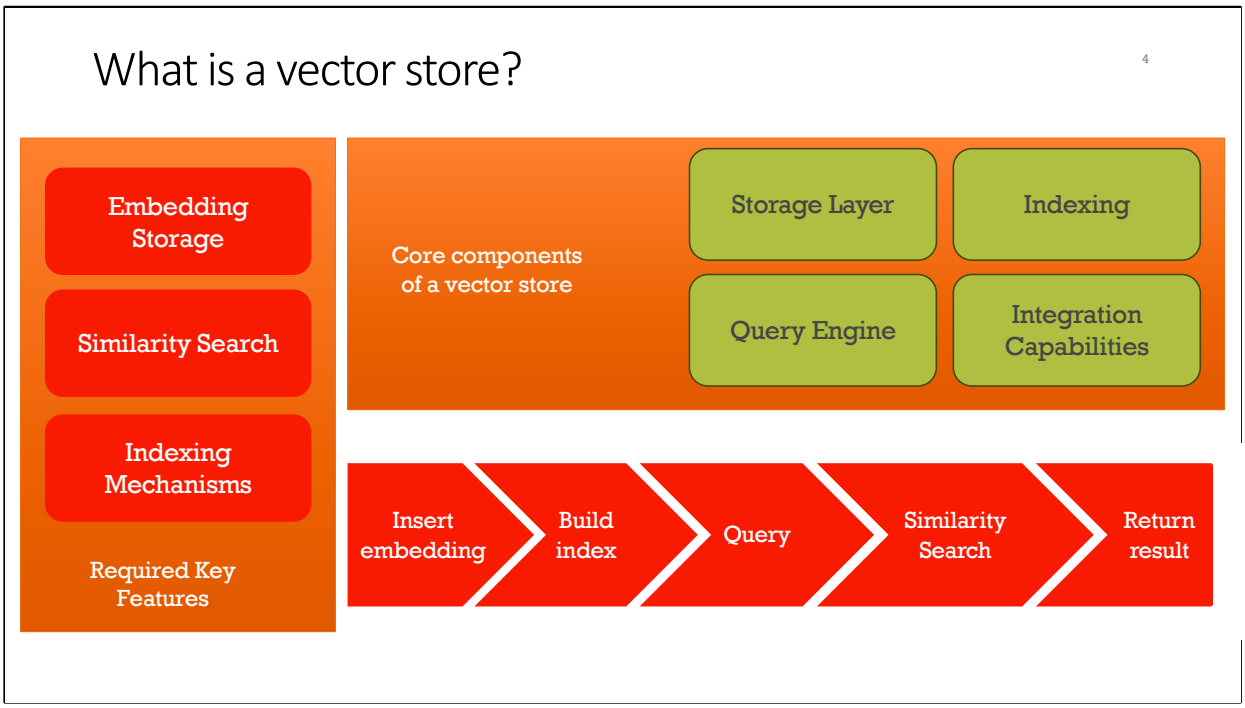

PostgreSQL as a vector store

1. Overview of AI Workflows	2. Why Postgres as a vector store	3. Storing and managing vectors	4. Querying the vector store
• Look at high-level architecture - LLMs, vector stores and JSON • Look at key vocabulary and concepts (embeddings, vectors, hybrid queries, etc.)	• 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	• 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	• 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**
• 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)	• 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	• 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	• 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

- What is a Vector Store?
- Why traditional databases fall short for vectors
- Key benefits of using Postgres
- Postgres vs other solutions
- Postgres for Vector operations
- Getting set up



1. Embedding Storage (Reveal 1):
"A vector store begins with the ability to store embeddings. These are high-dimensional numerical representations of data, like text or images, that capture their semantic meaning. For example, the phrase 'climate change' and 'global warming' might have similar embeddings because they mean similar things. The storage layer is where these embeddings are organized and kept ready for fast access."

