Section 2:
Definition & Preliminaries

A Retrieval-based LM: Definition

A language model (LM) that uses
an external datastore at test time

A Retrieval-based LM: Definition

A language model (LM) that uses an external datastore at test time

A Retrieval-based LM: Definition

A language model (LM) that uses
an external datastore at test time

A Retrieval-based LM: Definition

A language model (LM) that uses an external datastore at test time

A language model (LM)

$$P(x_n \mid x_1, x_2, \cdots, x_{n-1})$$

A language model (LM)

$$P(x_n \mid x_1, x_2, \cdots, x_{n-1})$$

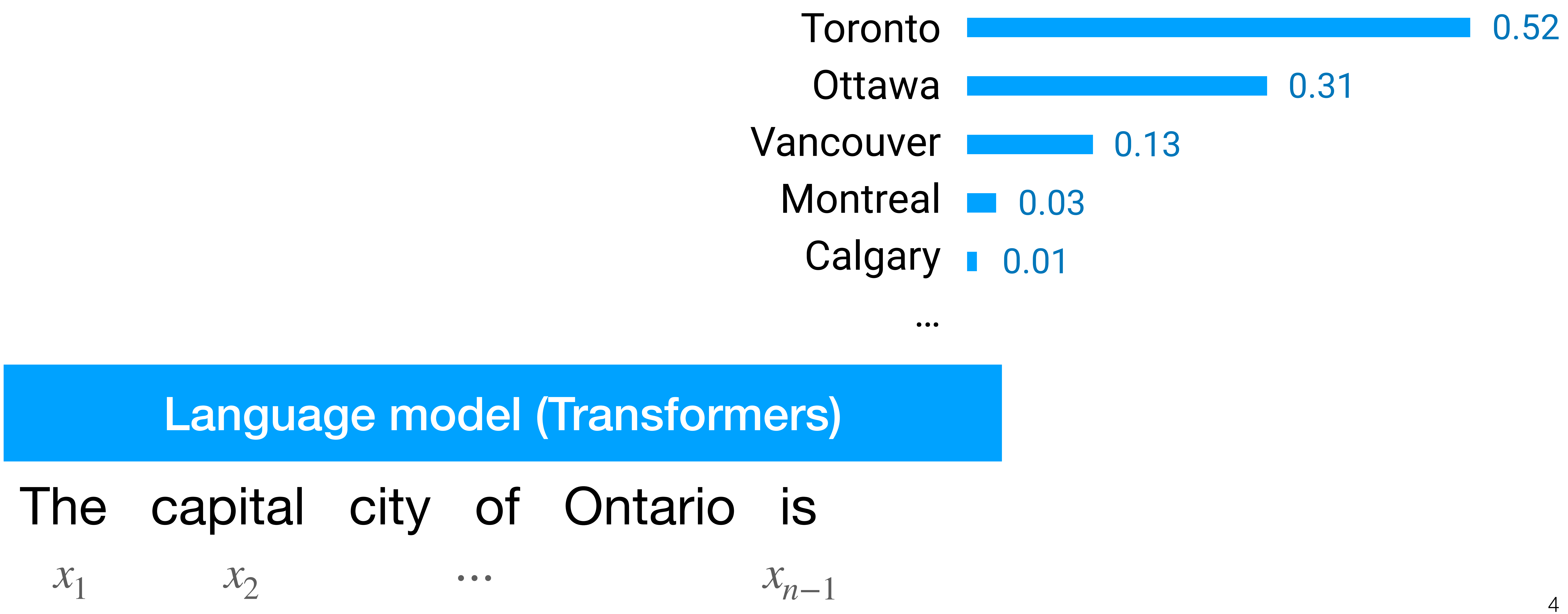
Language model (Transformers)

The capital city of Ontario is

x_1 x_2 $\cdots$ x_{n-1}

A language model (LM)

$$P(x_n \mid x_1, x_2, \cdots, x_{n-1})$$



A language model (LM): Categories

A language model (LM): Categories

Toronto

Autoregressive LM

The capital city of Ontario is ______

vs

capital Ontario

Masked LM

The ______ city of _______ is Toronto

A language model (LM): Categories

Toronto

Autoregressive LM

The capital city of Ontario is ______

vs

capital Ontario

Masked LM

The ______ city of _______ is Toronto

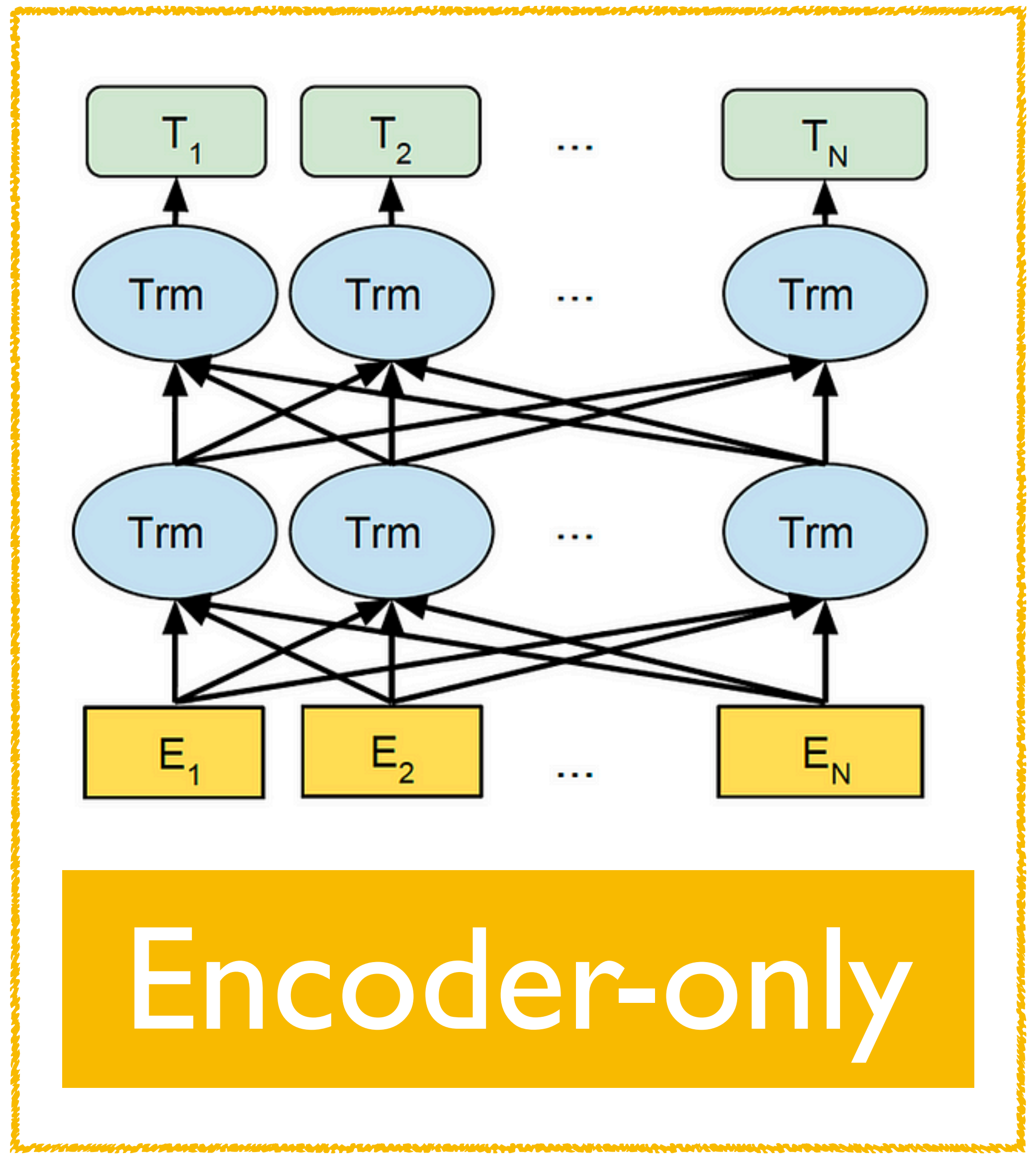


Image source: Devlin et al. 2019. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"

A language model (LM): Categories

Toronto

Autoregressive LM

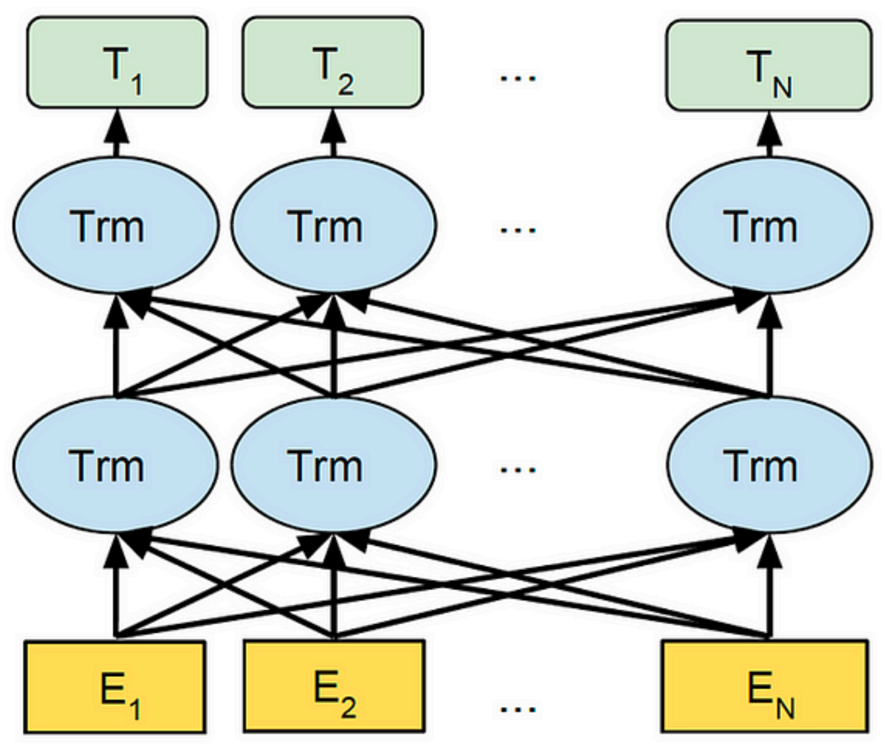
The capital city of Ontario is ______

vs

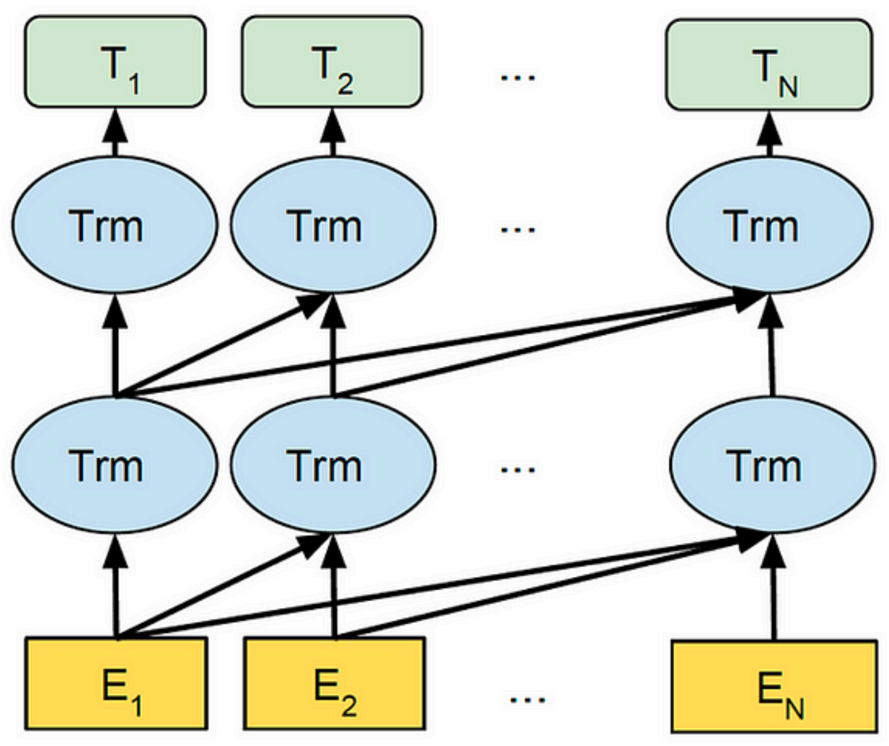
capital Ontario

Masked LM

The ______ city of _______ is Toronto



Encoder-only



Decoder-only

Image source: Devlin et al. 2019. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"

