Vector Databases

Tyler Foster

Machine Learning Seminar

Department of Scientific Computing, FSU

September 28, 2023

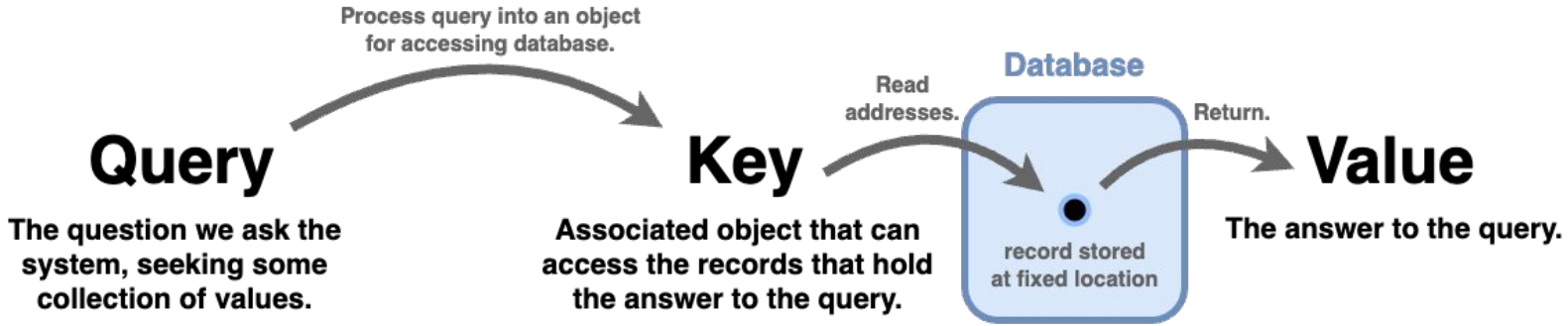
Databases.

Databases.

A diagrammatic picture of what happens when you interact with an abstract database:

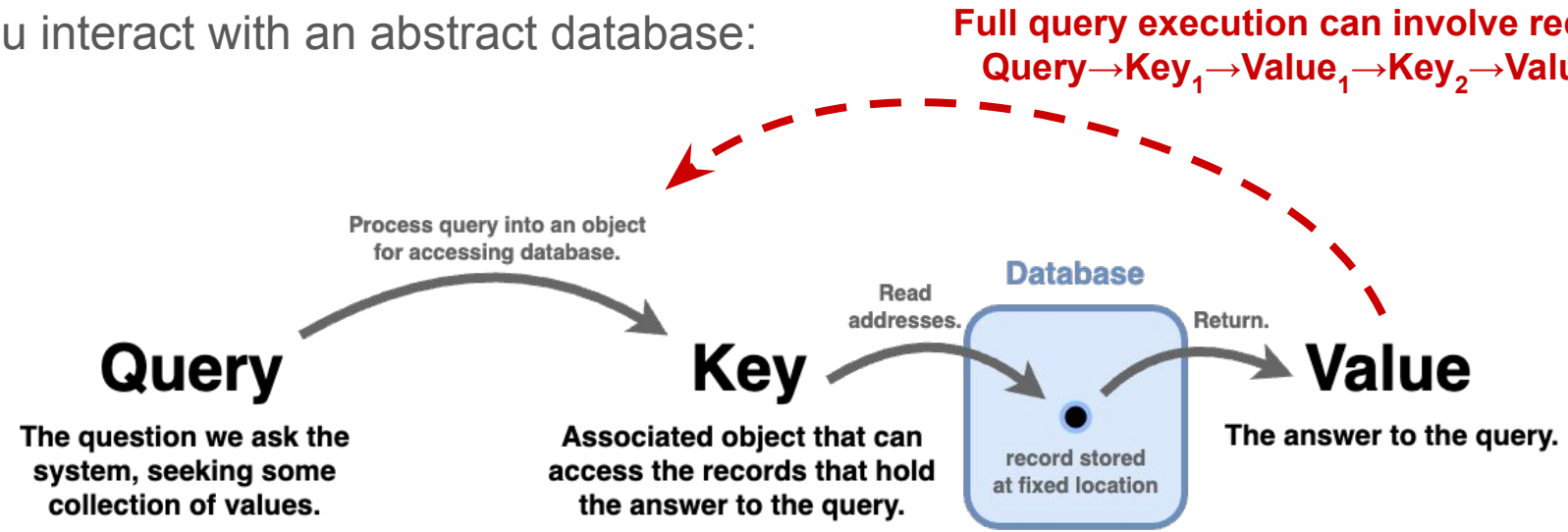
Databases.