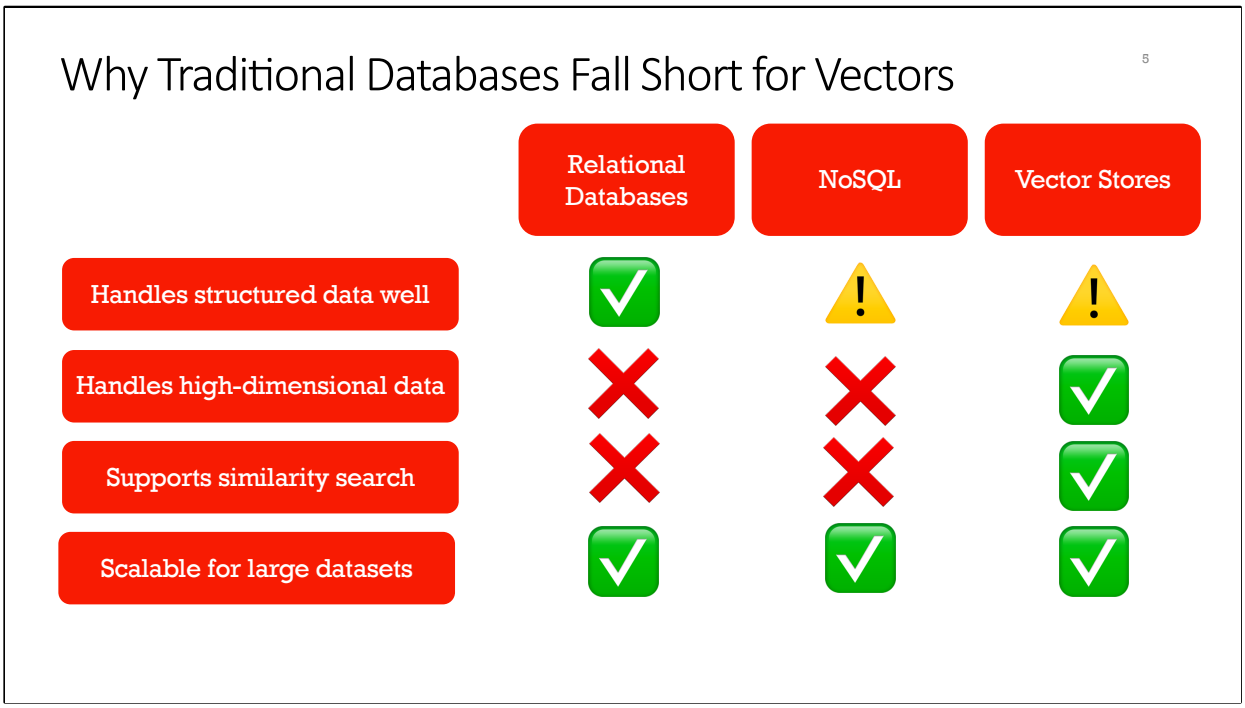
2. Similarity Search (Reveal 2):
"One of the core functions of a vector store is similarity search. This is what enables tasks like finding documents, images, or products that are most similar to a query. For example, searching for 'best programming books' retrieves results like 'Python tutorials' or 'Learn JavaScript,' because their embeddings are close to each other in the vector space."

3. Indexing Mechanisms (Reveal 3):
"To make similarity search fast and scalable, vector stores rely on indexing mechanisms. These are data structures and algorithms—like HNSW (Hierarchical Navigable Small World graphs)—that optimize the process of finding nearest neighbors in high-dimensional space. Without these, searching through millions of vectors would be computationally expensive."

4. Required Key Features (Reveal 4):
"Together, embedding storage, similarity search, and indexing form the core functionality of a vector store. These features allow us to efficiently store and query embeddings, enabling applications like semantic search, recommendation systems, and more."
5. Core Components of a Vector Store (Reveal 5):
"Now, let's break down the core components that make up a vector store into four main categories: storage layer, indexing, query engine, and integration capabilities. These components work together to deliver the functionality we just discussed."
6. Storage Layer (Reveal 6):
"The storage layer handles the organization and persistence of embeddings. This ensures that we can efficiently store high-dimensional data while scaling up to handle millions or even billions of vectors."
7. Indexing (Reveal 7):
"Indexing is key to fast search and retrieval. As mentioned earlier, it uses specialized algorithms like HNSW to structure the vectors in a way that reduces the search space for similarity queries."
8. Query Engine (Reveal 8):
"The query engine is where the actual similarity search happens. It compares the query vector to stored vectors using metrics like cosine similarity or Euclidean distance. The query engine is also where filtering and ranking take place to refine the results."
9. Integration Capabilities (Reveal 9):
"Finally, integration capabilities are what allow vector stores to work seamlessly with other systems. For example, integrating with APIs, LLMs, or traditional databases. This flexibility makes it possible to embed vector search into existing AI workflows."
10. Workflow Overview (Reveal 10):
"Now let's look at how these components interact in a typical workflow. This step-by-step process shows how data flows through a vector store."
11. Insert Embedding (Reveal 11):
"The first step is inserting an embedding into the vector store. This happens after a model like BERT or GPT converts raw data into a vector."
12. Build Index (Reveal 12):
"Once an embedding is added, the index is updated to reflect its position in the vector space. This step ensures that the system can find it efficiently during queries."
13. Query (Reveal 13):
"When a query is submitted, it is converted into an embedding, which is then matched against the stored embeddings using the index. This step narrows down potential matches."

