The GenAI Promise and Its Limits

What Generative AI Does Well

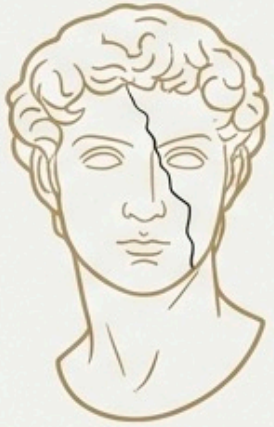
LLMs excel at tasks that rely on pattern recognition and language fluency:

- **Text generation**: Creating human-like responses, summaries, explanations
- **Language understanding**: Parsing intent, extracting meaning, following instructions
- **Pattern completion**: Continuing sequences, filling in blanks, generating variations
- **Translation and transformation**: Converting between formats, styles, languages

These capabilities emerge from training on vast amounts of text data.

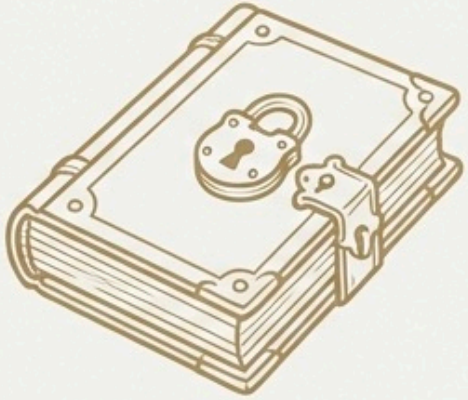
The Three Gaps in the GenAI Promise

LLMs excel at text generation, language understanding, and pattern completion. However, production systems must address three fundamental limitations.



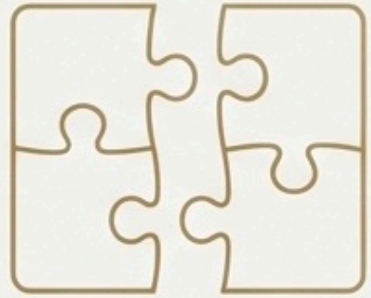
Hallucination (Confident but Wrong)

LLMs generate statistically *probable* text, not factually *verified* text. When they don't know, they fabricate with full confidence.



Knowledge Cutoff (Your Data is Invisible)

Models are trained on public data with a cutoff date. They have no access to your private company documents, databases, or real-time events.



Relationship Blindness (Can’t Connect the Dots)

LLMs process text sequentially and struggle to answer questions that require reasoning over relationships across multiple documents or sources.

1. Hallucination: Confident But Wrong

LLMs generate responses based on statistical likelihood, not factual verification.

The Problem:

- Produces the most *probable* continuation, not the most *accurate*
- Doesn't say "I don't know"—generates plausible-sounding text instead
- Complete with fabricated details and citations

Real Example: In 2023, US lawyers were sanctioned for submitting an LLM-generated brief with six fictitious case citations.

2. Knowledge Cutoff: No Access to Your Data

LLMs are trained at a specific point in time on publicly available data.

They don't know:

- Recent events after their training cutoff
- Your company's documents, databases, or internal knowledge
- Real-time data: current prices, live statistics, changing conditions

The Risk: Ask about your Q3 results or last week's board meeting, and the LLM may still generate a confident (and wrong) response.

3. Relationship Blindness: Can't Connect the Dots

LLMs process text sequentially and treat each piece in isolation.

Questions they struggle with:

- "Which asset managers own companies facing cybersecurity risks?"
- "What products are mentioned by companies that share risk factors?"
- "How are these two companies connected through their executives?"

These questions require *reasoning over relationships*—connecting entities across documents and traversing chains of connections.

Why These Limitations Matter

Limitation	Impact	Example
Hallucination	Can't trust answers without verification	Legal brief with fabricated citations
Knowledge cutoff	Can't answer questions about your data	"What did our CEO say last quarter?"
Relationship blindness	Can't reason across connected information	"Which investors are exposed to these risks?"

Building production AI systems means addressing these limitations directly.

