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)

\[ P(x_n | x_1, x_2, \ldots , x_{n-1}) \]

The capital city of Ontario is

\( x_1 \quad x_2 \quad \ldots \quad x_{n-1} \)
A language model (LM)

\[ P(x_n \mid x_1, x_2, \ldots, x_{n-1}) \]

The capital city of Ontario is

\[ x_1 \quad x_2 \quad \ldots \quad x_{n-1} \]
A language model (LM): Categories

Toronto

Autoregressive LM

The capital city of Ontario is ______
A language model (LM): Categories

**Autoregressive LM**

The capital city of Ontario is _____

**Masked LM**

The _____ city of ______ is Toronto
The capital city of Ontario is ______

The ______ city of _______ is Toronto

A language model (LM): Categories

The capital city of Ontario is ______

Autoregressive LM vs Masked LM

The ______ city of _______ is Toronto

Encoder-only

Decoder-only

A language model (LM): Categories

The capital city of Ontario is  ______

Autoregressive LM vs

The _____ city of _______ is  Toronto

Masked LM

Encoder-only

Decoder-only

Ontario

A language model (LM): Categories

Autoregressive LM vs Masked LM

The capital city of Ontario is ______

The ______ city of _______ is Toronto

Encoder-only & Decoder-only & Encoder-decoder

A language model (LM): Prompting
A language model (LM): Prompting

The capital city of Ontario is Toronto

Fact probing
A language model (LM): Prompting

The capital city of Ontario is Toronto

Cheaper than an iPod. It was great terrible

Fact probing Sentiment analysis
A language model (LM): Prompting

The capital city of Ontario is **Toronto**

Cheaper than an iPod. It was **great**

“Hello” in French is **Bonjour**
A language model (LM): Prompting

The capital city of Ontario is **Toronto**

Cheaper than an iPod. It was **great**

“Hello” in French is **Bonjour**

I’m good at math. $5 + 8 \times 12 = 101$
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

Perplexity

Toronto
Ottawa
Vancouver
Montreal
Calgary

... 0.52 0.31 0.13 0.03 0.01

\[-\log(0.52) = 0.284\]

Cheaper than an iPod. It was

LM

great
terrible

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**

Training time

The capital city of Ontario is _____

Test time
Retrieval-based LMs

The capital city of Ontario is Toronto

Training time

The capital city of Ontario is ____

Test time
Inference

Datastore

Input

Query

Index

LM
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

Datastore

Query

Index

LM

Input
Inference: Index

Retrieval input (not necessarily input to the LM)

Datastore

Query

Index

LM

Input
Inference: Index

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

Retrieval input (not necessarily input to the LM)
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

$\text{sim}$: a similarity score between two pieces of text
Goal: find a small subset of elements in a datastore that are the most similar to the query

$\textit{sim}$: a similarity score between two pieces of text

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

- $\text{tf}_{i,j}$: number of occurrences of $i$ in $j$
- $\text{df}_i$: number of documents containing $i$
- $N$: total number of documents
**Inference: Index**

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

\( \text{sim}: \) a similarity score between two pieces of text

**Example**

\[
\text{sim}(i, j) = \frac{tf_{i, j} \times \log N}{df_i} \quad \# \text{ of total docs}
\]

\[
\# \text{ of occurrences of } i \text{ in } j
\]

\[
\# \text{ of docs containing } i
\]

**Example**

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

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

\[
sim(i,j) = \frac{tf_{i,j} \times \log \frac{N}{df_i}}{\log \frac{N}{df_i}}
\]

- \(tf_{i,j}\): # of occurrences of \(i\) in \(j\)
- \(df_i\): # of docs containing \(i\)
- \(N\): # of total docs

Example

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

\[
\text{sim}: \text{a similarity score between two pieces of text}
\]

\[
\text{Index}: \text{given } q, \text{ return } \arg\text{Top-}k_{d \in \mathcal{D}}\text{sim}(q, d) \text{ through fast nearest neighbor search}
\]
Inference: Index

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

\( \text{sim} \): a similarity score between two pieces of text

**Index**: given \( q \), return \( \text{argTop-}k_{d \in D} \text{sim}(q, d) \) through fast nearest neighbor search over \( k \) elements from a datastore
Inference: Index

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

\( \text{sim} \): a similarity score between two pieces of text

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

\( k \) elements from a datastore

Can be a totally separate research area on how to do this fast & accurate
Software: FAISS, Distributed FAISS, SCaNN, etc...
Software: FAISS, Distributed FAISS, SCaNN, etc…