14. Similarity Search (Reveal 14):
"Next, the system performs similarity search to find the vectors closest to the query embedding based on cosine similarity, Euclidean distance, or other metrics."
15. Return Result (Reveal 15):
"Finally, the results are ranked and returned to the user. These results could be documents, images, or any other content that matches the query semantically."

"This entire workflow—powered by embedding storage, indexing, and efficient querying—forms the backbone of modern AI applications like semantic search, recommendation systems, and Q&A. In the next slide, we'll look at why traditional databases struggle with these kinds of tasks and why vector stores are essential."



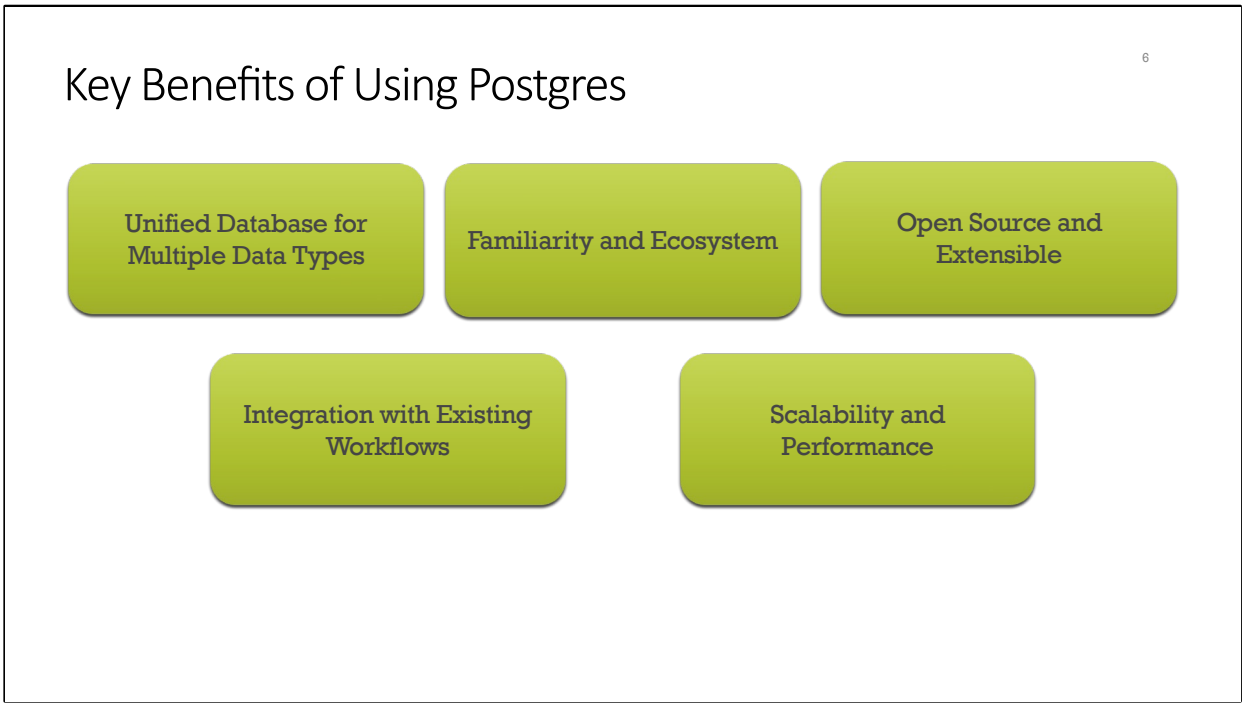
tured data, like rows and columns.
ct matches and basic range queries.

ors:
h-dimensional data effectively.
pport for vector similarity measures like
r L2 norm.
ng for 'most similar document
ires a brute-force approach, which is
xpensive.