The Solution: Providing Context

All three limitations have a common solution—**providing context**.

When you give an LLM relevant information in its prompt:

- It has facts to work with (reduces hallucination)
- It can access your specific data (overcomes knowledge cutoff)
- You can structure that information to show relationships (enables reasoning)

This is the foundation of **Retrieval-Augmented Generation (RAG)**.

Summary

In this lesson, you learned about the fundamental limitations of LLMs:

- **Hallucination**: LLMs generate probable responses, not verified facts
- **Knowledge cutoff**: LLMs can't access recent events or your private data
- **Relationship blindness**: LLMs struggle with cross-document reasoning

The solution is providing context—which leads us to RAG.

Next: Learn how traditional RAG works and why it has its own limitations.

Context and the Limits of Traditional RAG

The Power of Context

Providing context in prompts dramatically improves LLM responses.

When you include relevant information, the model can:

- Generate accurate summaries grounded in actual documents
- Answer questions about your specific data
- Reduce hallucination by having facts to reference

This insight leads to **Retrieval-Augmented Generation (RAG)**.

How Traditional RAG Works

Traditional RAG follows a simple pattern:

1. **Index documents**: Break documents into chunks and create embeddings
2. **Receive query**: User asks a question
3. **Retrieve context**: Find chunks with embeddings similar to the query
4. **Generate response**: Pass retrieved chunks to LLM as context

This works well for many use cases. But it has fundamental limitations.

The Problem with Traditional RAG

Traditional RAG treats documents as isolated, unstructured blobs.

What traditional RAG sees:

```
Chunk 1: "Apple Inc. faces cybersecurity risks including..."
Chunk 2: "BlackRock Inc. holds shares in technology companies..."
Chunk 3: "The semiconductor supply chain impacts..."
```

What traditional RAG misses:

- Which specific companies does BlackRock own?
- Do any of those companies face cybersecurity risks?
- How are supply chain issues connected to specific products?

Retrieves Similar Content, Not Connected Information

Traditional RAG can find text about cybersecurity and text about BlackRock.

But it can't tell you:

- Which asset managers are exposed to cybersecurity risks through their holdings

Why? Each chunk is independent—there's no understanding of how information connects.

Questions Traditional RAG Can't Answer

Question	Why Traditional RAG Struggles
"Which asset managers own companies facing cyber risks?"	Requires connecting ownership data to risk mentions
"What products are mentioned by companies that share risk factors?"	Requires finding shared entities across documents
"How many companies mention supply chain issues?"	Requires aggregation, not similarity search
"What executives work for companies in the tech sector?"	Requires traversing entity relationships

These questions need *structured context* that preserves relationships.

From Unstructured to Structured

The core insight: Information isn't truly unstructured.

Documents contain:

- **Entities**: Companies, people, products, risks
- **Relationships**: Owns, faces, mentions, works for

Traditional RAG ignores this structure. It treats a document as a bag of words to embed and search.

The GraphRAG Solution

GraphRAG extracts structure, creating a *knowledge graph* that preserves:

- **Entities**: The things mentioned in documents
- **Relationships**: How those things connect
- **Properties**: Attributes and details about entities

Traditional RAG asks: "What chunks are similar to this query?"

GraphRAG asks: "What entities and relationships are relevant to this query?"

Three Retrieval Patterns

GraphRAG enables three retrieval patterns:

Pattern	What It Does
Vector search	Find semantically similar content (what traditional RAG does)
Graph traversal	Follow relationships from relevant content to connected entities
Database queries	Answer precise questions about entities and relationships

The combination is more powerful than any single approach.

Summary

In this lesson, you learned:

- **Context improves LLM responses** by providing relevant information
- **Traditional RAG** retrieves similar chunks based on vector similarity
- **The limitation**: Traditional RAG treats documents as isolated blobs, missing relationships
- **Questions requiring relationships** can't be answered with similarity search alone
- **GraphRAG** extracts structure from documents, preserving entities and relationships

Next: Learn how to transform documents into knowledge graphs.

