RAG: Retrieval-Augmented Generation

📌 Entity Search (2010) — In Short

Goal

Instead of returning **documents** when a user searches, return **entities** (people, places, organizations, products) that best answer the query.

Example:

- Query: *"Apple founder"*
- **Entity search result:** *Steve Jobs*
- Instead of showing articles, pages, or PDFs.

Key Idea

Represent each **entity** using:

1. **Entity description** (Wikipedia page, profile, etc.)
2. **Context documents** mentioning the entity (blogs, articles, news)
3. **Structured attributes** (metadata, database facts)

Then **rank** entities, not documents, based on how well they **match the query**.

Main Components

Component	Description
Entity Representation	Collect textual and structured info about the entity.
Evidence Gathering	Retrieve documents that mention or describe the entity.
Scoring / Ranking	Combine evidence to compute how relevant each entity is to the query.

Scoring Strategy

Two-part scoring:

$$\text{Score}(entity) = \text{Match}(query, \text{entity info}) + \text{Support}(\text{from context documents})$$

- If many *reliable* documents link the entity to the query → rank higher.

Impact

This paper **shifted the search from document retrieval to knowledge retrieval**, becoming the foundation for:

- **Google Knowledge Graph**
- **Bing Entity Pane**
- **Wikidata-powered QA**
- **All modern RAG and entity-based retrieval models**

✅ One-Sentence Summary

Entity Search (2010) introduced the idea of treating **entities** (not documents) as the primary retrieval targets and ranking them based on textual + contextual evidence, laying the foundation for modern knowledge graphs and semantic search.

✅ Entity Search (2010) — 10-Line Answer

Entity Search (2010) introduced the idea of **retrieving entities instead of documents** in response to a query.

The system represents each entity using its **textual description**, **structured attributes**, and **context documents** mentioning it.

Given a query, the system retrieves documents likely related to the entity and extracts evidence from them.

Entities are then **ranked based on how strongly these documents support the association** between the query and the entity.

The ranking score combines **query–entity similarity** and **query–context evidence strength**.

This approach shifts search from **document-level retrieval** to **semantic, entity-centric retrieval**.

It improves user experience by returning **direct answers**, not just web pages.

The method supports **knowledge graph construction** by linking entities to their descriptions.

It laid the foundation for modern systems like the **Google Knowledge Graph** and **Bing Entity Pane**.

Thus, Entity Search (2010) is a **key milestone** in the transition from keyword search to **semantic search**.

✅ REALM (2020) — In Short

REALM (Retrieval-Augmented Language Model) is a model that improves language understanding by **integrating retrieval into training and inference**.

Instead of relying only on knowledge stored in model parameters, REALM **dynamically retrieves relevant text from a large corpus** during question answering or comprehension tasks.

How it works

1. A query is encoded.
2. The model **retrieves relevant passages** using a **learned embedding-based retriever** (not BM25).
3. Retrieved passages are **fed back into the model** as additional context.
4. The model is trained **end-to-end** so retrieval and reasoning improve together.

Key Idea

> The model learns to retrieve what it needs to answer, instead of memorizing all facts in its weights.

Impact

- Reduces **hallucinations**, because answers are grounded in retrieved text.
- Enables **up-to-date knowledge** without retraining the model.

- Inspired later systems like **RAG (2021)**, **FiD (2021)**, and modern **Retrieval-Augmented LLMs**.

✅ One-Sentence Summary

REALM augments language models with a learned retriever that fetches supporting documents during inference, allowing models to answer questions using external knowledge instead of memorizing everything.

✅ REALM (2020) — 10-Line Answer

REALM (Retrieval-Augmented Language Model) introduces a framework that **combines neural retrieval with language model reasoning**.

Instead of storing all world knowledge inside model parameters, REALM **retrieves relevant text passages** from a large corpus during both training and inference.

The system uses a **learned dense retriever**, which maps queries and documents into the same embedding space.

When a query is given, the retriever selects the most relevant documents, which are then fed into the language model to support understanding and answer generation.

Retrieval is **differentiable and trained end-to-end**, so the model gradually learns **which information to look for**.

This reduces the model's dependence on memorization and improves performance on knowledge-intensive tasks.