A language model (LM): Categories

Toronto

Autoregressive LM

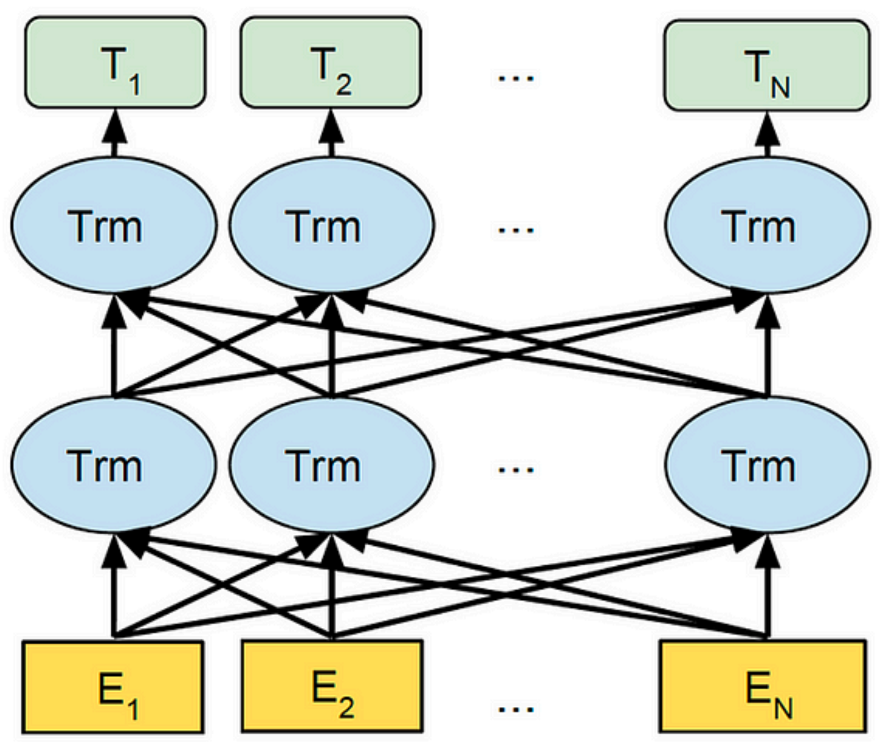
The capital city of Ontario is ______

vs

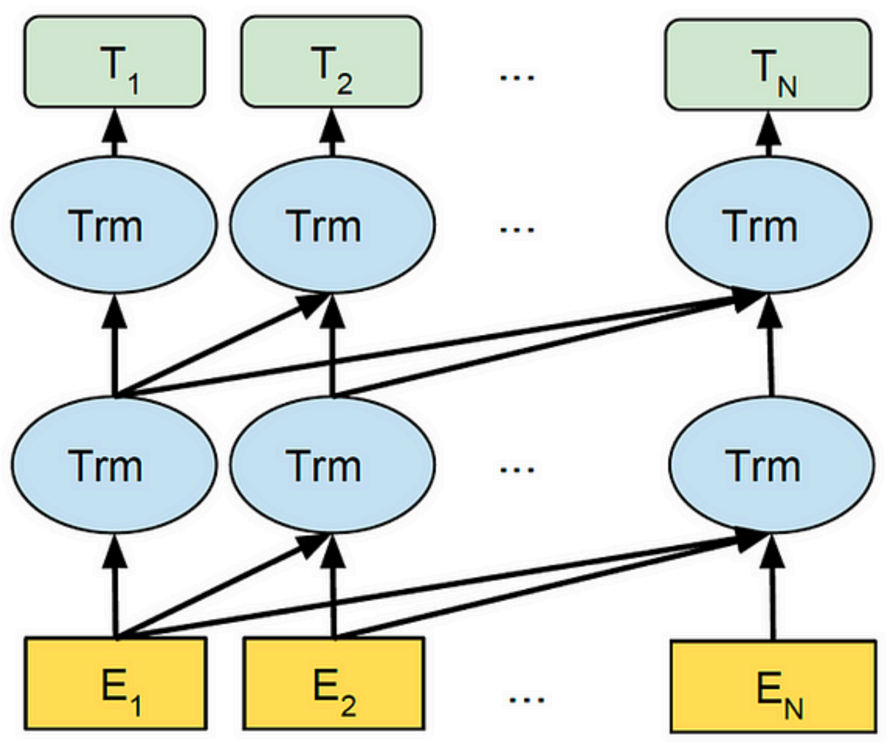
capital Ontario

Masked LM

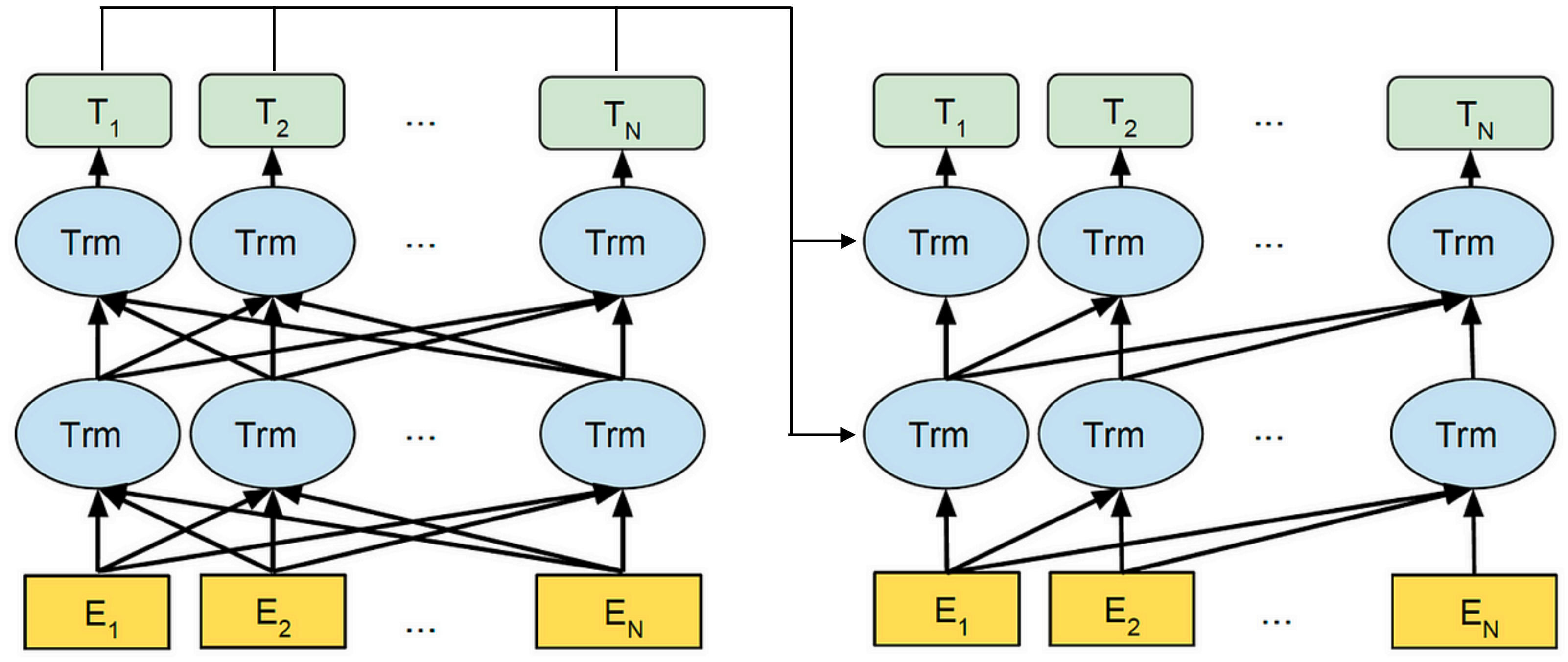
The ______ city of _______ is Toronto



Encoder-only



Decoder-only



Encoder-decoder

Image source: Devlin et al. 2019. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"