A diagrammatic picture of what happens when you interact with an abstract database:



Databases.

A diagrammatic picture of what happens when you interact with an abstract database:



Example 1.

Example 1.

Python dictionary:

Example 1.

Python dictionary:

```
[1]:  dict = {'A':1, 'B':2, 'C':3}

[2]:  dict['B']

[2]:  2
```

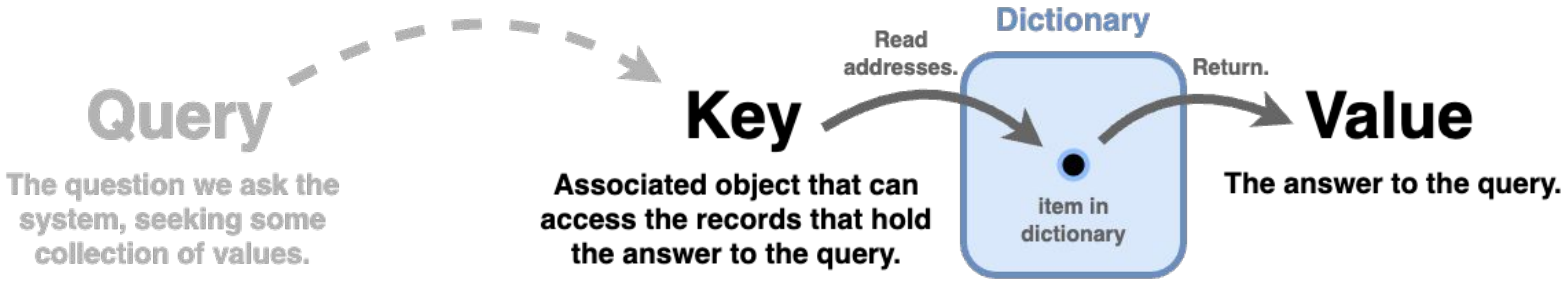
Example 1.

Python dictionary:

```
[1]:  dict = {'A':1, 'B':2, 'C':3}

[2]:  dict['B']

[2]:  2
```



Example 2.

Example 2.

SQL database:

Example 2.

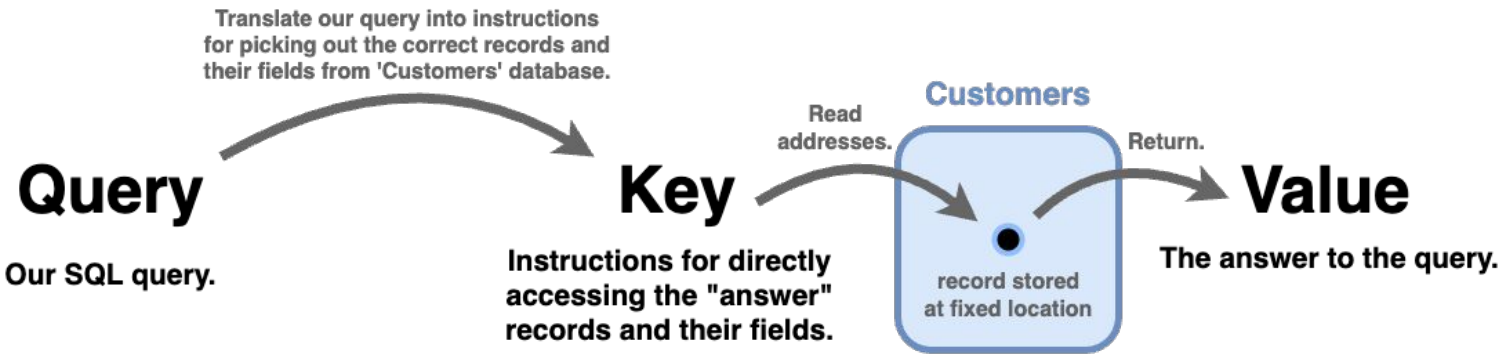
SQL database:

```
SELECT first_name, age
FROM Customers
WHERE last_name LIKE 'Q%' OR last_name LIKE 'R%';
```

Example 2.