REALM provides better factual correctness because it grounds responses in retrieved evidence.

It also allows the system to **update knowledge simply by changing the corpus**, without retraining the entire model.

REALM set the foundation for later Retrieval-Augmented Generation models such as **RAG, FiD, Atlas, and modern LLM tool-use**.

Thus, REALM marked a key shift from parameter-based knowledge storage to **retrieval-based reasoning** in language models.

🧠 Pretraining, Finetuning, and Joint Decoding of y

1) Pretraining

Purpose

To teach the model general language understanding **before** learning a specific task.

How it works

- The model is trained on large unlabeled text (Wikipedia, Books, Web).
- Common objective: **Masked Language Modeling (MLM)** — predict missing tokens.

Example:

```
Input: "The capital of France is [MASK]."
Model learns: "Paris"
```

Result

The model learns:

- Grammar
- Word meanings
- World knowledge
- Reasoning patterns

➡️ **But does NOT yet know how to perform specific tasks** (like QA, summarization, etc.).

---

2) Finetuning

Purpose

To adapt the pretrained model to a **specific task** using labeled examples.

Process

We train on task-specific datasets:

Task	Input	Output
QA	Question + passage	Answer
Classification	Text	Label
Summarization	Article	Summary

During finetuning:

- The model adjusts its weights to learn **how to map inputs → outputs** for the target task.

Result

The model now knows **how to use language knowledge to solve a real task**.

---

3) Joint Decoding of y (Answer Prediction with Retrieval)

This applies when the model:

- Retrieves external info (documents, passages)
- Then generates or selects the final answer **y**.

Core Idea

You **do retrieval and prediction together**, not separately.

$P(y \mid x) = \sum_{d \in \mathcal{D}} P(y \mid x, d) \cdot P(d \mid x)$

Where:

- (x) = input query
- (d) = retrieved document(s)
- (y) = output answer

Meaning

- The model **learns to retrieve** what is useful.
- At the same time, it **learns to generate the final output y** using that retrieved information.

This is called **joint training and joint decoding**.

🔁 Putting It All Together

Stage	What Happens	Why It Matters
Pretraining	Model learns general language knowledge	Builds strong base understanding
Finetuning	Model learns the specific task (QA, summarization, etc.)	Makes the model actually useful
Joint Decoding of y	Model retrieves and generates the answer in one process	Ensures the answer is **grounded** in retrieved evidence

---

🎯 Example (QA Task)

Query: "What is the capital of Japan?"

1. Pretraining taught the model how language works.
2. Finetuning taught it *how to answer questions.*
3. Joint decoding retrieves:
   - "Tokyo is the capital of Japan."

     And uses it to produce:
   - **Answer:** "Tokyo"

The answer is **not memorized;** it is **retrieved and verified**.

✅ One-Sentence Summary

> Pretraining teaches language patterns, finetuning trains the model on a specific task, and joint decoding ensures the final answer y is produced using both the query and the retrieved supporting documents.

🧠 Interaction Between Retrieved Passages (RAG-style Retrieval + LLM)

The slide is asking:

> If a Large Language Model (LLM) can handle a long context,

> can we just **give it the entire corpus** along with the question?

Problem

- Most corpora (e.g., Wikipedia) are **too large** to fit into the LLM input window.
- So instead of giving the whole corpus, we **retrieve only the most relevant pieces**.

This leads to the **retrieve → then generate** pipeline.

✅ Approximation Strategy (Steps)

1. **Encode the question**

   Convert the question into a **dense vector** using a retriever model (e.g., DPR, ColBERT, SPLADE+ANN search).

2. **Probe the dense index**

   The vector is used to **search a vector database** containing embeddings of all passages.

3. **Fetch top-k passages**

   Retrieve the **top ~20 most relevant passages** (k = 20 here).

4. **Concatenate (combine) the passages + question**

   All selected passages are placed **into the LLM context** in some chosen order

   (ordering question is important — sometimes sorted by relevance or chunk order).

5. **Let the LLM generate the answer**

   The LLM now **reads these passages and the question together**

   and outputs the answer using its decoder.