A language model (LM): Categories

Toronto

Autoregressive LM

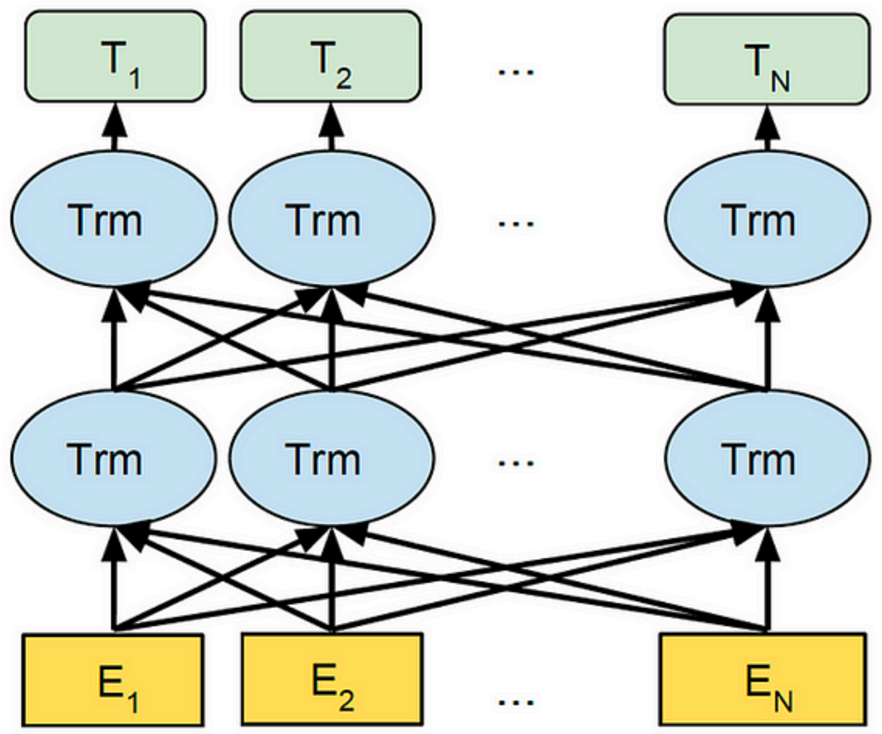
The capital city of Ontario is ______

vs

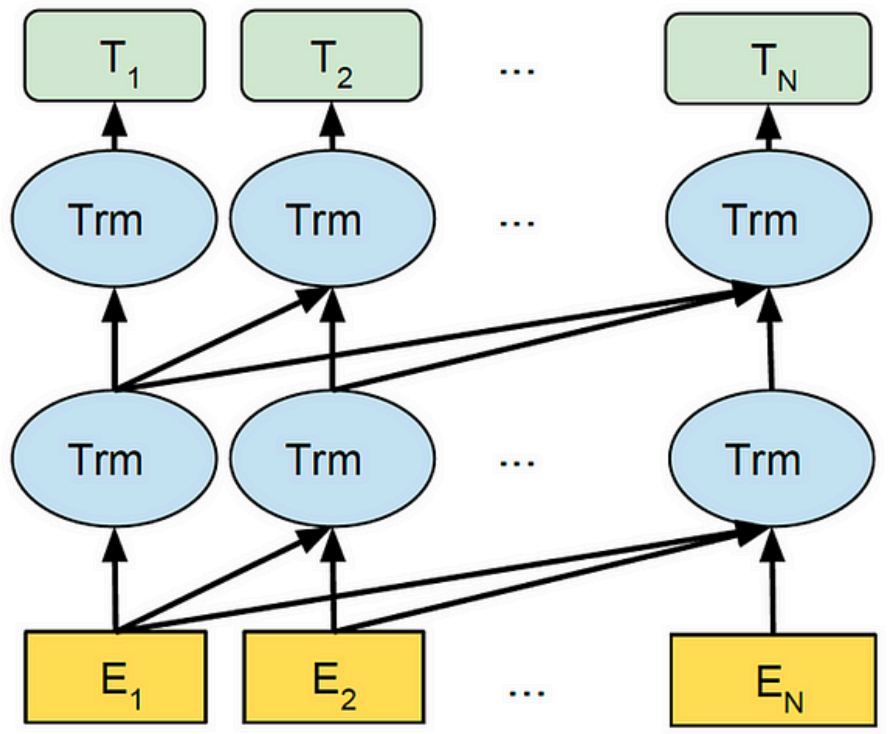
capital Ontario

Masked LM

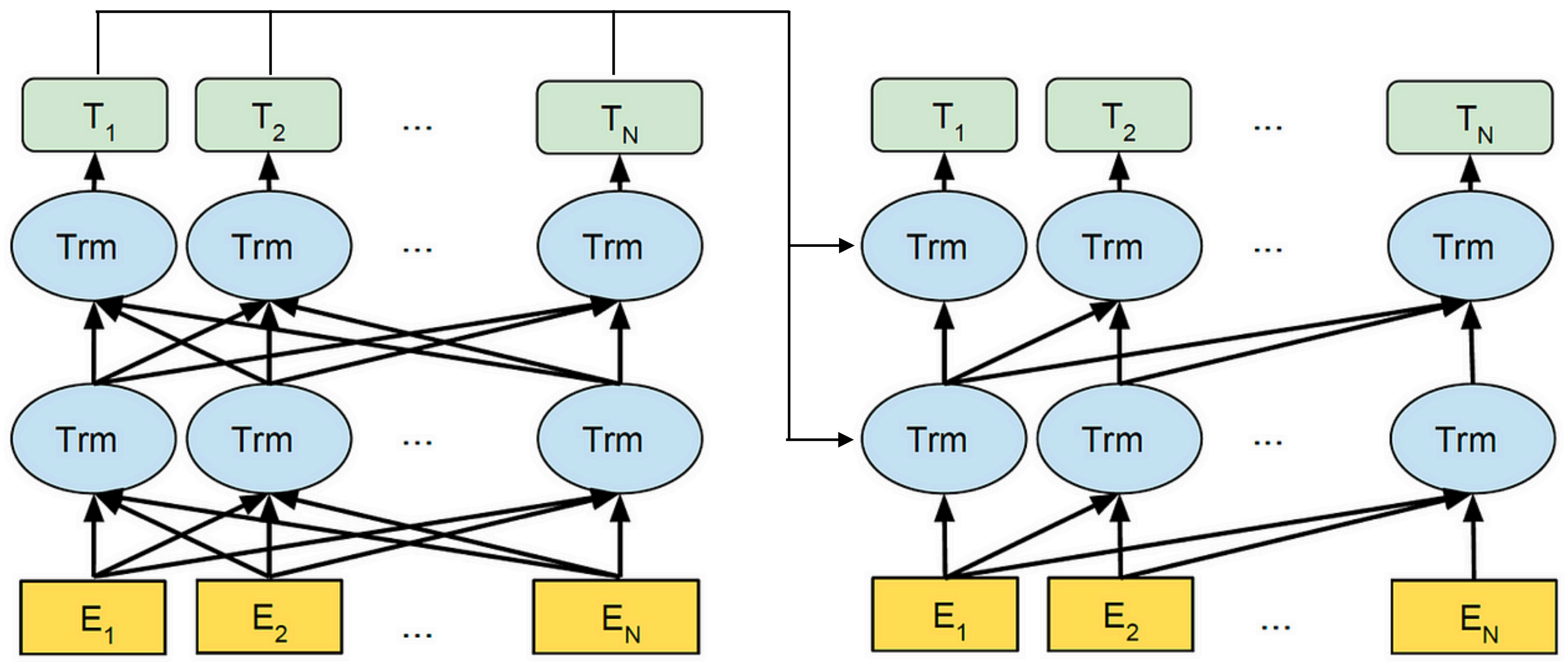
The ______ city of _______ is Toronto



Encoder-only



Decoder-only



Encoder-decoder

Image source: Devlin et al. 2019. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding"

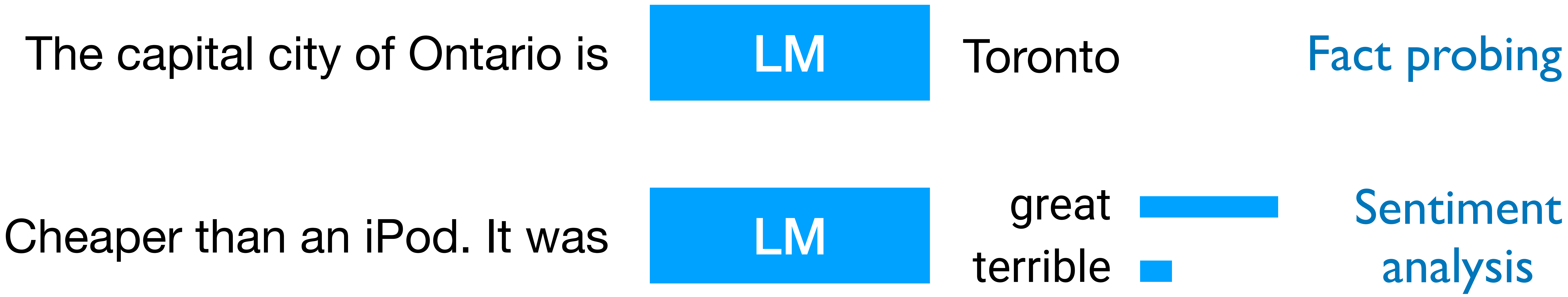
A language model (LM): Prompting

A language model (LM): Prompting

The capital city of Ontario is **LM** Toronto

Fact probing

A language model (LM): Prompting



A language model (LM): Prompting

The capital city of Ontario is → LM → Toronto — Fact probing

Cheaper than an iPod. It was → LM → great / terrible — Sentiment analysis

“Hello” in French is → LM → Bonjourno — Translation

A language model (LM): Prompting

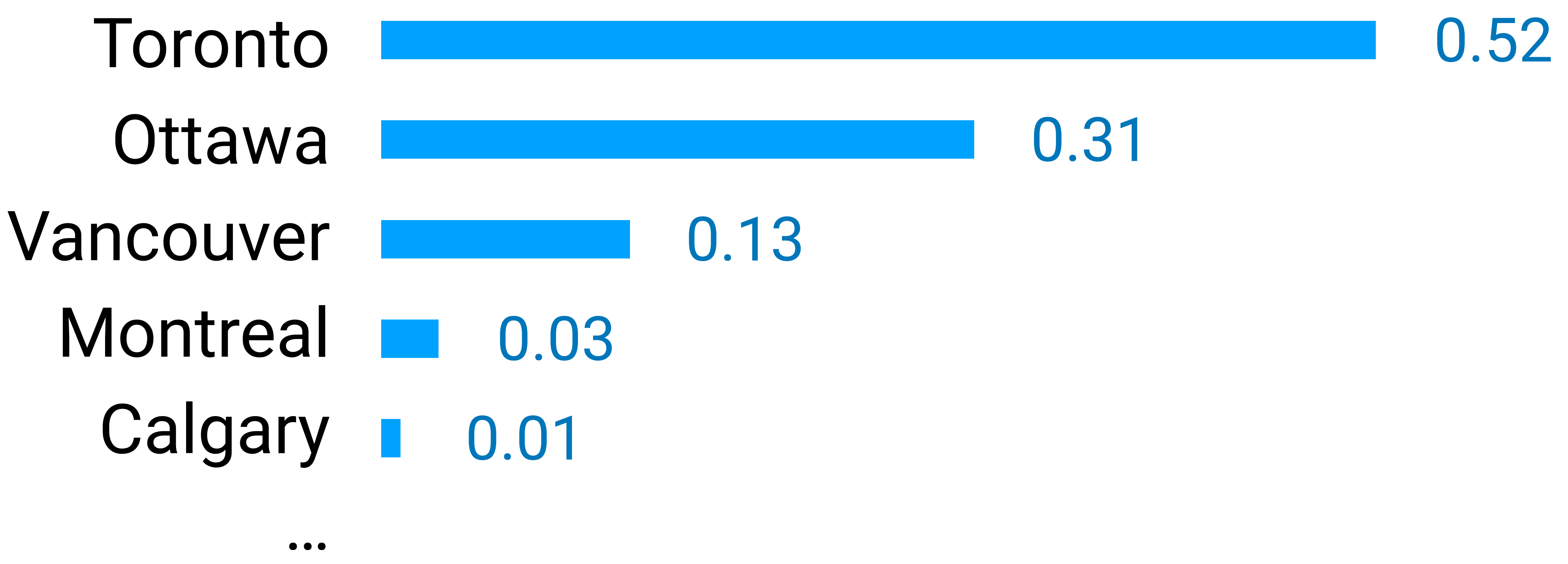
Prompt		Output	Task
The capital city of Ontario is	LM	Toronto	Fact probing
Cheaper than an iPod. It was	LM	great terrible	Sentiment analysis
“Hello” in French is	LM	Bonjourno	Translation
I’m good at math. 5 + 8 x 12 =	LM	101	Arithmetic

A language model (LM)

Often evaluated with

A language model (LM)