<table>
<thead>
<tr>
<th>Method</th>
<th>Class name</th>
<th>index_factory</th>
<th>Main parameters</th>
<th>Bytes/vector</th>
<th>Exhaustive</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Search for L2</td>
<td>IndexFlatL2</td>
<td>&quot;Flat&quot;</td>
<td>d</td>
<td>4d</td>
<td>yes</td>
<td>brute-force</td>
</tr>
<tr>
<td>Exact Search for Inner Product</td>
<td>IndexFlatIP</td>
<td>&quot;Flat&quot;</td>
<td>d</td>
<td>4d</td>
<td>yes</td>
<td>also for cosine (normalize vectors beforehand)</td>
</tr>
<tr>
<td>Hierarchical Navigable Small World graph exploration</td>
<td>IndexHNSWFlat</td>
<td>&quot;HNSW,Flat&quot;</td>
<td>d, M</td>
<td>4d + x * M * 2 * 4</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Inverted file with exact post-verification</td>
<td>IndexIVFFlat</td>
<td>&quot;IVFx,Flat&quot;</td>
<td>quantizer, d, nlist, metric</td>
<td>4d + 8</td>
<td>no</td>
<td>Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.</td>
</tr>
<tr>
<td>Locality-Sensitive Hashing (binary flat index)</td>
<td>IndexLSH</td>
<td>-</td>
<td>d, nbits</td>
<td>ceil(nbits/8)</td>
<td>yes</td>
<td>optimized by using random rotation instead of random projections</td>
</tr>
<tr>
<td>Scalar quantizer (SQ) in flat mode</td>
<td>IndexScalarQuantizer</td>
<td>&quot;SQ8&quot;</td>
<td>d</td>
<td>d</td>
<td>yes</td>
<td>4 and 6 bits per component are also implemented.</td>
</tr>
<tr>
<td>Product quantizer (PQ) in flat mode</td>
<td>IndexPQ</td>
<td>&quot;PQc&quot;, &quot;PQ/M&quot;*&quot;nbits</td>
<td>d, M, nbits</td>
<td>ceil(M + nbits / 8)</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>IVF and scalar quantizer</td>
<td>IndexIVFScalarQuantizer</td>
<td>&quot;IVFx,SQ8&quot;, &quot;IVFx,SQ8&quot;</td>
<td>quantizer, d, nlist, sype</td>
<td>SQfp16: 2 * d + 8, SQ8: d + 8 or SQ4: d/2 + 8</td>
<td>no</td>
<td>Same as the IndexScalarQuantizer</td>
</tr>
<tr>
<td>IVFADC (coarse quantizer+PQ on residuals)</td>
<td>IndexIVFFPQ</td>
<td>&quot;IVFx,PQ&quot;y&quot;x&quot;*nbits</td>
<td>quantizer, d, nlist, M, nbits</td>
<td>ceil(M + nbits/8)-8</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>IVFADC+R (same as IVFADC with re-ranking based on codes)</td>
<td>IndexIVFFQR</td>
<td>&quot;IVFx,PQy+z&quot;</td>
<td>quantizer, d, nlist, M, nbits, M_refine, nbits_refine</td>
<td>M+M_refine+8</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Class name</td>
<td><code>index_factory</code></td>
<td>Main parameters</td>
<td>Bytes/vector</td>
<td>Exhaustive</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>-------------------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>--------------</td>
<td>------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Exact Search for L2</td>
<td>IndexFlatL2</td>
<td>&quot;Flat&quot;</td>
<td>d</td>
<td>4+d</td>
<td>yes</td>
<td>brute-force</td>
</tr>
<tr>
<td>Exact Search for Inner Product</td>
<td>IndexFlatIP</td>
<td>&quot;Flat&quot;</td>
<td>d</td>
<td>4+d</td>
<td>yes</td>
<td>also for cosine (normalize vectors beforehand)</td>
</tr>
<tr>
<td>Hierarchical Navigable Small World graph exploration</td>
<td>IndexHNSFlat3</td>
<td>&quot;HNSM,Flat&quot;</td>
<td>d, M</td>
<td>4*d + x * M * 2 * 4</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Inverted file with exact post-verification</td>
<td>IndexIVFFlat</td>
<td>&quot;IVFx,Flat&quot;</td>
<td>quantizer, d, nlits, metric</td>
<td>4*d + 8</td>
<td>no</td>
<td>Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.</td>
</tr>
<tr>
<td>Locality-Sensitive Hashing (binary flat index)</td>
<td>IndexLSH</td>
<td>-</td>
<td>d, nbits</td>
<td>ceil(nbites/8)</td>
<td>yes</td>
<td>optimized by using random rotation instead of random projections</td>
</tr>
<tr>
<td>Scalar quantizer (SQ) in flat mode</td>
<td>IndexScalarQuantizer</td>
<td>&quot;SQ8&quot;</td>
<td>d</td>
<td>d</td>
<td>yes</td>
<td>4 and 6 bits per component are also implemented.</td>
</tr>
<tr>
<td>Product quantizer (PQ) in flat mode</td>
<td>IndexPQ</td>
<td>&quot;PQc&quot;, &quot;PQc<em>M&quot;</em>&quot;nbnts</td>
<td>d, M, nbits</td>
<td>ceil(M + nbits / 8)</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>IVF and scalar quantizer</td>
<td>IndexIVFScalarQuantizer</td>
<td>&quot;IVFx,SO4*&quot;</td>
<td>quantizer, d, nlits, ptype</td>
<td>SQfp16: 2 * d + 8, SQ8: d + 8 or SQ4: d/2 + 8</td>
<td>no</td>
<td>Same as the IndexScalarQuantizer</td>
</tr>
<tr>
<td>IVFADC (coarse quantizer+PQ on residuals)</td>
<td>IndexIVFFPQ</td>
<td>&quot;IVFx,PQ+y&quot;*&quot;nbnts</td>
<td>quantizer, d, nlits, M, nbits</td>
<td>ceil(M + nbits/8)+8</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>IVFADC+R (same as IVFADC with re-ranking based on codes)</td>
<td>IndexIVFFQR</td>
<td>&quot;IVFx,PQ+y+z&quot;*</td>
<td>quantizer, d, nlits, M, nbits, M_refine, nbits_refine</td>
<td>M*M_refine+8</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
Software: FAISS, Distributed FAISS, SCaNN, etc...