🧱 Visualization (Bottom of the Slide)

```
Passage 1 | Passage 2 | ... | Passage 20 | Question → (Encoder) → (Decoder) → Answer
```

- The LLM reads **20 passages + the question** as **input context**.
- Then the **decoder** generates the final answer.

This is the standard **RAG (Retrieval-Augmented Generation)** architecture.

💡 Key Insight Written on Slide

> "No loss of recall"

Meaning:

If retrieval works well, you don't need the **whole corpus**, only the **most relevant ~20 passages**.

So the LLM's reasoning quality does **not decrease**, even though the input size is much smaller.

🎯 Summary (Exam Ready)

Step	Action	Purpose
1	Encode question into dense vector	Convert text → searchable embedding
2	Retrieve top-k relevant passages	Reduces huge corpus → small relevant subset
3	Concatenate passages + question	Build the prompt context
4	LLM generates final answer	Uses retrieved evidence to produce grounded output

This is Retrieval-Augmented Generation:

retrieval provides information; LLMs provide reasoning and generation.

🧩 Fusion-in-Decoder (FiD)

Fusion-in-Decoder is a **retrieval-augmented generation** method for answering questions using **multiple retrieved passages**.

It is mostly used in **open-domain QA** tasks like:

- Natural Questions (NQ)
- TriviaQA
- SQuAD (open-domain version)

🧠 Core Idea

Instead of concatenating passages **before encoding**, FiD:

- Encodes **each retrieved passage separately,** together with the question.
- Then **fuses** all encoded representations **only in the decoder**.

This is why it is called:

> Fusion-in-Decoder
>
> (not Fusion-in-Encoder)

🔄 How FiD Works (Step-by-Step)

Step	What Happens	Why
1	Retrieve multiple passages using DPR / ColBERT / BM25	Gather evidence
2	For each passage: concatenate *Question + Passage*	Give context to encoder
3	Pass each pair **independently** through the encoder	Each passage forms its own encoded representation
4	**Concatenate all encoder outputs together**	This is the "fusion" step
5	The **decoder attends to the fused encoding** and generates answer	Decoder decides how to combine evidence

📊 Diagram Breakdown (Slide Visual)

```
Question + Passage 1 → Encoder → Encoded features ┐
Question + Passage 2 → Encoder → Encoded features ├── concat → Decoder → Answer
...
Question + Passage N → Encoder → Encoded features ┘
```

- **Encoders do *not* interact across passages**.
- **Fusion happens only in the decoder.**

🏁 Key Characteristics

Property	Meaning
Passages are encoded **separately**	No cross-passage mixing early on
Fusion is **late (in decoder)**	Decoder learns how to combine evidence
Useful for **single-hop QA**	Works well when answer appears in one passage
Less effective for **multi-hop reasoning**	Because passages don't talk to each other during encoding

🎯 Strengths

✅ **Simple and effective**

✅ Works very well when the answer exists in **one of the retrieved passages**

✅ Outperforms models like:

- Closed-book GPT-3
- DPR (bi-encoder retriever-based QA)
- REALM & early RAG models

⚠️ Limitations

❗ Passages **do not interact before the decoder** → hard to perform **multi-hop reasoning**

Example problem:

- If passage 1 tells *who*, and passage 2 tells *where*, FiD cannot easily combine them during encoding.

This is why FiD struggles with:

- Multi-hop datasets (e.g., HotpotQA)
- Reasoning chains requiring cross-passage linking

✅ One-Sentence Summary (Exam Ready)

> Fusion-in-Decoder retrieves multiple passages, encodes each with the question independently, and fuses them only in the decoder, making it powerful for single-

> hop QA but limited for multi-hop reasoning.

🔍 RAG: Retrieval-Augmented Generation (2020)

RAG combines:

1. A **retriever** (to fetch relevant text passages), and
2. A **generator** (to produce the final answer using the retrieved text).

✨ Goal

Instead of the model **memorizing facts**, it **retrieves them from an external corpus** and **generates grounded answers**.