Often evaluated with

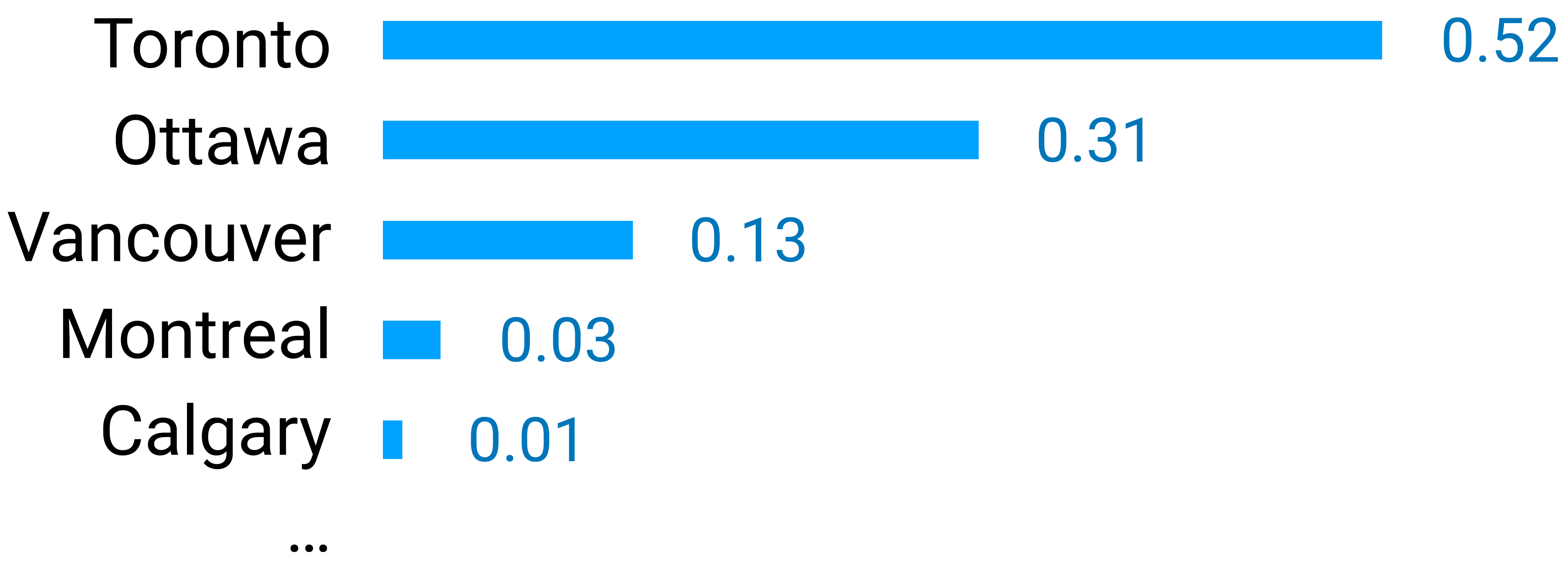


$$-\log(0.52) = 0.284$$

Perplexity

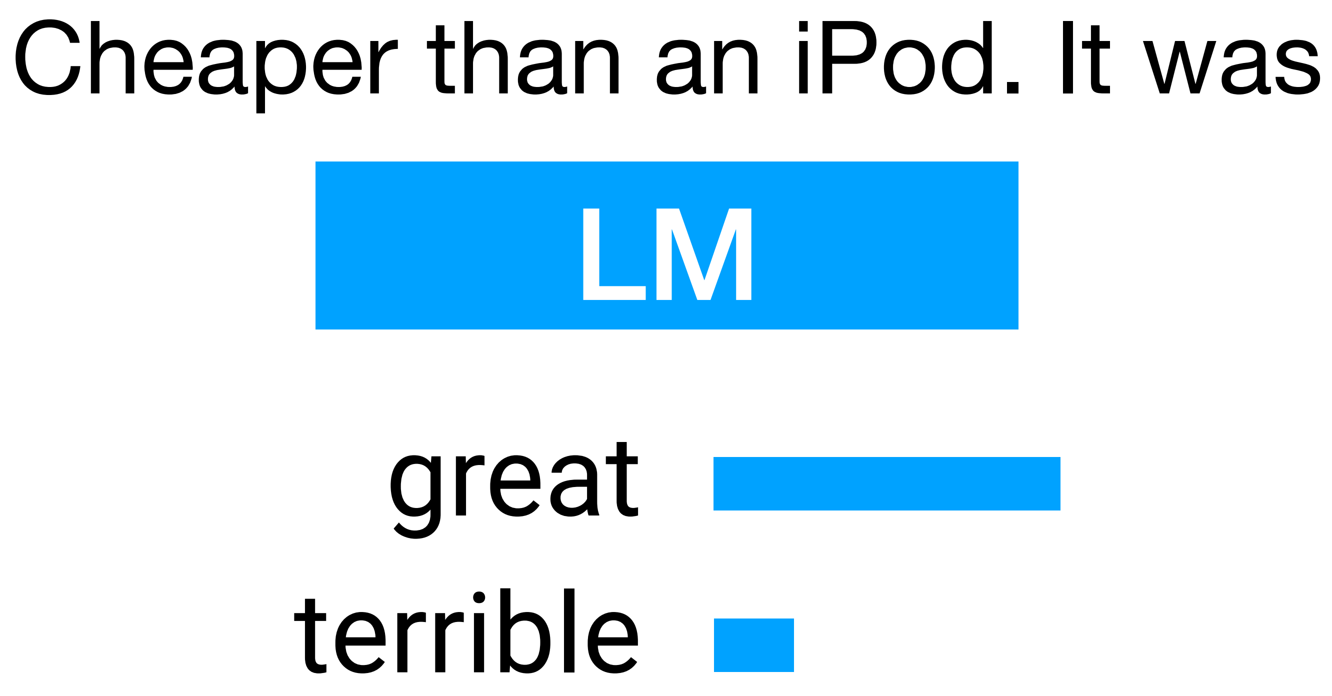
A language model (LM)

Often evaluated with



$$-\log(0.52) = 0.284$$

Perplexity



Prediction: positive ✓

Downstream accuracy

(Zero-shot or few-shot in-context learning, or fine-tuning)

(More in Section 5)