From Documents to Knowledge Graphs

with the Neo4j GraphRAG Package

The neo4j-graphrag Python Package

The official Neo4j GenAI package for Python provides a first-party library to integrate Neo4j with generative AI applications.

Key benefits:

- Long-term support and fast feature deployment
- Reduces hallucinations through domain-specific context
- Combines knowledge graphs with LLMs for GraphRAG

Supported Providers

LLMs:

- OpenAI, Anthropic, Cohere, Google, MistralAI, Ollama

Embeddings:

- OpenAI, sentence-transformers, provider-specific models

This flexibility lets you choose the models that best fit your requirements and budget.

Building and Querying

The package provides tools for both constructing and querying knowledge graphs:

Category	Components
Construction	`SimpleKGPipeline`, `Pipeline` class
Retrieval	`VectorRetriever`, `Text2CypherRetriever`, hybrid methods
Orchestration	`GraphRAG` class for retrieval + generation

SimpleKGPipeline

The key component for graph construction:

What it does:

1. Extracts text from documents (PDFs, text files)
2. Breaks text into manageable chunks
3. Uses an LLM to identify entities and relationships
4. Stores the structured data in Neo4j
5. Creates vector embeddings for semantic search

The Transformation Process

Step	What Happens
Document Ingestion	Read source documents (PDFs)
Chunking	Break into smaller pieces for processing
Entity Extraction	LLM identifies companies, products, risks, metrics
Relationship Extraction	LLM finds connections between entities
Graph Storage	Save entities and relationships to Neo4j
Vector Embeddings	Generate embeddings for semantic search

The SEC Filings Example

Throughout this workshop, you'll work with a knowledge graph built from SEC 10-K filings.

These documents contain:

- Companies and their business descriptions
- Risk factors they face
- Financial metrics they report
- Products and services they mention
- Executives who lead them

From PDF to Graph

In raw PDF form: Information is locked in narrative text.

In a knowledge graph: It becomes structured and queryable:

```
(Apple Inc)-[:FACES_RISK]->(Cybersecurity Threats)
(Apple Inc)-[:MENTIONS]->(iPhone)
(BlackRock Inc)-[:OWNS]->(Apple Inc)
```

The Complete Picture

After processing, your knowledge graph contains:

```
Documents → Chunks (with embeddings)
    ↓
Entities (Company, Product, RiskFactor, Executive, FinancialMetric)
    ↓
Relationships (FACES_RISK, MENTIONS, OWNS, WORKS_FOR, HAS_METRIC)
```

This structure enables questions that traditional RAG can't answer.

What the Graph Enables

Question Type	How the Graph Helps
"What risks does Apple face?"	Traverse FACES_RISK relationships
"Which companies mention AI?"	Find MENTIONS relationships to AI products
"Who owns Apple?"	Follow OWNS relationships from asset managers
"How many risk factors are there?"	Count RiskFactor nodes

Quality Depends on Decisions

The quality of your knowledge graph depends on several key decisions:

- **Schema design**: What entities and relationships should you extract?
- **Chunking strategy**: How large should chunks be?
- **Entity resolution**: How do you handle the same entity mentioned differently?
- **Prompt engineering**: How do you guide the LLM to extract accurately?

The following lessons cover each of these decisions.

Summary

In this lesson, you learned:

- **neo4j-graphrag** is the official package for building GraphRAG applications
- **SimpleKGPipeline** orchestrates the transformation from documents to graphs
- **The process**: Document → Chunks → Entity Extraction → Relationship Extraction → Graph Storage → Embeddings
- **Graph structure** enables queries that traverse relationships, not just find similar text

Next: Learn about schema design in SimpleKGPipeline.

Schema Design in SimpleKGPipeline

Why Schema Matters

Without a schema, extraction is unconstrained—the LLM extracts *everything*.

This creates graphs that are:

- **Non-specific**: Too many entity types with inconsistent labeling
- **Hard to query**: No predictable structure to write queries against
- **Noisy**: Irrelevant entities mixed with important ones

Providing a schema tells the LLM exactly what to look for.