🧠 How RAG Works (Pipeline)

1. **Encode the question** → convert to vector
2. **Search a dense index** → retrieve top-k relevant passages
3. **Feed the retrieved passages + question into a generator**
4. **Generator produces the final answer**

So:

$$\text{Answer} = \text{Generator}(\text{Question}, \text{Retrieved Passages})$$

The retriever usually uses:

- DPR (Dense Passage Retrieval)
- FAISS index
- Vector database

The generator is usually:

- BART or T5 seq2seq transformer

🎯 Retrieval Followed by Generation

🔷 Retriever

- Ranks passages for relevance
- Outputs top *k* passages (e.g., 5–20)

🔷 Generator (Decoder)

- Reads passages + question
- Produces answer one token at a time (autoregressively)

Key Point: Retrieval and generation are trained **jointly**, so the model *learns what to retrieve*.

🧩 RAG Variants

RAG has two decoding strategies depending on *how* retrieved passages influence generation:

---

1) RAG-Token

- *Retrieval happens at **every generated token**.

For each output token, the model **re-weights** which passage is most relevant.

$$P(y_i \mid y_{<i}, x) = \sum_{d \in D} P(y_i \mid y_{<i}, x, d) \cdot P(d \mid x)$$

Property	Meaning
Fine-grained fusion	The model chooses which passage contributes to each word it generates
Most accurate	Best factual grounding
More expensive	Retrieval reasoning happens per token

Useful when: The answer pieces come from **different passages**.

---

2) RAG-Sequence

Retrieval happens **once per entire answer**.

$$P(y \mid x) = \sum_{d \in D} P(y \mid x, d) \cdot P(d \mid x)$$

Property	Meaning
Coarse fusion	Chooses a passage once for the whole answer
Faster	Less compute
Slightly less accurate	Does not switch passages mid-answer

Useful when: The answer is contained mostly in **one passage**.

✅ Comparison Table

Feature	**RAG-Token**	**RAG-Sequence**
Fusion granularity	**Token-level**	**Sequence-level**
Retrieval influence	Dynamic per token	Fixed for entire answer
Accuracy	Higher	Slightly lower
Computation	Slower	Faster
Use-case	Multi-source answers	Single-source answers

 One-Sentence Exam-Ready Summaries

- **RAG:** Combines retrieval and generation so the model answers using external evidence, not just memory.
- **RAG-Token:** The model chooses which passage to rely on **for every generated token** (fine-grained fusion).
- **RAG-Sequence:** The model chooses a passage **once for the whole output** (coarse fusion).

🧠 Unlimiformer (2023)

Unlimiformer is a method designed to let **LLMs handle extremely long context lengths** (up to **millions of tokens**), **without modifying or retraining the model**.

It does this by adding a **retrieval mechanism inside the attention layer**, so the model reads only the **relevant parts** of a long context instead of everything.

🌍 The Problem It Solves

Standard Transformers have **quadratic attention complexity**:

$$\text{Attention} = O(n^2)$$

So:

- When **n** (context length) grows large → memory and compute explode.
- Feeding an entire book, database, or transcript becomes impossible.

Unlimiformer:

Instead of attending to **all** tokens, attention is limited to **only the most relevant ones**.

🎯 Key Idea

> Replace full attention over the input with retrieval-based sparse attention, using a vector index.

Meaning:

1. Break the input text into many small chunks.
2. Store their embeddings in a **vector database** (FAISS / ANN index).
3. During decoding, **each token** queries the index to retrieve **only relevant chunks**.
4. The model attends **only to those retrieved chunks**, not the whole input.

So the system behaves like:

- **Local Transformer + Global Retrieval**

⚙️ How It Works Step-by-Step

Step	What Happens	Purpose
1	Split long document into chunks (e.g., paragraphs)	Manageable units
2	Encode each chunk into a dense vector	Enable search
3	Store vectors in an ANN index	Efficient retrieval
4	During decoding, each token retrieves top-k relevant chunks	Relevance-based context

Step	What Happens	Purpose
5	Attention is computed **only** over the retrieved chunks	Avoids quadratic explosion

This allows the LLM to **behave as if it has unlimited context**, even though it doesn't.

🧠 Why It's Called *Unlimi-former*

- **Unlimited context** + **Transformer**
- No architectural changes → just **add retrieval** around it.