A Retrieval-based LM: Definition

A language model (LM) that uses an external datastore at test time

A Retrieval-based LM: Definition

A language model (LM) that uses
an external datastore at test time

Typical LMs

The capital city of Ontario is **Toronto**

LM

Training time

The capital city of Ontario is _____

LM

Test time

Retrieval-based LMs

The capital city of Ontario is **Toronto**

LM

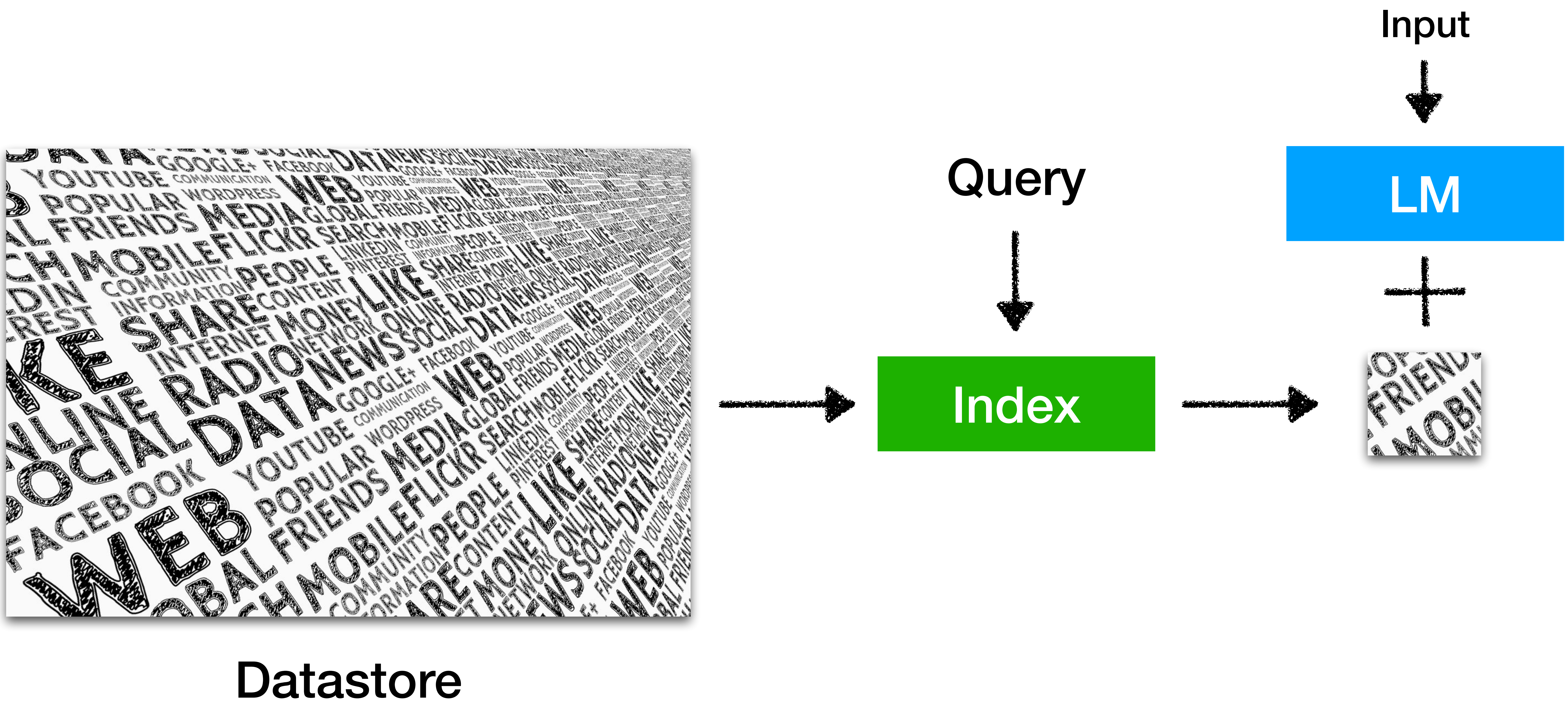
Training time

The capital city of Ontario is _____

LM

Test time

Inference



Inference: Datastore

Input

Query

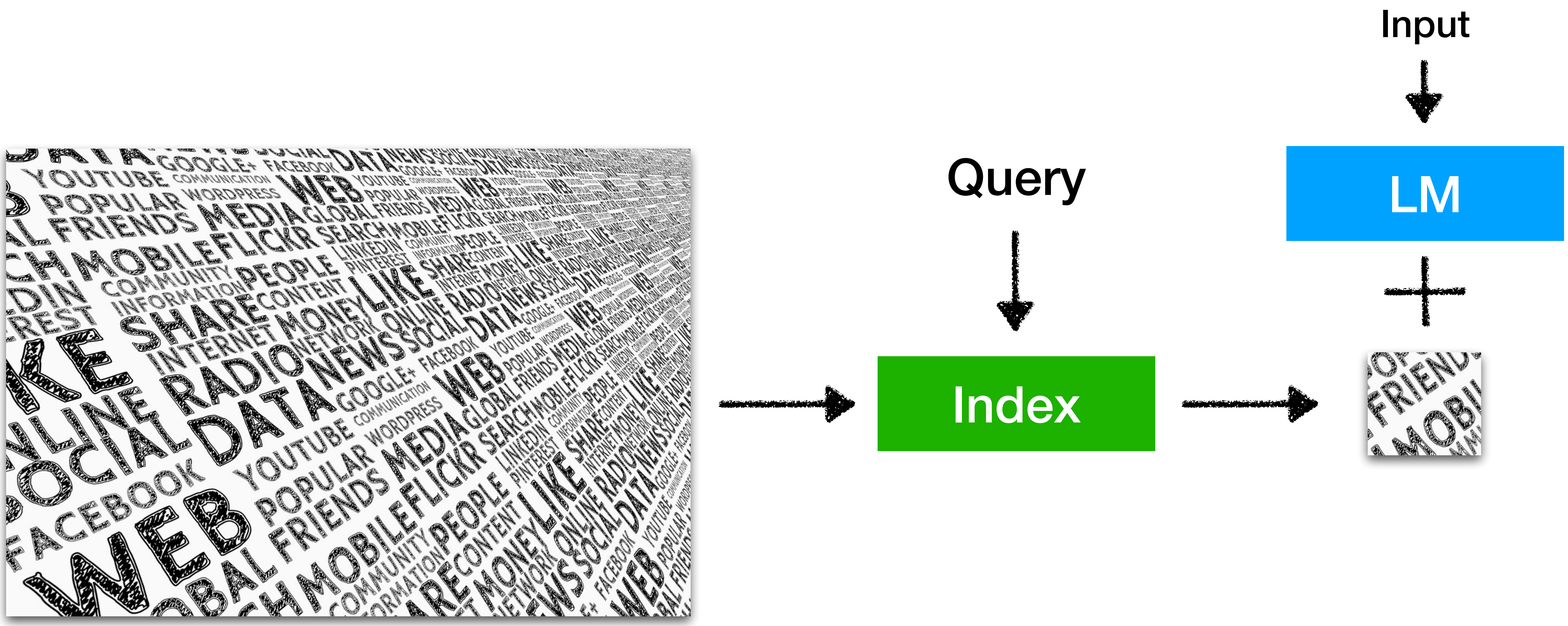
LM

Index

Datastore

Raw text corpus

Inference: Datastore

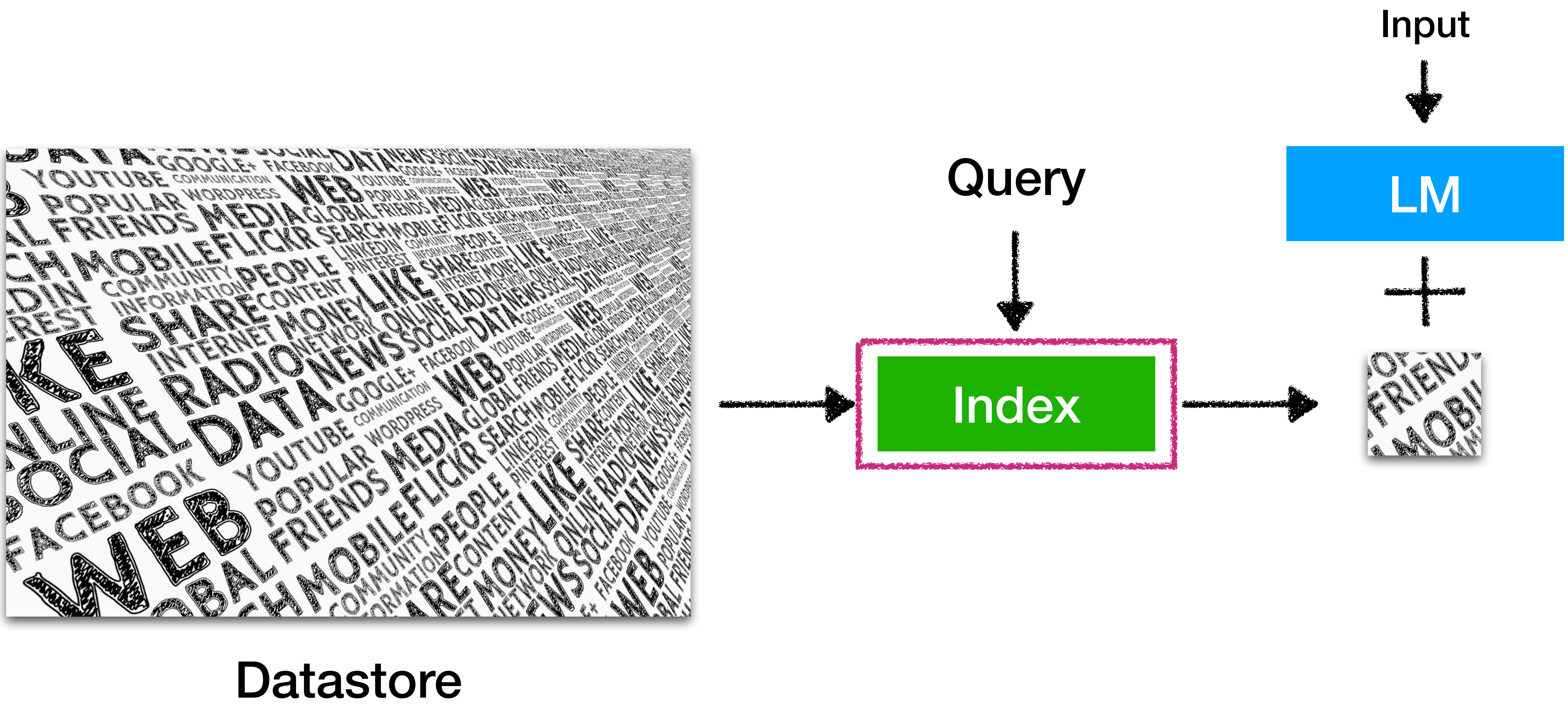


Datastore

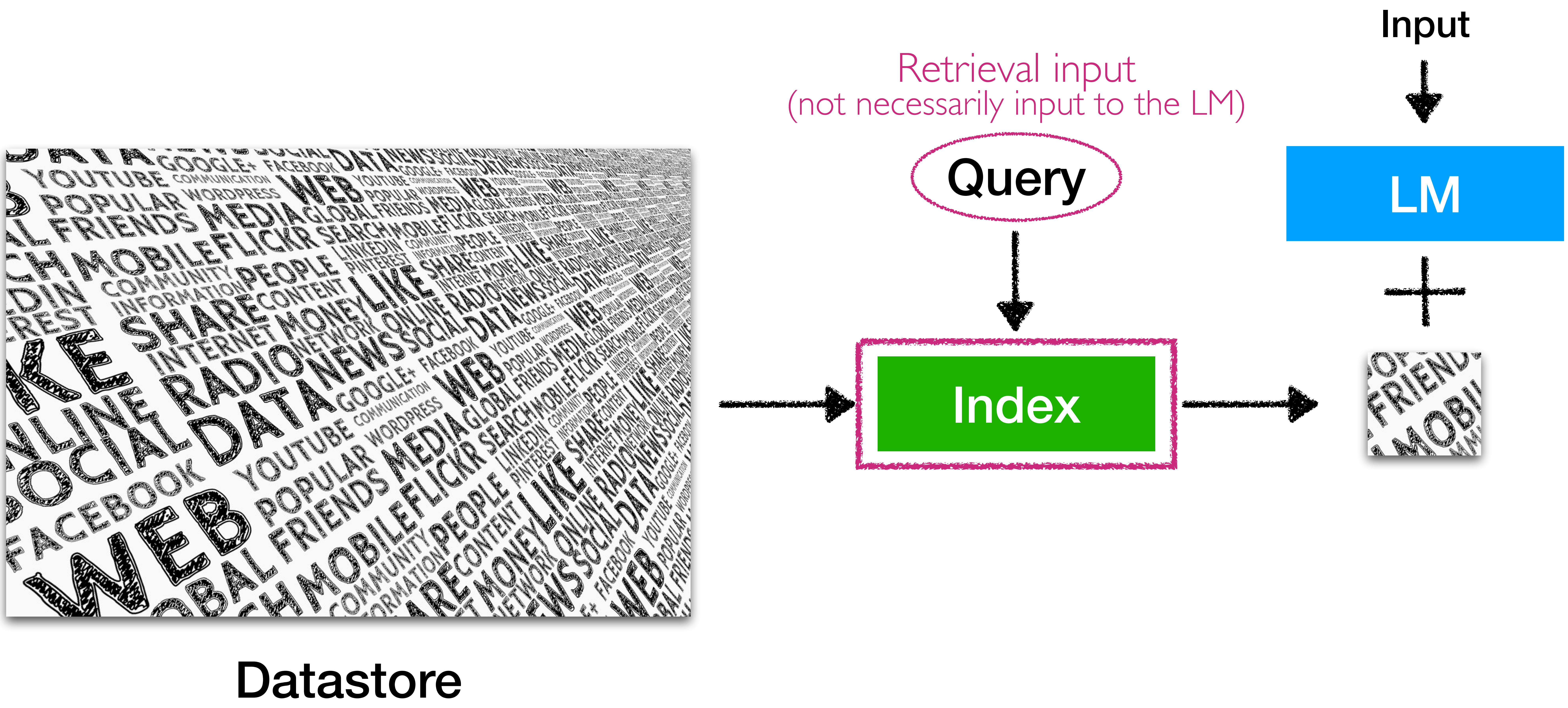
Raw text corpus

At least billions~trillions of tokens
Not labeled datasets
Not structured data (knowledge bases)

Inference: Index

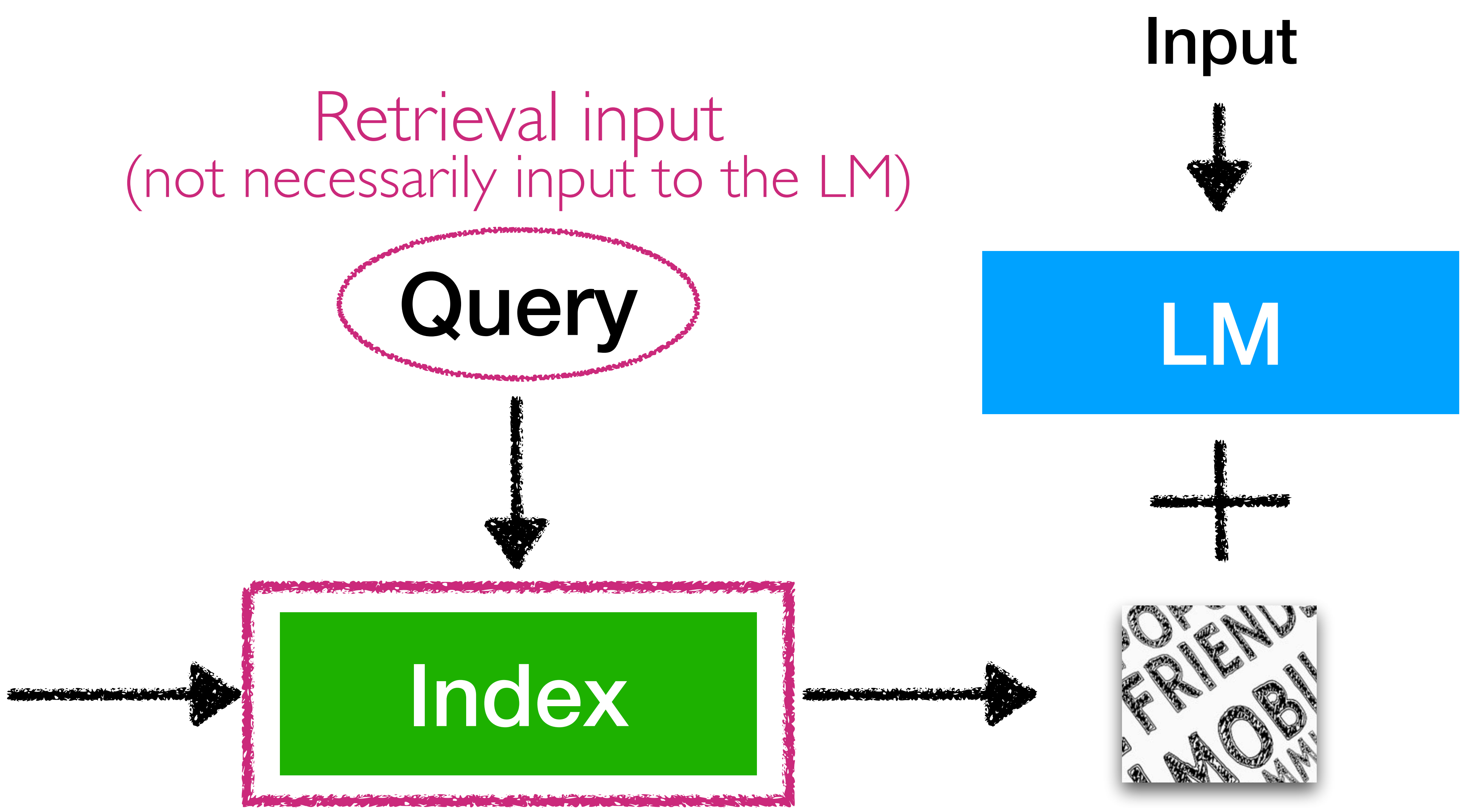


Inference: Index



Inference: Index

Datastore



Find a small subset of elements in a datastore that are the most similar to the query

Inference: Index

Goal: find a small subset of elements in a datastore that are the most similar to the query

Inference: Index

Goal: find a small subset of elements in a datastore
that are the most similar to the query

sim: a similarity score between two pieces of text

Inference: Index

Goal: find a small subset of elements in a datastore
that are the most similar to the query

sim: a similarity score between two pieces of text

Example

$$\text{sim}(i,j) = \text{tf}_{i,j} \times \log \frac{N}{\text{df}_i}$$

$\text{tf}_{i,j}$: # of occurrences of i in j

N: # of total docs

df_i: # of docs containing i

Inference: Index

Goal: find a small subset of elements in a datastore that are the most similar to the query

sim: a similarity score between two pieces of text

Example

$$\text{sim}(i, j) = \text{tf}_{i,j} \times \log \frac{N}{\text{df}_i}$$

$\text{tf}_{i,j}$: # of occurrences of i in j

N: # of total docs

df_i: # of docs containing i

Example

$$\text{sim}(i, j) = \text{Encoder}(i) \cdot \text{Encoder}(j)$$

Encoder: Maps the text into an h-dimensional vector

Inference: Index

Goal: find a small subset of elements in a datastore that are the most similar to the query

sim: a similarity score between two pieces of text

Example

$$\text{sim}(i, j) = \text{tf}_{i,j} \times \log \frac{N}{\text{df}_i}$$

$\text{tf}_{i,j}$: # of occurrences of i in j

N: # of total docs

df_i: # of docs containing i

Example

$$\text{sim}(i, j) = \text{Encoder}(i) \cdot \text{Encoder}(j)$$

Encoder: Maps the text into an h-dimensional vector

An entire field of study on how to get (or learn) the similarity function better

(We'll see some in Section 4)

Inference: Index

Goal: find a small subset of elements in a datastore
that are the most similar to the query

sim: a similarity score between two pieces of text

Index: given q, return $\mathrm{argTop}\text{-}k_{d \in \mathcal{D}} \mathrm{sim}(q, d)$ through fast nearest neighbor search