SQL database:

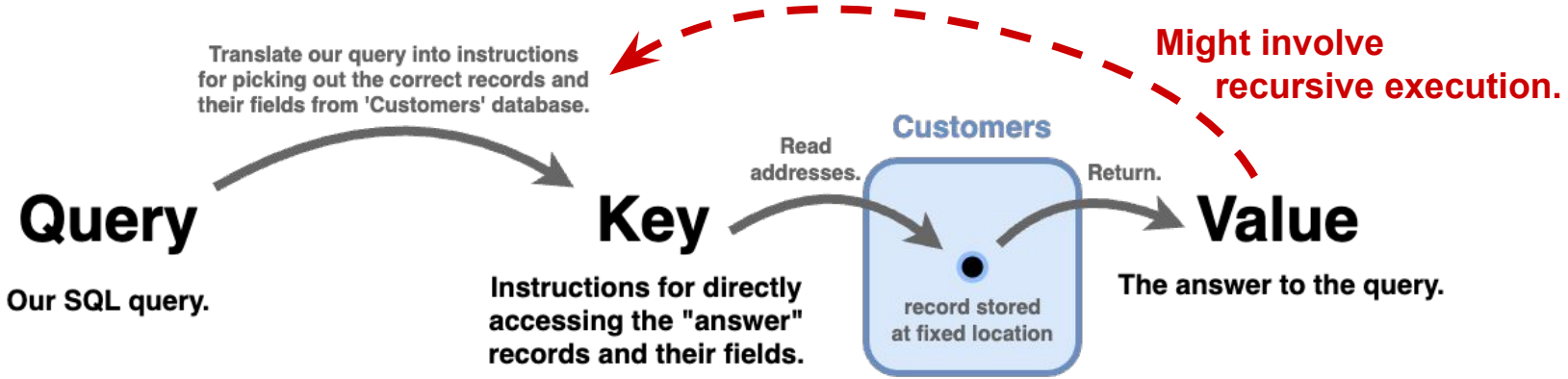
```
SELECT first_name, age
FROM Customers
WHERE last_name LIKE 'Q%' OR last_name LIKE 'R%';
```



Example 2.

SQL database:

```
SELECT first_name, age
FROM Customers
WHERE last_name LIKE 'Q%' OR last_name LIKE 'R%';
```



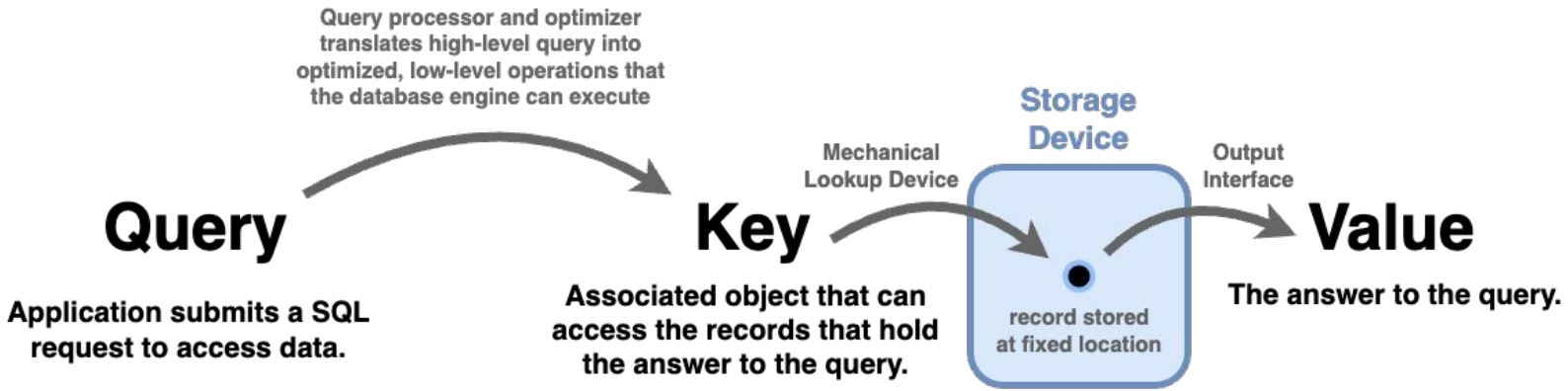
Example 3.