🔁 Comparison with RAG

Feature	RAG	Unlimiformer
Retrieval happens	Before generation (once)	During generation (every step)
Context fed to model	Fixed set of passages	Dynamically chosen per token
Focus	Retrieve facts	Maintain long sequence understanding
Best for	Open-domain QA	Book summarization, long transcripts

Unlimiformer = retrieval used to expand context window, not only to fetch facts.

🧩 Example

Task: Summarize a *300-page book*.

- Traditional LLM → cannot input full book.
- RAG → retrieves important passages but loses narrative flow.
- **Unlimiformer** → retrieves relevant text *on-the-fly* as it generates the summary.

So the summary remains **coherent + accurate**.

✅ Advantages

Benefit	Why it matters
Scales to millions of tokens	Handles entire books / long docs
No retraining required	Works with existing LLMs (GPT-type)

Benefit	Why it matters
Preserves coherence	Maintains context beyond window size
Efficient	Only attends to relevant chunks

⚠️ Limitations

Limitation	Cause
Retrieval errors → wrong context used	Depends on embedding quality
Slower inference	Retrieval occurs during generation
Needs vector index storage	Additional memory for chunks

🧾 One-Sentence Exam Answer

> Unlimiformer enables Transformers to handle very long contexts by adding retrieval-based sparse attention: instead of attending to all tokens, each generated token retrieves only the most relevant encoded chunks from a vector index, allowing models to process inputs far beyond their native context window.

🧠 Retrieval from Tables

Retrieval from tables refers to the process of **finding relevant rows, cells, or facts** from structured tabular data (like spreadsheets, CSV files, or SQL tables) **given a natural language query**.

This is different from retrieving text passages because tables have **structure** — rows, columns, headers, and cell values — which must be taken into account.

1️⃣ Why Retrieval from Tables is Needed

Many real-world facts are stored in tables:

- Product catalogs
- Financial reports
- Medical records
- Government datasets

Traditional text-based retrieval (e.g., BM25, DPR) does **not** work well because:

- Meaning is carried in the **relationships between columns**, not just words.

So we need retrieval systems that **understand table structure**.

2️⃣ Types of Queries Over Tables

Query Type	Example	Required Understanding
Lookup	"What is the capital of France?"	Find cell where row=France, col=Capital
Filter / Condition	"Countries with population > 50M"	Apply condition on a column
Aggregation	"How many gold medals did USA win?"	Sum or count column values
Multi-column reasoning	"Which city has the highest GDP per capita?"	Compare across multiple columns

---

3️⃣ Two Main Approaches

A) Structured Retrieval (SQL-style reasoning)

1. Convert query → logical form (SQL)
2. Execute query on table
3. Return answer

This is used in models like:

- **Seq2SQL**
- **Spider**
- **Tapas (Google)** (table-BERT model)

This approach works when:

- Table is structured
- Query matches the schema well

B) Embedding-Based Retrieval (Vector Search Over Tables)

Here, we convert table elements into **embeddings**:

- **Cell embeddings**
- **Row embeddings**
- **Column header embeddings**
- **Full table embeddings**

Then we perform similarity search between:

$$\text{Embedding}(\text{Query}) \quad \text{and} \quad \text{Embeddings}(\text{Table Data})$$

This is used in **RAG over Tables** and large LLM-based pipelines.

4️⃣ Encoding Strategy (How to Represent Tables)

Representation	Description	Strength
Row Embedding	Treat each row as a sentence	Good for lookup and filtering
Column Embedding	Represent each column header + type	Helps with schema understanding
Cell Embedding	Fine-grained encoding	Needed for cell-level QA
Table Embedding	Whole table encoded with attention	Good for summary or high-level reasoning

Many modern approaches combine multiple of these.

5️⃣ Model Example: TAPAS (Table-BERT)

TAPAS encodes:

- Question tokens
- Table cells
- Row/column positions

Using a **Transformer** with **row + column positional embeddings**.

The model predicts the answer by **selecting cells** or applying **aggregation** (SUM, COUNT, AVG).

This enables:

✅ Question answering

✅ Aggregation

✅ Logical reasoning

6️⃣ Interaction with RAG / LLMs

When used inside Retrieval-Augmented Generation:

1. Convert table to row/column/cell embeddings.
2. At query time, retrieve **relevant rows or tables**.
3. Provide the retrieved table content to the LLM to generate a natural answer.