Inference: Index

Goal: find a small subset of elements in a datastore that are the most similar to the query

sim: a similarity score between two pieces of text

Index: given q, return $\text{argTop-}k_{d \in \mathcal{D}} \text{sim}(q, d)$ through fast nearest neighbor search

k elements from a datastore

Inference: Index

Goal: find a small subset of elements in a datastore
that are the most similar to the query

sim: a similarity score between two pieces of text

Can be a totally separate research area on
how to do this fast & accurate

Index: given q, return $\text{argTop-}k_{d \in \mathcal{D}} \text{sim}(q, d)$ through fast nearest neighbor search

k elements from a datastore

Software: FAISS, Distributed FAISS, SCaNN, etc…

Software: FAISS, Distributed FAISS, SCaNN, etc…

Method	Class name	`index_factory`	Main parameters	Bytes/vector	Exhaustive	Comments
Exact Search for L2	`IndexFlatL2`	`"Flat"`	`d`	`4*d`	yes	brute-force
Exact Search for Inner Product	`IndexFlatIP`	`"Flat"`	`d`	`4*d`	yes	also for cosine (normalize vectors beforehand)
Hierarchical Navigable Small World graph exploration	`IndexHNSWFlat`	`"HNSW,Flat"`	`d`, `M`	`4*d + x * M * 2 * 4`	no	
Inverted file with exact post-verification	`IndexIVFFlat`	`"IVFx,Flat"`	`quantizer`, `d`, `nlists`, `metric`	`4*d + 8`	no	Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.
Locality-Sensitive Hashing (binary flat index)	`IndexLSH`	-	`d`, `nbits`	`ceil(nbits/8)`	yes	optimized by using random rotation instead of random projections
Scalar quantizer (SQ) in flat mode	`IndexScalarQuantizer`	`"SQ8"`	`d`	`d`	yes	4 and 6 bits per component are also implemented.
Product quantizer (PQ) in flat mode	`IndexPQ`	`"PQx"`, `"PQ"M"x"nbits`	`d`, `M`, `nbits`	`ceil(M * nbits / 8)`	yes	
IVF and scalar quantizer	`IndexIVFScalarQuantizer`	"IVFx,SQ4" "IVFx,SQ8"	`quantizer`, `d`, `nlists`, `qtype`	SQfp16: 2 * `d` + 8, SQ8: `d` + 8 or SQ4: `d/2` + 8	no	Same as the `IndexScalarQuantizer`
IVFADC (coarse quantizer+PQ on residuals)	`IndexIVFPQ`	`"IVFx,PQ"y"x"nbits`	`quantizer`, `d`, `nlists`, `M`, `nbits`	`ceil(M * nbits/8)+8`	no	
IVFADC+R (same as IVFADC with re-ranking based on codes)	`IndexIVFPQR`	`"IVFx,PQy+z"`	`quantizer`, `d`, `nlists`, `M`, `nbits`, `M_refine`, `nbits_refine`	`M+M_refine+8`	no	

Software: FAISS, Distributed FAISS, SCaNN, etc…

Method	Class name	index_factory	Main parameters	Bytes/vector	Exhaustive	Comments
Exact Search for L2	IndexFlatL2	"Flat"	d	4*d	yes	brute-force
Exact Search for Inner Product	IndexFlatIP	"Flat"	d	4*d	yes	also for cosine (normalize vectors beforehand)
Hierarchical Navigable Small World graph exploration	IndexHNSWFlat	"HNSW,Flat"	d, M	4*d + x * M * 2 * 4	no	
Inverted file with exact post-verification	IndexIVFFlat	"IVFx,Flat"	quantizer, d, nlists, metric	4*d + 8	no	Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.
Locality-Sensitive Hashing (binary flat index)	IndexLSH	-	d, nbits	ceil(nbits/8)	yes	optimized by using random rotation instead of random projections
Scalar quantizer (SQ) in flat mode	IndexScalarQuantizer	"SQ8"	d	d	yes	4 and 6 bits per component are also implemented.
Product quantizer (PQ) in flat mode	IndexPQ	"PQx", "PQ"M"x"nbits	d, M, nbits	ceil(M * nbits / 8)	yes	
IVF and scalar quantizer	IndexIVFScalarQuantizer	"IVFx,SQ4" "IVFx,SQ8"	quantizer, d, nlists, qtype	SQfp16: 2 * d + 8, SQ8: d + 8 or SQ4: d/2 + 8	no	Same as the IndexScalarQuantizer
IVFADC (coarse quantizer+PQ on residuals)	IndexIVFPQ	"IVFx,PQ"y"x"nbits	quantizer, d, nlists, M, nbits	ceil(M * nbits/8)+8	no	
IVFADC+R (same as IVFADC with re-ranking based on codes)	IndexIVFPQR	"IVFx,PQy+z"	quantizer, d, nlists, M, nbits, M_refine, nbits_refine	M+M_refine+8	no	

Exact Search

Software: FAISS, Distributed FAISS, SCaNN, etc…

Method	Class name	index_factory	Main parameters	Bytes/vector	Exhaustive	Comments
Exact Search for L2	IndexFlatL2	"Flat"	d	4*d	yes	brute-force
Exact Search for Inner Product	IndexFlatIP	"Flat"	d	4*d	yes	also for cosine (normalize vectors beforehand)
Hierarchical Navigable Small World graph exploration	IndexHNSWFlat	"HNSW,Flat"	d, M	4*d + x * M * 2 * 4	no	
Inverted file with exact post-verification	IndexIVFFlat	"IVFx,Flat"	quantizer, d, nlists, metric	4*d + 8	no	Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.
Locality-Sensitive Hashing (binary flat index)	IndexLSH	-	d, nbits	ceil(nbits/8)	yes	optimized by using random rotation instead of random projections
Scalar quantizer (SQ) in flat mode	IndexScalarQuantizer	"SQ8"	d	d	yes	4 and 6 bits per component are also implemented.
Product quantizer (PQ) in flat mode	IndexPQ	"PQx", "PQ"M"x"nbits	d, M, nbits	ceil(M * nbits / 8)	yes	
IVF and scalar quantizer	IndexIVFScalarQuantizer	"IVFx,SQ4" "IVFx,SQ8"	quantizer, d, nlists, qtype	SQfp16: 2 * d + 8, SQ8: d + 8 or SQ4: d/2 + 8	no	Same as the IndexScalarQuantizer
IVFADC (coarse quantizer+PQ on residuals)	IndexIVFPQ	"IVFx,PQ"y"x"nbits	quantizer, d, nlists, M, nbits	ceil(M * nbits/8)+8	no	
IVFADC+R (same as IVFADC with re-ranking based on codes)	IndexIVFPQR	"IVFx,PQy+z"	quantizer, d, nlists, M, nbits, M_refine, nbits_refine	M+M_refine+8	no	

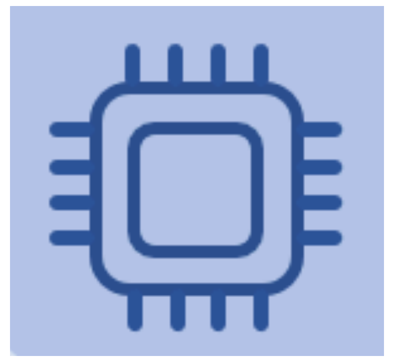
Exact Search

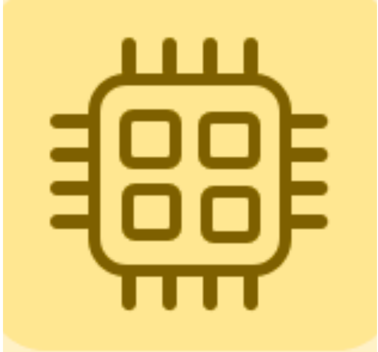
Approximate Search
(Relatively easy to scale to ~1B elements)

Software: FAISS, Distributed FAISS, SCaNN, etc…

Method	Class name	index_factory	Main parameters	Bytes/vector	Exhaustive	Comments
Exact Search for L2	`IndexFlatL2`	`"Flat"`	`d`	`4*d`	yes	brute-force
Exact Search for Inner Product	`IndexFlatIP`	`"Flat"`	`d`	`4*d`	yes	also for cosine (normalize vectors beforehand)
Hierarchical Navigable Small World graph exploration	`IndexHNSWFlat`	`"HNSW,Flat"`	`d`, `M`	`4*d + x * M * 2 * 4`	no	
Inverted file with exact post-verification	`IndexIVFFlat`	`"IVFx,Flat"`	`quantizer`, `d`, `nlists`, `metric`	`4*d + 8`	no	Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.
Locality-Sensitive Hashing (binary flat index)	`IndexLSH`	-	`d`, `nbits`	`ceil(nbits/8)`	yes	optimized by using random rotation instead of random projections
Scalar quantizer (SQ) in flat mode	`IndexScalarQuantizer`	`"SQ8"`	`d`	`d`	yes	4 and 6 bits per component are also implemented.
Product quantizer (PQ) in flat mode	`IndexPQ`	`"PQx"`, `"PQ"M"x"nbits`	`d`, `M`, `nbits`	`ceil(M * nbits / 8)`	yes	
IVF and scalar quantizer	`IndexIVFScalarQuantizer`	"IVFx,SQ4" "IVFx,SQ8"	`quantizer`, `d`, `nlists`, `qtype`	SQfp16: 2 * `d` + 8, SQ8: `d` + 8 or SQ4: `d/2` + 8	no	Same as the `IndexScalarQuantizer`
IVFADC (coarse quantizer+PQ on residuals)	`IndexIVFPQ`	`"IVFx,PQ"y"x"nbits`	`quantizer`, `d`, `nlists`, `M`, `nbits`	`ceil(M * nbits/8)+8`	no	
IVFADC+R (same as IVFADC with re-ranking based on codes)	`IndexIVFPQR`	`"IVFx,PQy+z"`	`quantizer`, `d`, `nlists`, `M`, `nbits`, `M_refine`, `nbits_refine`	`M+M_refine+8`	no	

Exact Search





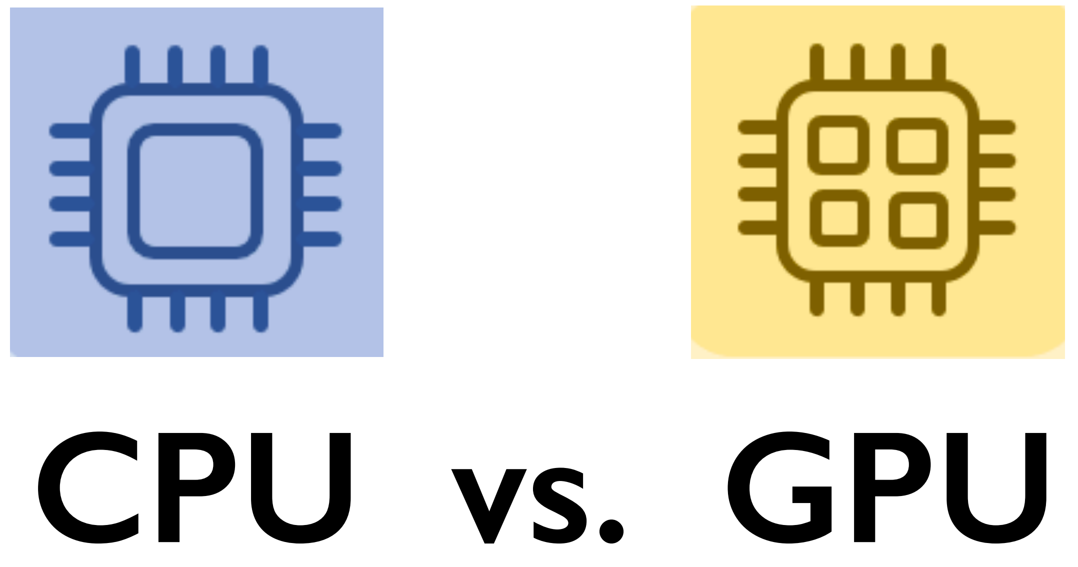
CPU vs. GPU

Approximate Search
(Relatively easy to scale to ~1B elements)

Software: FAISS, Distributed FAISS, SCaNN, etc…

Method	Class name	index_factory	Main parameters	Bytes/vector	Exhaustive	Comments
Exact Search for L2	IndexFlatL2	"Flat"	d	4*d	yes	brute-force
Exact Search for Inner Product	IndexFlatIP	"Flat"	d	4*d	yes	also for cosine (normalize vectors beforehand)
Hierarchical Navigable Small World graph exploration	IndexHNSWFlat	"HNSW,Flat"	d, M	4*d + x * M * 2 * 4	no	
Inverted file with exact post-verification	IndexIVFFlat	"IVFx,Flat"	quantizer, d, nlists, metric	4*d + 8	no	Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.
Locality-Sensitive Hashing (binary flat index)	IndexLSH	-	d, nbits	ceil(nbits/8)	yes	optimized by using random rotation instead of random projections
Scalar quantizer (SQ) in flat mode	IndexScalarQuantizer	"SQ8"	d	d	yes	4 and 6 bits per component are also implemented.
Product quantizer (PQ) in flat mode	IndexPQ	"PQx", "PQ"M"x"nbits	d, M, nbits	ceil(M * nbits / 8)	yes	
IVF and scalar quantizer	IndexIVFScalarQuantizer	"IVFx,SQ4" "IVFx,SQ8"	quantizer, d, nlists, qtype	SQfp16: 2 * d + 8, SQ8: d + 8 or SQ4: d/2 + 8	no	Same as the IndexScalarQuantizer
IVFADC (coarse quantizer+PQ on residuals)	IndexIVFPQ	"IVFx,PQ"y"x"nbits	quantizer, d, nlists, M, nbits	ceil(M * nbits/8)+8	no	
IVFADC+R (same as IVFADC with re-ranking based on codes)	IndexIVFPQR	"IVFx,PQy+z"	quantizer, d, nlists, M, nbits, M_refine, nbits_refine	M+M_refine+8	no	