Schema in SimpleKGPipeline

SimpleKGPipeline accepts a `schema` parameter that guides extraction:

- **Node types**: What kinds of entities should be extracted
- **Relationship types**: What connections between entities matter
- **Patterns**: Which node-relationship-node combinations are valid

The pipeline uses the schema to guide extraction, prune invalid data, and ensure consistency.

Three Schema Modes

Mode	Description	Best For
User-Provided	You define exactly what to extract	Production systems
Extracted	LLM discovers schema from documents	Exploration
Free	No constraints, extract everything	Initial discovery

User-Provided Schema

```
schema = {
    "node_types": [
        {"label": "Company", "description": "A business organization"},
        {"label": "RiskFactor", "description": "A business risk or threat"},
        {"label": "Product", "description": "A product or service"},
    ],
    "relationship_types": [
        {"label": "FACES_RISK", "description": "Company faces this risk"},
        {"label": "MENTIONS", "description": "Company mentions this product"},
    ],
    "patterns": [
        ("Company", "FACES_RISK", "RiskFactor"),
        ("Company", "MENTIONS", "Product"),
    ]
}
```

Automatic Schema Extraction

Let the LLM discover the schema from your documents:

```
pipeline = SimpleKGPipeline(
    driver=driver,
    llm=llm,
    embedder=embedder,
    schema="EXTRACTED",  # or simply omit schema
)
```

Useful when you don't know what entities exist in your documents.

Free Mode

Extract everything without constraints:

```
pipeline = SimpleKGPipeline(
    driver=driver,
    llm=llm,
    embedder=embedder,
    schema="FREE",
)
```

Most comprehensive extraction, but inconsistent structure.

Defining Node Types

Node types can be simple strings or detailed dictionaries:

Simple:

```
node_types = ["Company", "Product", "RiskFactor"]
```

With descriptions and properties:

```
node_types = [
    {"label": "Company", "description": "A business organization"},
    {
        "label": "Product",
        "properties": [{"name": "category", "type": "STRING"}]
    }
]
```

Descriptions help the LLM understand what each type means.

Patterns: Valid Connections

Patterns specify which relationships are valid between node types:

```
patterns = [
    ("Company", "FACES_RISK", "RiskFactor"),
    ("Company", "MENTIONS", "Product"),
    ("Executive", "WORKS_FOR", "Company"),
]
```

Without patterns, the LLM might create nonsensical relationships like:

`(Product)-[:FACES_RISK]->(Company)`

Schema for This Workshop

Node Type	Description
Company	Organizations filing reports
RiskFactor	Business risks identified
Product	Products and services mentioned
Executive	Company leaders
FinancialMetric	Financial measures reported

Relationships for This Workshop

Relationship	Pattern
FACES_RISK	Company → RiskFactor
MENTIONS	Company → Product
WORKS_FOR	Executive → Company
HAS_METRIC	Company → FinancialMetric

This focuses extraction on business-relevant information.

When to Use Each Mode

Mode	Best For
User-Provided	Production systems with known query patterns
Extracted	Exploration when you're learning the domain
Free	Initial discovery of what's in your documents

For most production GraphRAG applications, a user-provided schema produces the most reliable results.

Learn More

This lesson covers how to use schemas in SimpleKGPipeline.

For deep dives into schema design principles—including iterative refinement, domain modeling, and advanced patterns—see the dedicated **Neo4j GraphRAG Python course** on GraphAcademy.

Summary

In this lesson, you learned:

- **Schema guides extraction**: Tells SimpleKGPipeline what to find
- **Three modes**: User-provided (control), Extracted (discovery), Free (exploration)
- **Node and relationship types**: Define what to extract with optional descriptions
- **Patterns**: Specify valid connections between node types
- **Production systems**: Benefit most from user-provided schemas

Next: Learn about chunking strategies and their trade-offs.

Chunking Strategies

Why Chunking Matters

LLMs have context limits. You can't pass an entire 200-page SEC filing to an LLM for entity extraction.