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

**Exact Search** (Relatively easy to scale to ~1B elements)
### Software: FAISS, Distributed FAISS, SCaNN, etc…

<table>
<thead>
<tr>
<th>Method</th>
<th>Class name</th>
<th>index_factory</th>
<th>Main parameters</th>
<th>Bytes/vector</th>
<th>Exhaustive</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Search for L2</td>
<td>IndexFlatL2</td>
<td>&quot;Flat&quot;</td>
<td>d</td>
<td>4+d</td>
<td>yes</td>
<td>brute-force</td>
</tr>
<tr>
<td>Exact Search for Inner Product</td>
<td>IndexFlatIP</td>
<td>&quot;Flat&quot;</td>
<td>d</td>
<td>4+d</td>
<td>yes</td>
<td>also for cosine (normalize vectors beforehand)</td>
</tr>
<tr>
<td>Hierarchical Navigable Small World graph exploration</td>
<td>IndexHNSWFlat</td>
<td>&quot;HNSW,Flat&quot;</td>
<td>d, M</td>
<td>4<em>d + x + M = 2</em>4</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Inverted file with exact post-verification</td>
<td>IndexIVFFlat</td>
<td>&quot;IVFF,Flat&quot;</td>
<td>quantizer, d, nlist, metric</td>
<td>4*d + 8</td>
<td>no</td>
<td>Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.</td>
</tr>
<tr>
<td>Locality-Sensitive Hashing (binary flat index)</td>
<td>IndexLSH</td>
<td>-</td>
<td>d, nbits</td>
<td>ceil(nbits/8)</td>
<td>yes</td>
<td>optimized by using random rotation instead of random projections</td>
</tr>
<tr>
<td>Scalar quantizer (SQ) in flat mode</td>
<td>IndexScalarQuantizer</td>
<td>&quot;SQ8&quot;</td>
<td>d</td>
<td>d</td>
<td>yes</td>
<td>4 and 6 bits per component are also implemented.</td>
</tr>
<tr>
<td>Product quantizer (PQ) in flat mode</td>
<td>IndexPQ</td>
<td>&quot;PQ&quot;,&quot;PV^M&quot; x nbits</td>
<td>d, M, nbits</td>
<td>ceil(M + nbits / 8)</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>iV and scalar quantizer</td>
<td>IndexIVScalarQuantizer</td>
<td>&quot;iVFF,SOQ&quot;,&quot;iVFF,SOQ&quot;</td>
<td>quantizer, d, nlist, yplayer</td>
<td>SQp16: 2 * d + 8, SQ8: d + 8 or SQ4: d/2 + 8</td>
<td>no</td>
<td>Same as the IndexScalarQuantizer</td>
</tr>
<tr>
<td>iVFADC (coarse quantizer+PQ on residuals)</td>
<td>IndexIVFPQ</td>
<td>&quot;IVFPQ,&quot;y&quot;x&quot; x nbits</td>
<td>quantizer, d, nlist, M, nbits</td>
<td>ceil(M + nbits/8)+6</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>iVFADC+R (same as iVFADC with re-ranking based on codes)</td>
<td>IndexIVFPQR</td>
<td>&quot;IVFPQ&quot;,&quot;y&quot;+2&quot;</td>
<td>quantizer, d, nlist, M, nbits, M_refine, nbits_refine</td>
<td>M + M_refine + 8</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Class name</td>
<td>index_factory</td>
<td>Main parameters</td>
<td>Bytes/vector</td>
<td>Exhaustive</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>--------------------------</td>
<td>----------------</td>
<td>-----------------</td>
<td>--------------</td>
<td>------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Exact Search for L2</td>
<td>IndexFlatL2</td>
<td>&quot;Flat&quot;</td>
<td>d</td>
<td>4*d</td>
<td>yes</td>
<td>brute-force</td>
</tr>
<tr>
<td>Exact Search for Inner Product</td>