Strengths:
- Great for unstructured or semi-structured data (e.g., JSON documents).
- Scalable and fast for key-value lookups.

Limitations for Vectors:
- No optimized indexing for high-dimensional data.
- Limited or no support for similarity queries, making nearest neighbor search infeasible for large datasets.

What Vector Stores Do Di
- Use advanced data stru Navigable Small World) efficiently.
- Support for cosine simi distance out of the box.
- Designed to handle bill increase in search time.



Postgres is more than just a relational database—it's a versatile platform that supports multiple data types and modern use cases. Let's look at why it's a great choice for vector storage and AI workflows.

1. Unified Database for Multiple Data Types:

•Postgres supports relational data, JSON, and vector embeddings in one system.

•Enables hybrid queries that combine structured filters (e.g., price, location) with semantic similarity search.

Example:

•"Retrieve hotels in London with a pool (structured query) that are similar to 'luxury resorts' (vector query)."

2. Familiarity and Ecosystem:

•Widely used by developers, making it easier to adopt without significant learning curves.

•Compatible with many tools and frameworks in the Postgres ecosystem.

Example:

•"If your team already uses Postgres, adding vector capabilities through pgvector is a seamless transition."

3. Open Source and Extensible:

•No vendor lock-in, with community-driven innovation.

•Extensions like pgvector provide robust vector capabilities.

•Works well with Docker for quick deployments.

4. Integration with Existing Workflows:

•Combines well with relational operations, making it ideal for applications requiring metadata alongside embeddings.

Example:

•"For a product recommendation engine, use embeddings for similarity search while joining relational tables for inventory or pricing."

5. Scalability and Performance:

•Scales effectively for medium to large datasets.

•Supports parallel queries and indexing techniques like HNSW for performance optimization.

These benefits make Postgres a strong contender for organizations looking to integrate vector search into their workflows without adopting entirely new tools or systems. Next, we'll compare Postgres to other vector solutions.

Postgres vs other solutions

7

	Postgres w/pgvector	Weaviate	Pinecone	Elasticsearch
Ease of use	✅ Familiar for devs	⚠️ Medium learning curve	✅ Fully managed	⚠️ Steep learning curve
Scalability	✅ Medium to large data	✅ High	✅ Very high	✅ High
Integration	✅ Unified	✅ Rich schema support	❌ Limited	⚠️ Basic JSON
Performance	⚠️ Good w/ indexing	✅ Optimized	✅ Optimized	⚠️ Mixed performance
Deployment options	✅ On-prem/Docker	✅ Cloud/on-prem	✅ Cloud-only	✅ Cloud/on-prem
Cost	✅ Free/Open-source	✅ Free/Open-source core	❌ Pay-per-query	⚠️ Varies

Postgres for Vector operations - pgvector

8

- **Vector Data Type:**
  - Store embeddings directly as a vector data type (e.g., VECTOR(1536)).
- **Similarity Metrics:**
  - Supports distance functions like cosine similarity, Euclidean distance, and inner product.
- **Efficient Indexing:**
  - HNSW (Hierarchical Navigable Small World graphs) for fast nearest neighbour search.
  - IVFFlat (InVerted File Flat) is a simple vector index that splits data into buckets.
  - Reduces the computational cost of large-scale similarity queries.

Performance Optimizations:

•Parallel Queries:

- Postgres can process multiple queries simultaneously to improve throughput.

•Index Configuration:

- Proper indexing ensures that even large datasets (millions of vectors) are queried efficiently.

•Partitioning:

- Partition tables to improve performance for massive datasets.

4. Real-World Example:

Scenario:

"An e-commerce platform uses Postgres to store product embeddings and retrieve similar products:

•Query: Find products similar to 'noise-canceling headphones' (vector similarity).

•Filter: Only include products under $200 (relational query).

Getting started

- Let's check our database is up and running and check we can send a request to our model


Questions
?