



Generating and Storing Vectors

1. Overview of AI Workflows	2. Why Postgres as a vector store	3. Storing and managing vectors	4. Querying the vector store
<ul style="list-style-type: none"> Look at high-level architecture - LLMs, vector stores and JSON Look at key vocabulary and concepts (embeddings, vectors, hybrid queries, etc.) 	<ul style="list-style-type: none"> What is a vector store? Key concepts and use cases. Why Postgres and how does it compare with other market tools Setting up Postgres with vector capabilities (pgvector) Lab: Install and configure Postgres using Docker 	<ul style="list-style-type: none"> Generating embeddings: Overview of tools and workflows Storing and organizing embeddings in Postgres Strategies for handling large datasets including chunking Dense and sparse vectors Lab: Generate embeddings for a dataset and store them 	<ul style="list-style-type: none"> Techniques for similarity search: k-NN, cosine similarity Using indexes to optimize vector queries Reranking results Lab: Query stored vectors to retrieve similar items (document/image search)
5. Querying LLMs with retrieved data	6. NoSQL with JSON in Postgres	7. Integrating Vector, Relational and JSON Data	8. Putting it all together
<ul style="list-style-type: none"> Recap on querying LLMs vis APIs Best practices for combining vector retrieval with LLM prompts Prompt configuration parameters (temperature, top-k, etc) Lab: Build a pipeline where vector store results enhance LLM responses (context-aware Q&A, etc) 	<ul style="list-style-type: none"> Overview of JSON/JSONB support in Postgres Querying JSONB data with SQL Indexing JSONB data for performance Lab: Design a schema mixing vector, relational and JSONB data for a sample project 	<ul style="list-style-type: none"> Building hybrid queries to power advanced workflows Case study: Combining embeddings, metadata (relational) and configurations (JSON) Lab: Implement a hybrid query to support a sample AI use case 	<ul style="list-style-type: none"> Full stack pipeline demo: Retrieve data, query the LLM and return results Debugging and optimising the workflow Spotlight on LLM frameworks Lab: Build a working application combining all elements



Plan for the session

- Why storing vectors matters
- Generating embeddings (curl/API client and Python)
- Storing embeddings (manually and Python)
- Lab: Generate and store embeddings
- Strategies for handling large datasets
- Dense vs. sparse vectors
- Lab

Why storing vectors matters



- **Avoid Recomputing Work** – Storing vectors prevents the need to reprocess raw data every time a similarity search is performed, saving computational resources.
- **Enable Semantic Search** – Vectors capture meaning, allowing searches to be based on concepts rather than exact keywords.
- **Facilitate Fast Retrieval** – Optimized vector indexes (like FAISS or pgvector) enable efficient similarity searches across large datasets.
- **Support AI and Recommendation Systems** – Vectors help power recommendation engines, content discovery, and personalized user experiences.
- **Improve Multimodal Data Processing** – Images, text, and audio can all be converted into embeddings for unified retrieval.
- **Enable Scalable AI Applications** – Storing vectors in a dedicated database allows efficient retrieval even as data grows.
- **Enhance Ranking and Clustering** – Vectors allow for advanced ranking, grouping similar items together, and detecting anomalies.
- **Power LLM Context Expansion** – Vector storage enables long-term memory for LLM applications by retrieving relevant context efficiently.