Documents must be broken into smaller pieces—**chunks**—that fit within processing limits.

But chunking isn't just a technical necessity. How you chunk documents affects both:

- **Extraction quality**
- **Retrieval quality**

The Dual Impact of Chunk Size

Chunk size creates a fundamental trade-off:

For Entity Extraction:

- Larger chunks provide more context for understanding entities
- The LLM sees more surrounding text, making better extraction decisions
- "The Company" can be resolved to "Apple Inc" when full context is visible

For Retrieval:

- Smaller chunks enable more precise matches
- When searching, you want the most relevant *portion*, not a huge blob
- Less irrelevant content mixed with relevant content

The Trade-off Illustrated

Large Chunks (2000 chars)

```
[Full paragraph about Apple's
risk factors, mentioning
cybersecurity, supply chain,
and regulatory risks with
full context about each]
```

- ✅ Better entity extraction
- ❌ Less precise retrieval
- ❌ Returns more than needed

Small Chunks (200 chars)

```
[Apple faces cybersecurity...]
[Supply chain disruptions...]
[Regulatory changes in...]
```

Chunk Size Parameters

The `FixedSizeSplitter` has two key parameters:

`chunk_size`: Maximum number of characters per chunk

`chunk_overlap`: Characters shared between consecutive chunks

```
from neo4j_graphrag.experimental.components.text_splitters.fixed_size_splitter import FixedSizeSplitter

splitter = FixedSizeSplitter(chunk_size=500, chunk_overlap=50)
```

Overlap ensures context isn't lost at chunk boundaries.

Configuring the Pipeline

Pass a custom text splitter to SimpleKGPipeline:

```
from neo4j_graphrag.experimental.components.text_splitters.fixed_size_splitter import FixedSizeSplitter

# Create a splitter with your preferred settings
splitter = FixedSizeSplitter(chunk_size=500, chunk_overlap=50)

# Pass it to the pipeline
pipeline = SimpleKGPipeline(
    driver=driver,
    llm=llm,
    embedder=embedder,
    entities=entities,
    relations=relations,
    text_splitter=splitter,  # Custom splitter
)
```

Typical Chunk Sizes

Chunk Size	Best For
200-500 chars	High-precision retrieval, FAQ-style content
500-1000 chars	Balanced extraction and retrieval
1000-2000 chars	Context-heavy extraction, narrative documents
2000+ chars	Maximum context, fewer chunks

For SEC filings with complex, interconnected information, **500-1000 characters** often provides a good balance.

Evaluating Chunk Quality

After chunking, evaluate the results:

```
// Check chunk count per document
MATCH (d:Document)<-[:FROM_DOCUMENT]-(c:Chunk)
RETURN d.path, count(c) AS chunkCount
ORDER BY chunkCount DESC

// Check chunk size distribution
MATCH (c:Chunk)
RETURN
    min(size(c.text)) AS minSize,
    max(size(c.text)) AS maxSize,
    avg(size(c.text)) AS avgSize
```

What to Look For

Good chunks:

- Reasonable count per document
- Consistent sizes
- Contain coherent, complete thoughts

Signs of problems:

- Too few chunks → may be too large
- Highly variable sizes → inconsistent processing
- Incomplete sentences → overlap may be too small

Experiment and Iterate

The best chunk size depends on:

- Document type and structure
- Query patterns you expect
- Balance between extraction and retrieval needs

Start with a moderate size (500-800 characters), evaluate results, and adjust.

Summary

In this lesson, you learned:

- **Chunking breaks documents** into processable pieces
- **Chunk size affects both** entity extraction quality and retrieval precision
- **Larger chunks** = better context for extraction, less precise retrieval
- **Smaller chunks** = more precise retrieval, less context for extraction
- **Overlap** helps preserve context across chunk boundaries
- **Experimentation** is key to finding the right balance

Next: Learn about entity resolution—handling duplicate entities.