Exact Search



Approximate Search
(Relatively easy to scale to ~1B elements)

More info: https://github.com/facebookresearch/faiss/wiki

Inference: Search

Datastore

Query

Index

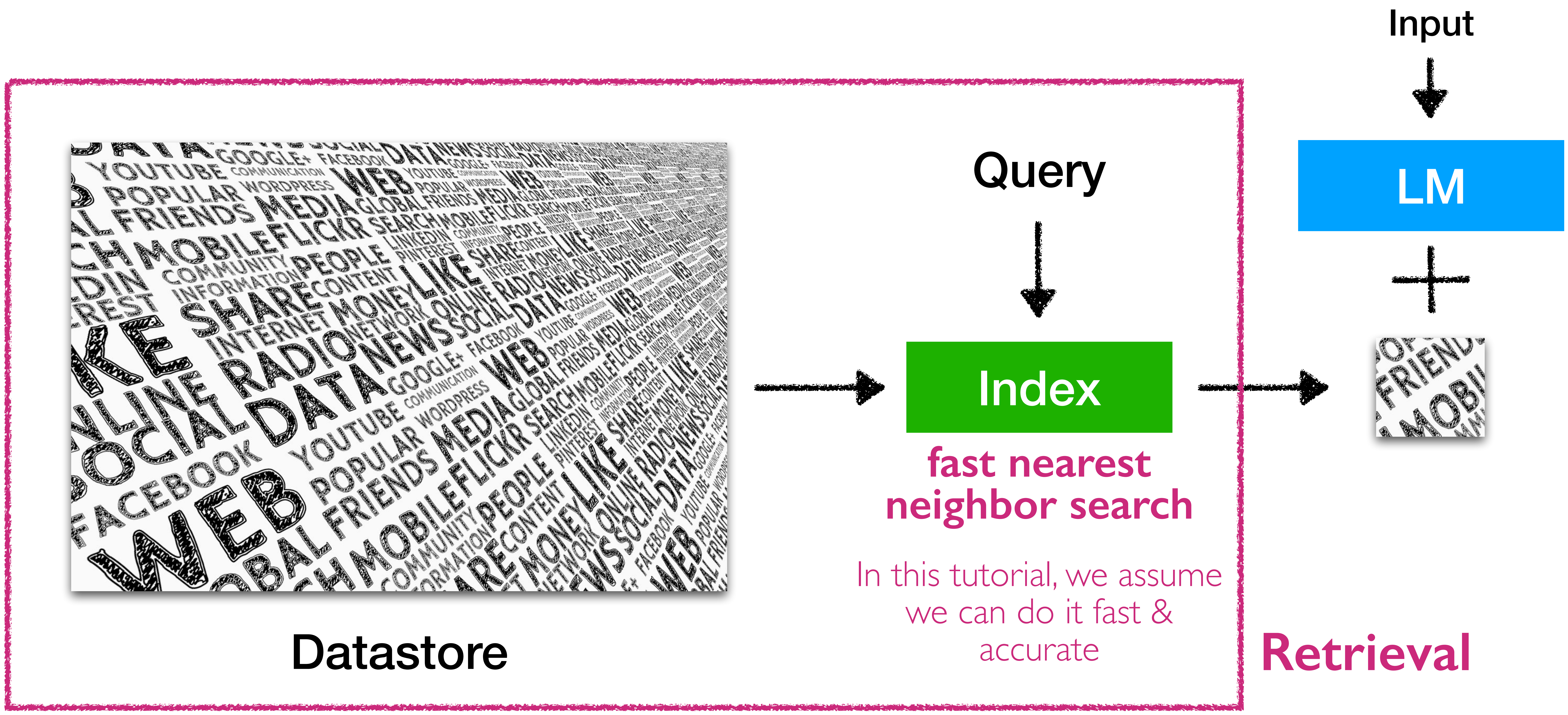
fast nearest neighbor search

In this tutorial, we assume we can do it fast & accurate

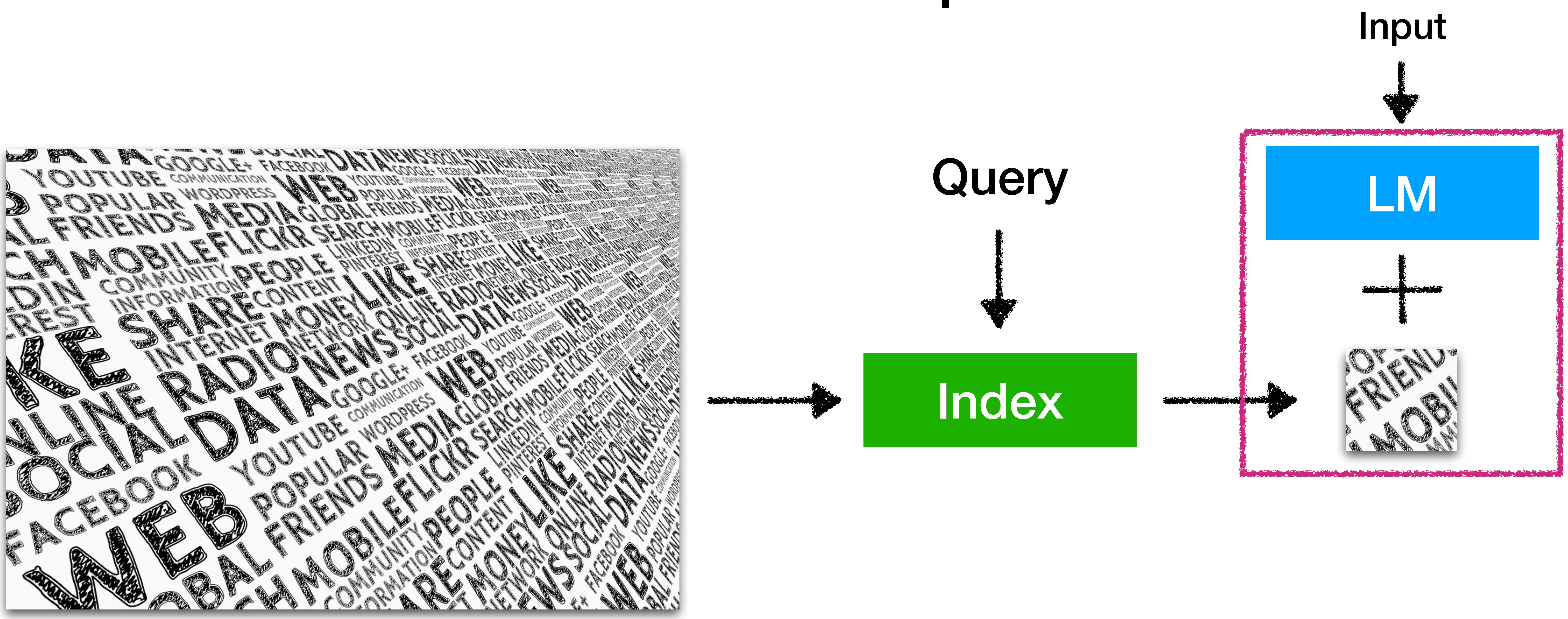
Input

LM

Inference: Search