<td>IndexFlatIP</td>
<td>&quot;Flat&quot;</td>
<td>d</td>
<td>4*d</td>
<td>yes</td>
<td>also for cosine (normalize vectors beforehand)</td>
</tr>
<tr>
<td>Hierarchical Navigable Small World graph exploration</td>
<td>IndexHNSWFlat</td>
<td>&quot;HNSW,Flat&quot;</td>
<td>d, M</td>
<td>4<em>d + x</em>M = 2*x / 4</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Inverted file with exact post-verification</td>
<td>IndexIVFFlat</td>
<td>&quot;IVFx,Flat&quot;</td>
<td>quantizer, d, nlist, metric</td>
<td>4*d + 8</td>
<td>no</td>
<td>Takes another index to assign vectors to inverted lists. The 8 additional bytes are the vector id that needs to be stored.</td>
</tr>
<tr>
<td>Locality-Sensitive Hashing (binary flat index)</td>
<td>IndexLSH</td>
<td>-</td>
<td>d, nbits</td>
<td>cell(nbits/8)</td>
<td>yes</td>
<td>optimized by using random rotation instead of random projections</td>
</tr>
<tr>
<td>Scalar quantizer (SQ) in flat mode</td>
<td>IndexScalarQuantizer</td>
<td>&quot;SQ8&quot;</td>
<td>d</td>
<td>d</td>
<td>yes</td>
<td>4 and 6 bits per component are also implemented.</td>
</tr>
<tr>
<td>Product quantizer (PQ) in flat mode</td>
<td>IndexPQ</td>
<td>&quot;PQ8&quot;, &quot;PQ7*M&quot;*nbits</td>
<td>d, M, nbits</td>
<td>ceil(M + nbits / 8)</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>iV F and scalar quantizer</td>
<td>IndexIVFScalarQuantizer</td>
<td>&quot;IVFx,SQ4&quot;, &quot;IVFx,SQ8&quot;</td>
<td>quantizer, d, nlist, metric</td>
<td>SQfp16: 2*d + 8, SQ8: d + 8 or SQ4: d/2 + 8</td>
<td>no</td>
<td>Same as the IndexScalarQuantizer</td>
</tr>
<tr>
<td>iVFADC (coarse quantizer+PQ on residuals)</td>
<td>IndexIVFFPQ</td>
<td>&quot;IVFx,PQ*y&quot;*nbits</td>
<td>quantizer, d, nlist, M, nbits</td>
<td>cell(M + nbits/8) + 8</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>iVFADC+R (same as iVFADC with re-ranking based on codes)</td>
<td>IndexIVFFQR</td>
<td>&quot;IVFx,PQ*y+2&quot;*M</td>
<td>quantizer, d, nlist, M, nbits, M.refine, nbits_refine, M.M_refine8</td>
<td>M*M.refine8</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

**Software:** FAISS, Distributed FAISS, SCaNN, etc...

**Exact Search**

**Approximate Search**

(Relatively easy to scale to ~1B elements)

In this tutorial, we assume we can do it fast & accurate.
In this tutorial, we assume we can do it fast & accurate.
Inference: Incorporation

Datastore

Query

Index

LM

Input
Questions to answer

Datastore

Query

Index

LM

Input
Questions to answer

What's the query & when do we retrieve?

Input

Query

LM

Index

Datastore
Questions to answer

What’s the query & when do we retrieve?

Input

LM

What do we retrieve?

Query

Index

Datastore
Questions to answer

What’s the query & when do we retrieve?

How do we use retrieval?

What do we retrieve?

Input

Query

Index

LM

Datastore
Questions to answer

What’s the query & when do we retrieve?
What do we retrieve?
How do we use retrieval?

We’ll answer these questions in Section 3!
Notations

Datastore $\mathcal{D}$

Query $q$

LM

Input $x$ → Output $y$