Entity Resolution

The Duplicate Entity Problem

When entities are extracted from text, the same real-world entity can appear with different names:

- "Neo4j" vs "Neo4j Graph Database" vs "Neo4j, Inc."
- "Apple" vs "Apple Inc" vs "Apple Inc." vs "the Company"
- "Tim Cook" vs "Timothy Cook" vs "CEO Tim Cook"

Without resolution: Your graph contains multiple nodes representing the same thing.

Why This Breaks Queries

```
// This might miss risks if Apple appears under different names
MATCH (c:Company {name: 'Apple Inc'})-[:FACES_RISK]->(r:RiskFactor)
RETURN r.name
```

If some risks are connected to "Apple" and others to "APPLE INC", your query returns incomplete results.

You can't trust basic queries like "How many risk factors does Apple face?"

Why Entity Resolution Matters

Entity resolution ensures:

- **Query accuracy**: One node per real-world entity
- **Relationship completeness**: All relationships connect to the canonical entity
- **Aggregation correctness**: Counts and summaries reflect reality

Default Resolution in SimpleKGPipeline

By default, `SimpleKGPipeline` performs basic resolution:

- Entities with the **same label** and **identical name** are merged
- "Company: Apple Inc" + "Company: Apple Inc" = one node

But it misses variations:

- "Apple Inc" and "APPLE INC" (case difference)
- "Apple Inc" and "Apple Inc." (punctuation)
- "Apple" and "Apple Inc" (name variation)

Resolution Trade-offs

Too Aggressive

- "Apple Inc" (tech) merged with "Apple Records" (music)
- Distinct entities incorrectly combined
- Relationships become meaningless

Too Conservative

- "Apple Inc" and "APPLE INC" remain separate
- Queries miss connections
- Aggregations are wrong

The right balance depends on your domain.

Resolution Strategies

Strategy 1: Upstream Normalization

Guide the LLM during extraction:

```
prompt_template = """
When extracting company names, normalize to official names:
- "Apple", "Apple Inc", "the Company" → "APPLE INC"
- Use uppercase for company names
- Use the full legal name when known
"""
```

Strategy 2: Reference Lists

Provide a canonical list of entities:

```
prompt_template = """
Only extract companies from this approved list:
- APPLE INC
- MICROSOFT CORP
- ALPHABET INC

Match variations to the canonical name.
"""
```

This works well when you know the entities in advance.

Strategy 3: Post-Processing Resolvers

Apply resolvers after extraction:

```
from neo4j_graphrag.experimental.components.entity_resolvers import FuzzyMatchResolver

resolver = FuzzyMatchResolver(
    driver=driver,
    similarity_threshold=0.85,  # How similar names must be to merge
)

# Run after pipeline completion
resolver.resolve()
```

Available Resolvers:

- **SpacySemanticMatchResolver**: Semantic similarity using spaCy
- **FuzzyMatchResolver**: String similarity using RapidFuzz

Disabling Resolution

You can disable entity resolution entirely:

```
pipeline = SimpleKGPipeline(
    driver=driver,
    llm=llm,
    embedder=embedder,
    entities=entities,
    relations=relations,
    perform_entity_resolution=False,  # No resolution
)
```

Useful for debugging or applying custom resolution logic later.

Validating Resolution

After resolution, verify your entity counts:

```
// Check for potential duplicates
MATCH (c:Company)
WITH c.name AS name, collect(c) AS nodes
WHERE size(nodes) > 1
RETURN name, size(nodes) AS duplicates

// Check company name variations
MATCH (c:Company)
WHERE c.name CONTAINS 'Apple' OR c.name CONTAINS 'APPLE'
RETURN c.name, count{(c)-[:FACES_RISK]->()} AS risks
```

Summary

In this lesson, you learned:

- **Entity resolution** merges duplicate nodes representing the same real-world entity
- **Default resolution** catches exact matches only
- **Post-processing resolvers** catch variations using semantic or fuzzy matching
- **The trade-off**: Too aggressive merges distinct entities; too conservative keeps duplicates
- **Strategies include**: Upstream normalization, reference lists, post-processing resolvers

Next: Learn about vectors and semantic search.