Walkthrough: Generating and storing embeddings

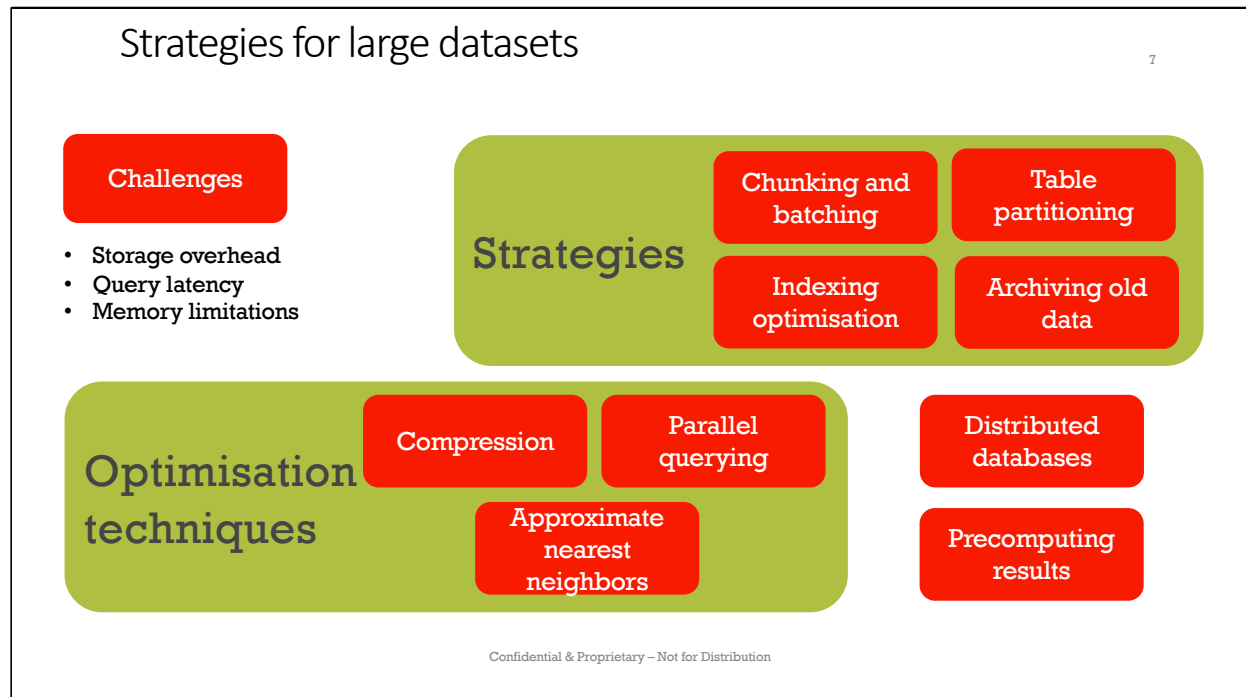


Lab: From Single Book Entry to Dynamic Data Integration



Strategies for large datasets

7



Strategies for Handling Large Datasets

When managing large datasets in PostgreSQL with vector embeddings, the primary goal is to ensure scalability, performance, and efficient resource utilization. Here are some strategies to address challenges like query latency, memory usage, and storage:

1. Chunking and Batch Processing

•Why?

- Loading or processing a large dataset at once can overwhelm the database or embedding generation service.

•How?

- Split the dataset into smaller, manageable chunks and process each chunk separately.

•**Implementation Example:** `chunk_size = 100` for `i` in `range(0, len(data), chunk_size)`: `chunk = data[i:i+chunk_size]` `process_chunk(chunk)`

2. Partitioning the Table

•Why?

- As datasets grow, querying and maintaining indexes on a single table becomes slower.

•How?

- Partition the table based on a logical key (e.g., id, timestamp, or a categorical value like category).

•**Example SQL for Range Partitioning:** CREATE TABLE items_partitioned (LIKE items INCLUDING ALL) PARTITION BY RANGE (id); CREATE TABLE items_p1 PARTITION OF items_partitioned FOR VALUES FROM (1) TO (1000000); CREATE TABLE items_p2 PARTITION OF items_partitioned FOR VALUES FROM (1000001) TO (2000000);

3. Indexing Optimization

•Why?

- The choice of indexing method directly impacts query performance for large datasets.

•Strategies:

- Tune the parameters for HNSW or IVFFlat indexes based on dataset size and query patterns.
- Consider separate indexes for different similarity metrics if multiple query types are needed.

4. Parallel Querying

•Why?

- Large datasets benefit from using multiple CPU cores for faster query execution.

•How?

- Enable parallel query execution in PostgreSQL.
- Example: Adjust these PostgreSQL configuration parameters: SET max_parallel_workers = 4; SET parallel_setup_cost = 1000;

5. Using Approximate Nearest Neighbors (ANN)

•Why?

- Exact similarity searches can become prohibitively slow for very large datasets.

•How?

- Use indexing methods like HNSW or IVFFlat to perform approximate nearest neighbor searches. These trade a small amount of accuracy for significant speed improvements.

6. Archiving and Pruning Old Data

•Why?

- Retaining all historical data in the same table may degrade performance unnecessarily.

•How?

- Archive older or less frequently accessed data into separate tables or databases.
- Use pg_dump or other tools to move old data to cold storage.

7. Monitoring and Resource Optimization

•Why?

- Proactively monitor database performance to avoid bottlenecks as the dataset grows.

•**How?**

- Use PostgreSQL's `pg_stat_activity` or `pg_stat_user_indexes` to track query performance.
- Identify and optimize slow queries using `EXPLAIN ANALYZE`.

8. Compression for Storage Efficiency

•**Why?**

- Large datasets consume significant disk space, which can affect performance and costs.

•**How?**

- Enable compression for tables with embeddings using extensions like `pg_compress`.

•**Example:** `CREATE TABLE items_compressed (id SERIAL PRIMARY KEY, name TEXT, item_data JSONB, embedding vector(1024) COMPRESS);`

9. Distributed Databases for Scalability

•**Why?**

- A single database instance may not scale beyond a certain point.

•**How?**

- Use tools like **Citus** (distributed PostgreSQL) to shard the dataset across multiple nodes.
- Query processing is distributed, reducing load on any single node.

10. Precomputing Results

•**Why?**

- Frequently queried data can be precomputed and stored to reduce query overhead.

•**How?**

- Maintain a materialized view of precomputed similarity results.
- Refresh the view periodically: `CREATE MATERIALIZED VIEW similar_items AS SELECT id, name, embedding <-> '[0.1, 0.2, 0.3]' AS distance FROM items WHERE distance < 0.5; REFRESH MATERIALIZED VIEW similar_items;`

Demo: Batch insertion





Sparse vs. Dense Vectors

9



Feature	Sparse Vectors	Dense Vectors
Definition		
Dimensionality		
Zero Values		
Generation Method		
Interpretability		
Query Matching		
Performance		
Use Case		

Use Case Scenarios (Icons + Short Descriptions):

•Sparse Vectors:




-  **Keyword-Based Search** – Find documents with exact word matches
-  **FAQs & Legal Documents** – Explicit term-based retrieval

•Dense Vectors:

-  **AI-Powered Semantic Search** – Matches related concepts, even with different words
-  **Recommendation Systems** – Context-aware content retrieval

Hybrid Approach Section (Smaller Text, Bottom of Slide):

•Combining Sparse & Dense for Best Results:

-  **Sparse Vectors:** Filter results by exact terms
-  **Dense Vectors:** Rank results by semantic relevance
-  **Example:** Search query "Python APIs for databases" → Sparse filtering (TF-IDF) → Dense ranking (BERT embeddings)