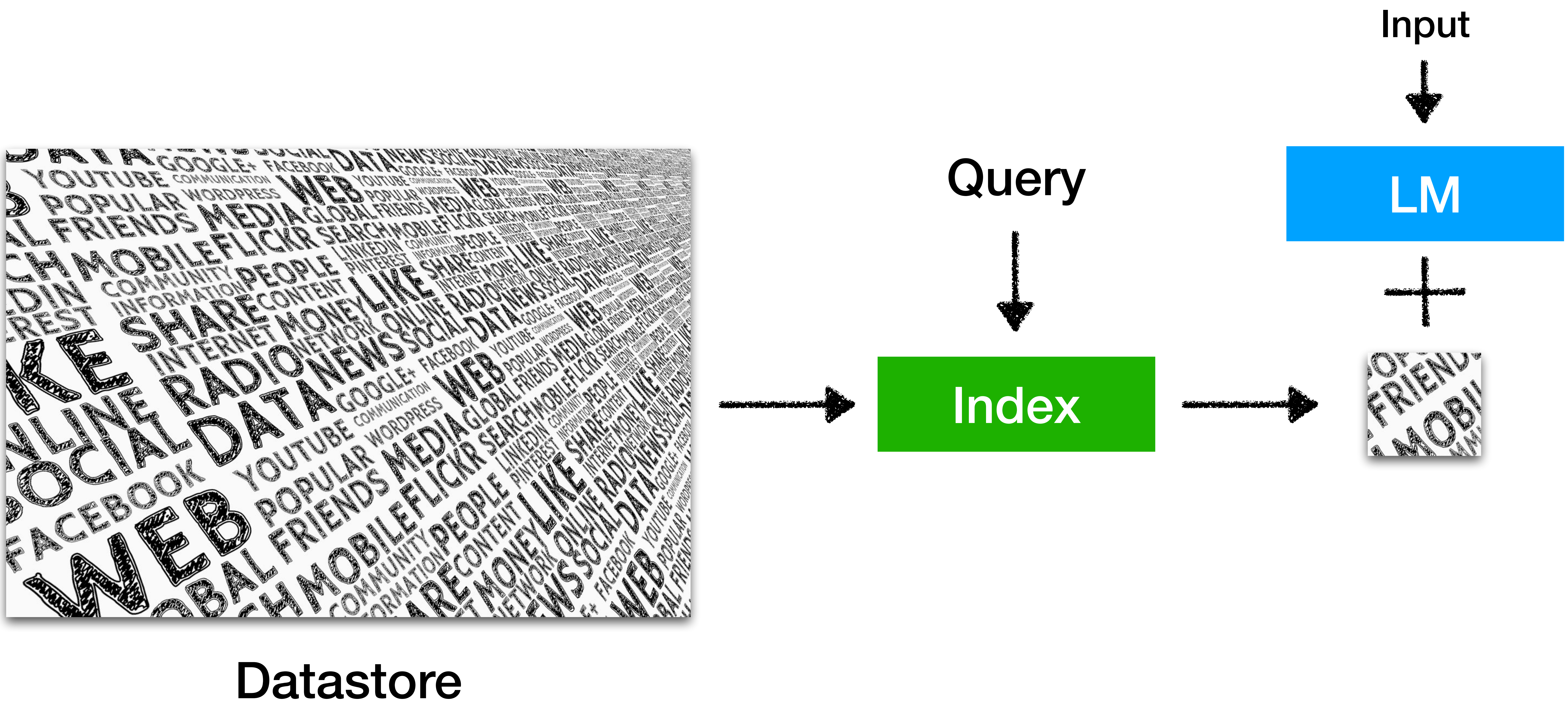


Inference: Incorporation

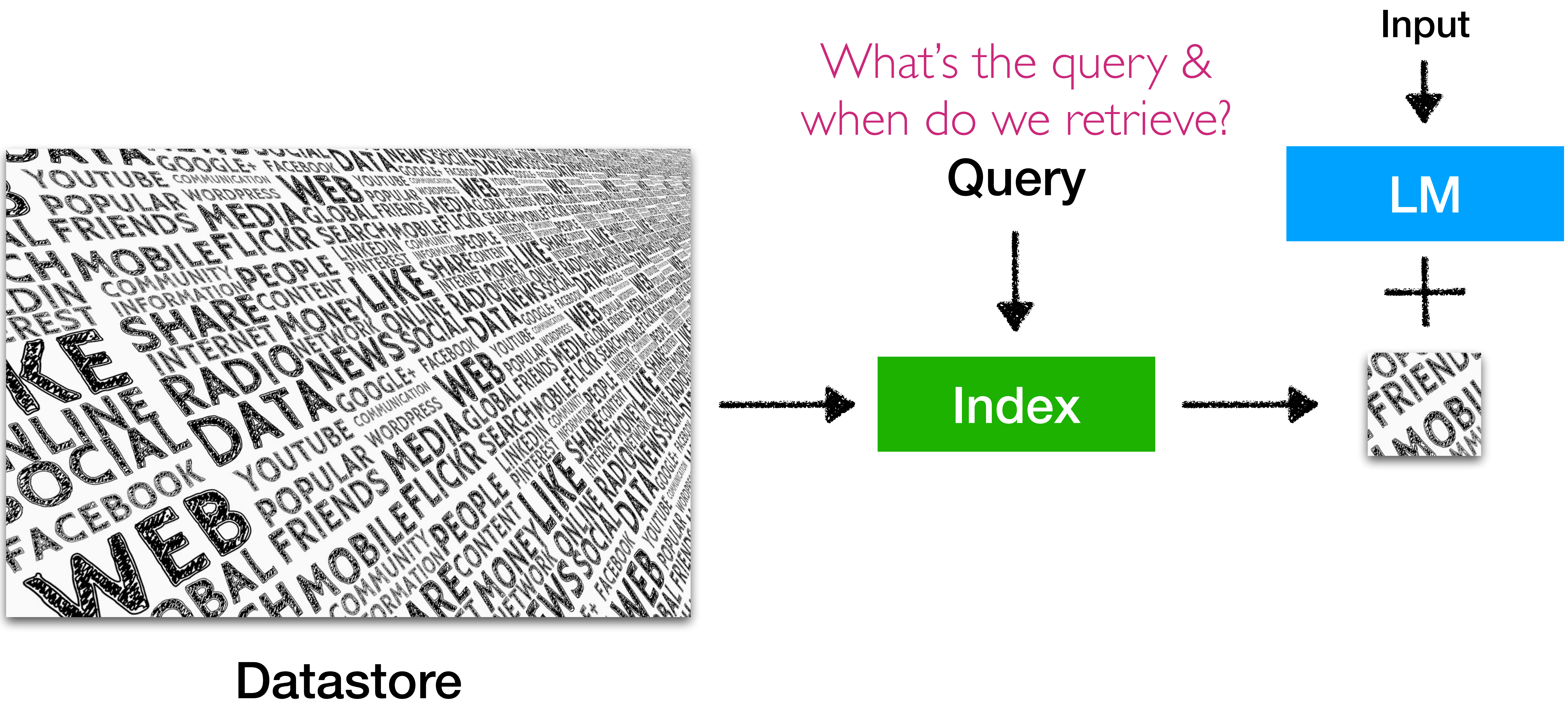


Datastore

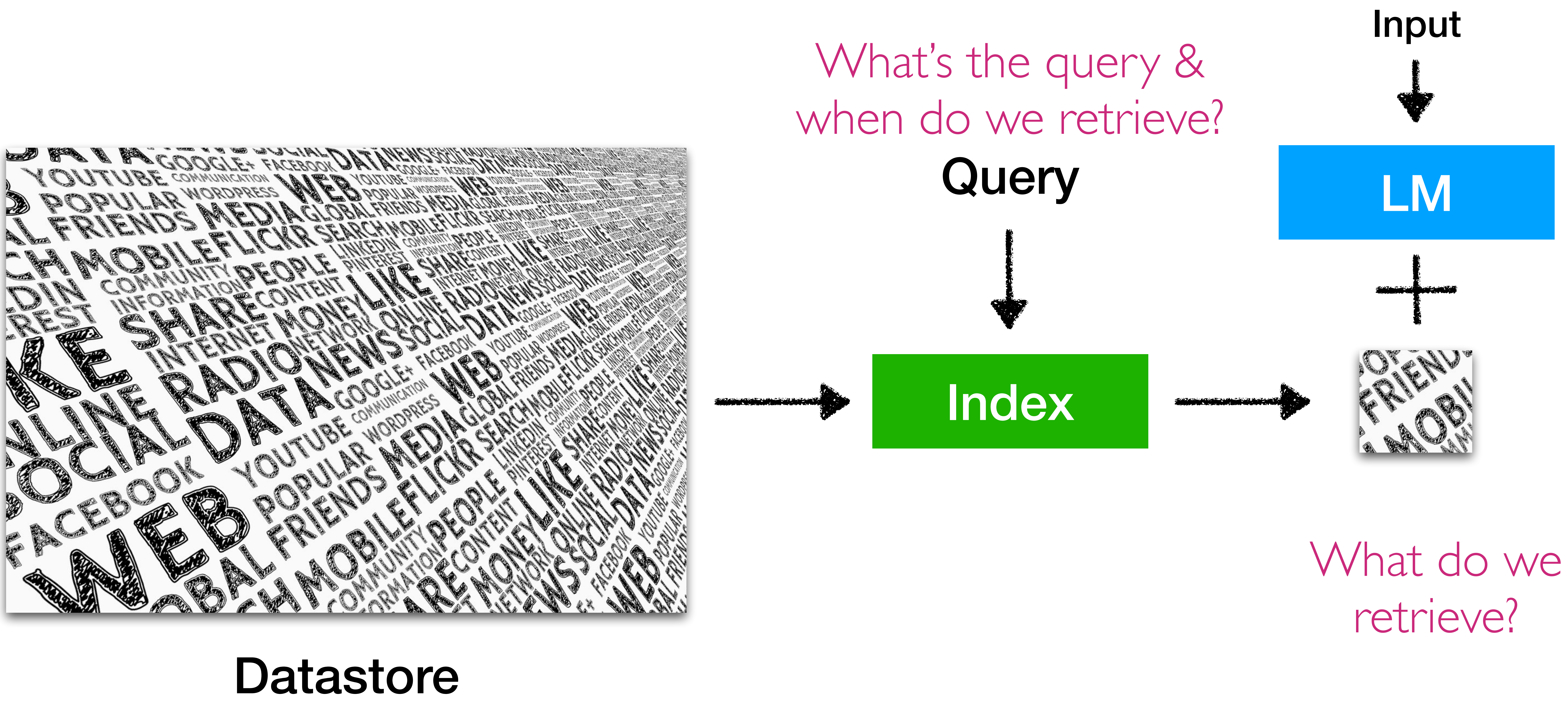
Questions to answer



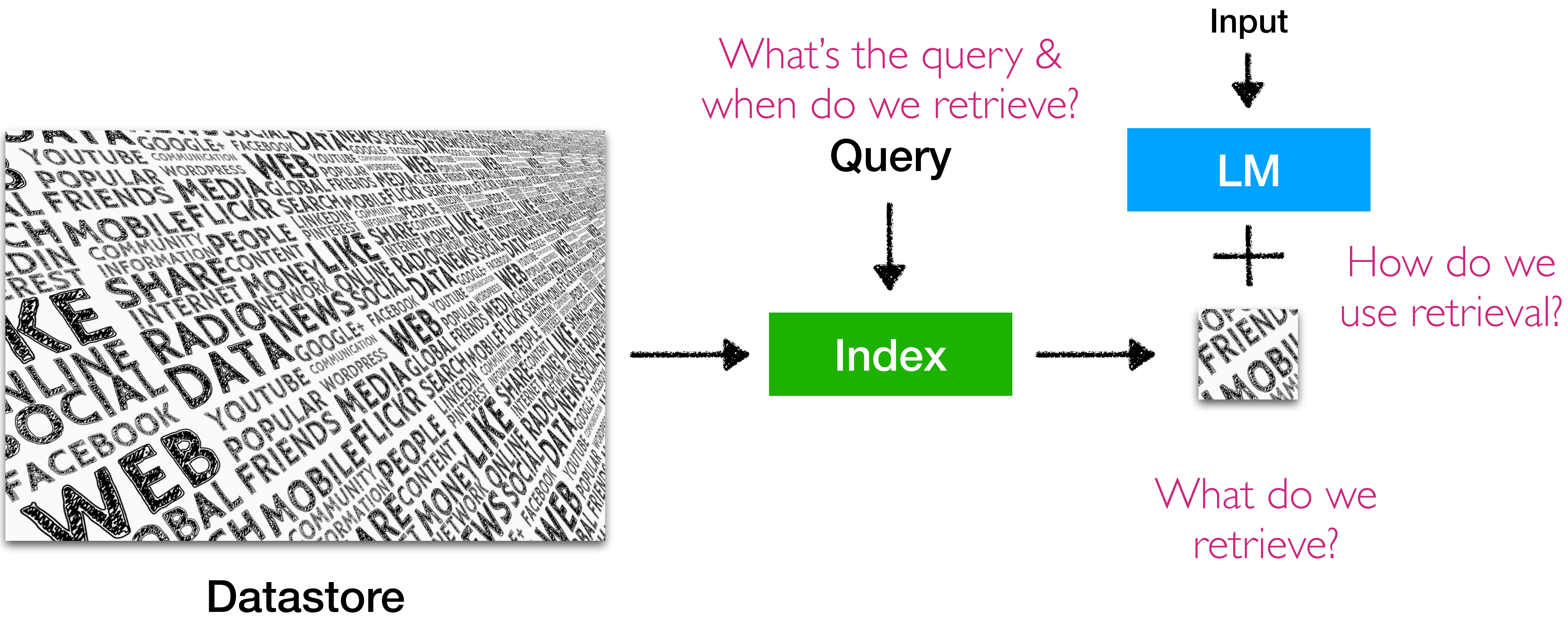
Questions to answer



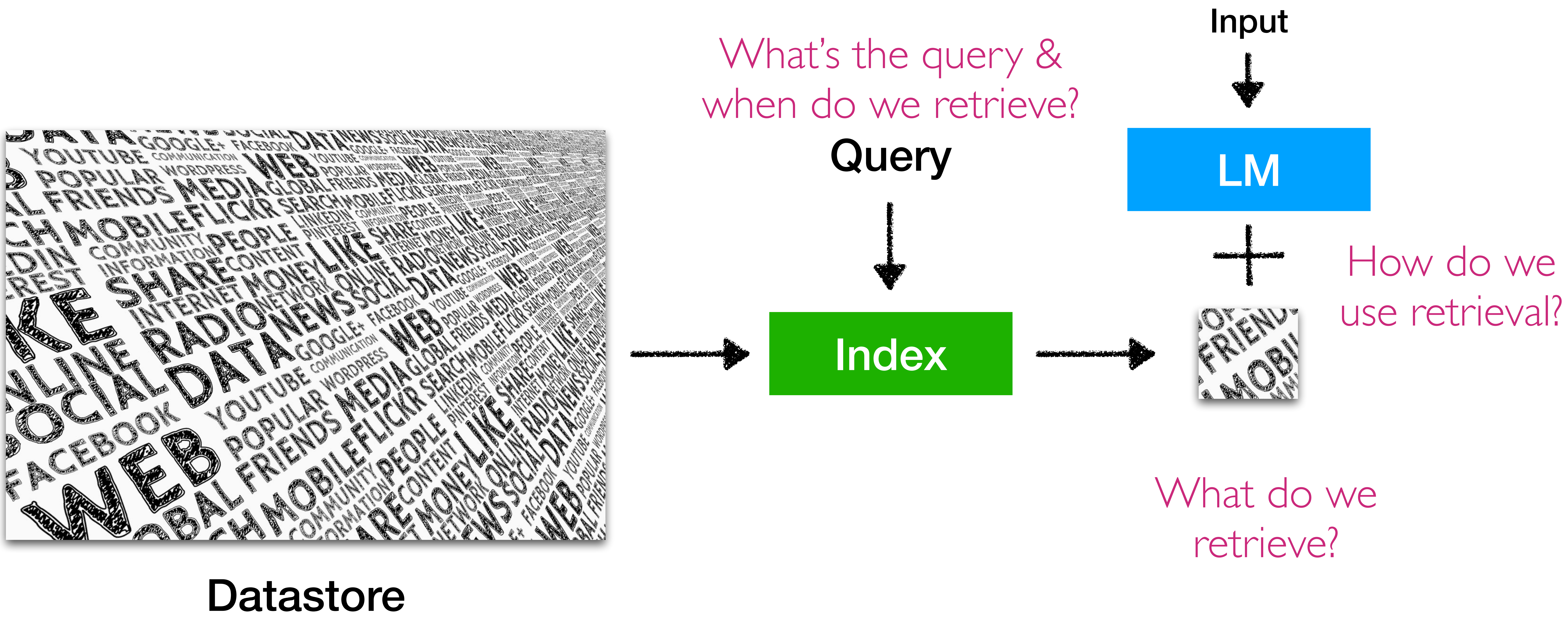
Questions to answer



Questions to answer



Questions to answer



We'll answer these questions in Section 3!

Notations