the "cache hit" case.)

- **Cache miss handling**: For any IDs *not* in the cache, fetch their vector values from S3 (or S3 Vectors) and return those. You can also insert these fetched vectors into the cache (possibly with a TTL or eviction priority) so that future queries will hit them.

This is exactly the scheme described in the example architecture: "The router uses [the index] to find candidate vector IDs. It then checks Redis for raw vectors: *Cache hit*: Redis returns vectors… *Cache miss*: Router pulls vectors from S3, re-ranks, and returns results" [3] .

You might also apply a quality check: if the top-K neighbors from cache alone already meet your quality/recall threshold, you could skip fetching extra vectors from S3. For example, if the nearest cached vectors have cosine similarity above a certain cutoff, assume the answer is "good enough." Otherwise, fetch more from S3. This kind of thresholding can reduce S3 I/O. In practice, a simple approach is to always fetch missing vectors (as above), and optimize later if needed.

- **Index compression** (optional): For speed/memory, you might use **Product Quantization** (PQ) or HNSW indexes in FAISS/JECQ to reduce the size of the in-memory index [11] [9] . PQ, for example, compresses 768-dim vectors into compact codes and accelerates distance computations [11] . This helps when the number of total vectors (billions) is huge. However, implementing PQ is more advanced and may be optional for a POC; FAISS can auto-tune or you can start with a simpler IVF or HNSW index.

In summary, your retrieval algorithm is essentially: *"search index → get IDs → lookup in cache → fetch missing from S3"*. For cache eviction, start with a proven policy like LRU/LFU. As one source notes, improved cache strategies (e.g. query grouping, prefetching popular clusters [5]) exist but can be added later. Initially, simple usage-based eviction should suffice.

Testing and Benchmarking

For an end-to-end prototype, set up a mini environment that mimics the intended architecture and measure its behavior. Here are steps:

1. **Synthetic dataset**: Generate or obtain a set of vectors (e.g. 100K–1M vectors, 128–768 dimensions). You can use random data or a public dataset (like SIFT, word embeddings, or image embeddings). Split it into "hot" vs "cold" subsets (say 5–20% hot in cache, rest cold in S3).

2. **Infrastructure**: Run your chosen in-memory store (e.g. Redis/Valkey) locally or in Docker. Store the hot vectors there (or have your code fill the cache on first access). Simulate S3 by either using a real S3 bucket (free tier or MinIO local) or simply storing vectors in files/objects keyed by ID.

3. **Indexing**: Build the ANN index (FAISS or valkey-search) on the *full* dataset of IDs. The index does *not* need raw vectors except to build distance tables; it only outputs IDs.

4. **Query testing**: Write a query loop that picks random or realistic query vectors and does the two-phase search (index → cache/S3). Measure:

5. **Latency**: time from query to results (break down into index time, cache fetch time, S3 fetch time).
6. **Hit rate**: fraction of needed vectors served by cache vs fetched from S3.
7. **Recall/accuracy** (optional): compare results to a ground-truth search if using approximate indexes.

8. **Resource usage**: memory used by cache, number of S3 read operations, etc.

9. **Benchmarks**: Record average and percentile (e.g. 95th) latencies for queries with and without cache, and under different cache sizes. Check how hit rate improves response time. This will show the trade-offs. For example, you might find that when 20% of vectors are cached, 90% of queries hit entirely in cache, giving very low latency; when cache is smaller, more S3 I/O occurs, slowing queries.

10. **AWS testing**: Finally, if feasible, deploy on AWS. For a weekend POC you could use an EC2 (or local computer) running Redis, and an S3 Vector bucket (if you have preview access) or a regular S3 bucket. Use the AWS SDK (Boto3) to store/query vectors in S3 Vectors and measure actual query latencies. Optionally, you could also try the AWS "OpenSearch Vector" approach (the blog suggests moving infrequently accessed vectors to S3 and keeping hot ones in OpenSearch [12]), but that is more involved.

While the connected sources don't give specific testing scripts, they do emphasize measuring performance. For example, Janea's article includes a cost comparison and discusses search speed-ups [13]. Use similar metrics: query **throughput** (queries/sec), **P95/P99 latency**, and **cache hit ratio**. Even a simple Python script timing each query (using `time.time()` around your search logic) will give you these numbers.

In summary, test by simulating queries against both the cache+index and pure S3 scenarios, logging how often you access S3 vs memory and how that affects latency. This will validate that your two-tier system meets performance goals. If needed, you can adjust cache size or eviction policy and re-run benchmarks to find a good balance between speed and memory usage.

Sources: Recent AWS and industry blogs outline this multi-tier design and its benefits [1] [2] [3]. Vector database reviews and module docs also note Redis/Valkey and FAISS as top open-source tools [7] [14], and cache management strategies (LRU/LFU) are well-known for such use cases [10].

---

[1] [12] Introducing Amazon S3 Vectors: First cloud storage with native vector support at scale (preview) | AWS News Blog
https://aws.amazon.com/blogs/aws/introducing-amazon-s3-vectors-first-cloud-storage-with-native-vector-support-at-scale/

[2] [3] [4] [9] [10] [11] [13] Build a Vector Database on Amazon S3 Vectors and Cut SaaS Costs - Janea Systems
https://www.janeasystems.com/blog/how-to-build-cost-efficient-vector-db-s3

[5] [6] CaGR-RAG: Context-aware Query Grouping for Disk-based Vector Search in RAG Systems
https://arxiv.org/html/2505.01164v1

[7] Valkey · Introducing Vector Search To Valkey
https://valkey.io/blog/introducing-valkey-search/

[8] [14] Top 10 open source vector databases
https://www.instaclustr.com/education/vector-database/top-10-open-source-vector-databases/