Vectors and Semantic Search

What is a Vector?

Vectors are lists of numbers.

The vector `[1, 2, 3]` represents a point in three-dimensional space.

In machine learning, vectors can represent much more complex data—including the *meaning* of text.

What are Embeddings?

Embeddings are numerical representations of text encoded as high-dimensional vectors (often 1,536 dimensions).

The key property: Similar meanings produce similar vectors.

- "Apple's business strategy" and "the company's strategic approach" → vectors close together
- "Apple's business strategy" and "banana nutrition facts" → vectors far apart

This enables **semantic search**—finding content by meaning, not just keywords.

Why Vectors Matter for GraphRAG

Your knowledge graph now has:

- Structured entities (companies, risks, products)
- Relationships (FACES_RISK, OWNS, MENTIONS)
- Text chunks from source documents

But how do you *find* relevant information when a user asks a question?

Without Vectors vs With Vectors

Without vectors:

- You need exact keyword matches
- "What challenges does Apple face?" won't find chunks about "risks" or "threats"

With vectors:

- The question and chunks become embeddings
- You find chunks with similar *meaning*, regardless of exact words
- "Challenges" finds content about "risks" and "threats"

Similarity Search

Vector similarity is typically measured by **cosine similarity**—the angle between two vectors:

Score	Meaning
Near 1.0	Very similar meanings
Near 0.5	Somewhat related
Near 0.0	Unrelated

When you search, your question becomes an embedding, and the system finds chunks with embeddings close to your question.

Storing Vectors in Neo4j

When SimpleKGPipeline processes documents:

1. Each chunk gets an embedding from the embedding model
2. The embedding is stored as a property on the Chunk node
3. A vector index enables fast similarity search across all chunks

```
// Chunks have embedding properties
MATCH (c:Chunk)
RETURN c.text, size(c.embedding) AS embeddingDimensions
LIMIT 1
```

Searching a Vector Index

```
// Create an embedding for the query
WITH genai.vector.encode(
    "What risks does Apple face?",
    "OpenAI",
    { token: $apiKey }
) AS queryEmbedding

// Search the vector index for similar chunks
CALL db.index.vector.queryNodes('chunkEmbeddings', 5, queryEmbedding)
YIELD node, score

RETURN node.text AS content, score
ORDER BY score DESC
```

This finds the 5 chunks most semantically similar to the query.

Combining Vectors with Graph Traversal

The real power of GraphRAG: Start with semantic search, then traverse the graph.

```
WITH genai.vector.encode(
    "What risks does Apple face?",
    "OpenAI",
    { token: $apiKey }
) AS queryEmbedding

CALL db.index.vector.queryNodes('chunkEmbeddings', 5, queryEmbedding)
YIELD node, score

// Traverse to connected entities
MATCH (node)<-[:FROM_CHUNK]-(entity)
RETURN node.text AS content, score, collect(entity.name) AS relatedEntities
```

Returns both similar text AND the entities extracted from that text.

The Complete Knowledge Graph

Your knowledge graph now has everything needed for GraphRAG:

Component	Purpose
Documents	Source provenance
Chunks	Searchable text units
Embeddings	Enable semantic search
Entities	Structured domain knowledge
Relationships	Connections between entities

Three Retrieval Patterns

This structure enables three retrieval patterns:

1. **Vector search**: Find semantically similar content
2. **Vector + Graph**: Find similar content, then traverse to related entities
3. **Text2Cypher**: Query the graph structure directly

You'll learn these patterns in detail in Lab 5.

Summary

In this lesson, you learned:

- **Vectors** are numerical representations of data
- **Embeddings** encode text meaning as high-dimensional vectors
- **Similar meanings** produce similar vectors, enabling semantic search
- **Neo4j stores vectors** alongside graph data with vector indexes
- **Semantic search** finds relevant chunks by meaning, not keywords
- **Vector + Graph** combines semantic search with relationship traversal

Your knowledge graph is complete. In Lab 5, you'll learn how to retrieve context using Vector, Vector Cypher, and Text2Cypher retrievers.