This is called **Table-RAG**.

7️⃣ Key Challenges

Challenge	Explanation
Schema mismatch	Query terms don't match column names literally
Aggregation reasoning	Must perform math/logic, not just lookup
Multi-hop reasoning	Requires combining multiple rows/tables
LLM hallucination	LLM may invent values not in the table

✅ One-Sentence Summary (Exam Ready)

> Retrieval from tables means selecting the most relevant rows, cells, or table segments based on a question, using either structured SQL-like reasoning or embedding-based similarity search, and then providing the retrieved information to the model for answering.

🧠 Knowledge Graph RAG (Retrieval-Augmented Generation with Knowledge Graphs)

1️⃣ What is Knowledge Graph RAG?

Knowledge Graph RAG is a type of Retrieval-Augmented Generation where the **retrieval step comes from a Knowledge Graph (KG)** rather than (or in addition to) text documents.

Instead of retrieving **passages**, the system retrieves **entities and relations** from a KG, then provides that structured knowledge to the **LLM**, which generates the final answer.

2️⃣ Why Use a Knowledge Graph?

Text Retrieval (e.g., DPR, RAG)	Knowledge Graph Retrieval
Retrieves paragraphs / sentences	Retrieves **factual triples** (h, r, t)
Text may contain redundancy or noise	KG knowledge is **structured, precise**
LLM must infer relationships itself	KG **explicitly encodes relationships**
May hallucinate links	KG helps enforce **factual grounding**

Key Idea:

KG retrieval provides the model with **clean, verifiable facts** rather than long text blocks.

3️⃣ How Knowledge Graph RAG Works (Pipeline)

```
Query (natural language)
        ↓
(1) Entity Linking / Query Understanding
        ↓
Identify relevant entities in the KG
        ↓
(2) Subgraph Retrieval (Knowledge Retrieval)
        ↓
Return triples / relations / neighboring nodes
        ↓
(3) Convert KG facts into LLM-friendly text
        ↓
(4) LLM generates final answer
```

Breakdown:

Step	Description
1. Entity Linking	Match words in query to KG entities (e.g., “Tesla” → *Nikola Tesla* or *Tesla Inc.*).
2. Subgraph Retrieval	Retrieve relevant neighbors, paths, relations from the KG.
3. Format for Prompt	Convert structured triples → readable text or embeddings.
4. Generation	LLM uses grounded facts to produce the answer.

4️⃣ Types of Knowledge Fusion into the LLM

Approach	How KG knowledge enters
Text Conversion	Convert triples → natural language sentences in the prompt
Embedding Fusion	Embed KG entities + relations and combine with text embeddings
Graph Neural Networks (R-GCN)	Encode subgraph using GNN before sending to LLM
Neural SPARQL / Query Planning	Use KG reasoning first, then LLM fills explanation

5️⃣ Example

Query: *“Who discovered gravity?”*

KG Subgraph Retrieval:

```
(Newton, discovered, Gravity)
(Newton, bornIn, Lincolnshire)
(Newton, field, Physics)
```

Converted Prompt:

```
Facts:
- Isaac Newton is the scientist who discovered gravity.
- He was born in Lincolnshire.
- He worked in Physics.
Answer the question using these facts:
Question: Who discovered gravity?
```

LLM Output:

“Isaac Newton.”

→ No hallucination, grounded answer ✅

6️⃣ Benefits of KG-RAG

Benefit	Explanation
Reduces hallucination	Model grounded by verified structured facts
Supports multi-hop reasoning	KGs already encode relationships
More interpretable	You can show supporting triples or graph paths
Less noise than text retrieval	No long, irrelevant paragraphs

---

7️⃣ Limitations

Challenge	Impact
Requires entity linking	Errors cause wrong retrieval
KGs are often incomplete	Missing facts → incomplete answers
Hard to represent uncertainty	KGs are mostly binary truth-based
Complex to maintain and update	Especially domain-specific KGs

---

8️⃣ One-Sentence Exam Summary

> Knowledge Graph RAG retrieves structured factual triples or subgraphs from a KG and feeds them to the LLM, enabling grounded reasoning and reducing hallucination by combining symbolic knowledge with neural generation.