Example 3.

Physical DataBase Management System (DBMS):

Example 3.

Physical DataBase Management System (DBMS):



Observation.

Observation.

SQL-like queries are formed from a small set of operators:

Observation.

SQL-like queries are formed from a small set of operators:

- Logical operators: AND, OR, LIKE, ALL, etc.

Observation.

SQL-like queries are formed from a small set of operators:

- Logical operators: AND, OR, LIKE, ALL, etc.
- Numerical operators: +, -, >, etc.

Observation.

SQL-like queries are formed from a small set of operators:

- Logical operators: AND, OR, LIKE, ALL, etc.
- Numerical operators: +, -, >, etc.
- Set theoretical operators: UNION, INTERSECT, etc.

Observation.

SQL-like queries are formed from a small set of operators:

- Logical operators: AND, OR, LIKE, ALL, etc.
- Numerical operators: +, -, >, etc.
- Set theoretical operators: UNION, INTERSECT, etc.

Processing a SQL-like query amounts to parsing composite statements formed from these operators.

Observation.

SQL-like queries are formed from a small set of operators:

- Logical operators: AND, OR, LIKE, ALL, etc.
- Numerical operators: +, -, >, etc.
- Set theoretical operators: UNION, INTERSECT, etc.

Processing a SQL-like query amounts to parsing composite statements formed from these operators.

…But how well can we really describe the world with this language?

Problem.

Problem.

Think about the kinds of queries we make constantly as humans existing in reality.

Problem.

Think about the kinds of queries we make constantly as humans existing in reality.

Problem.

Think about the kinds of queries we make constantly as humans existing in reality.

The SQL query format and its specific query-to-key processing cannot support “realistic” querying.

High-level idea for a solution.

High-level idea for a solution.

Database records should include extra fields that encode more of the real-life-meanings attached to records.

High-level idea for a solution.

Database records should include extra fields that encode more of the real-life-meanings attached to records.

AND

High-level idea for a solution.

Database records should include extra fields that encode more of the real-life-meanings attached to records.

AND

Query-to-key processing and the key's address-reads should utilize the extra fields to enable more realistic querying.

“Pre-” Vector Database Example.

“Pre-” Vector Database Example.

Geospatial database:

“Pre-” Vector Database Example.

Geospatial database:

Records in a geospatial database are some variation on

“Pre-” Vector Database Example.

Geospatial database:

Records in a geospatial database are some variation on

	property 1	property 2	…	property *n*	position
record	$value_1$	$value_2$	…	$value_n$	**r**

“Pre-” Vector Database Example.

Geospatial database:

Records in a geospatial database are some variation on

	property 1	property 2	···	property *n*	position
record	$value_1$	$value_2$	···	$value_n$	**r**

Position vector **r** in $\mathbf{R}^3$ describing location on Earth

“Pre-” Vector Database Example.

Geospatial database:

Records in a geospatial database are some variation on

	property 1	property 2	···	property *n*	position
record	$value_1$	$value_2$	···	$value_n$	**r**

Position vector **r** in $\mathbf{R}^3$ describing location on Earth

Any part of query processing or address reading that Involves linear algebra in $\mathbf{R}^3$ (including navigation via $\mathbf{SO}(3) \subset \mathbf{GL}(3)$) has fast implementation on GPUs.

“Pre-” Vector Database Example.

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

Not real SQL operators!
Geospatial meaning only.

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

How query execution might work here:

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

How query execution might work here:

1. `position.predict_location(15:30:00)` accesses some public repository of wind currents (a *tensor field* **F** in $\mathbf{R}^3$).

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

How query execution might work here:

1. `position.predict_location(15:30:00)` accesses some public repository of wind currents (a *tensor field* **F** in $\mathbf{R}^3$). For the present position **r** of each aloft balloon, compute the time evolution of **r** in **F** to get expected position **p** of the balloon at 3:30 PM.

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

How query execution might work here:

1. `position.predict_location(15:30:00)` accesses some public repository of wind currents (a *tensor field* **F** in $\mathbf{R}^3$). For the present position **r** of each aloft balloon, compute the time evolution of **r** in **F** to get expected position **p** of the balloon at 3:30 PM. **(A linear-algebraic computation on GPUs!)**

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

How query execution might work here:

1. `position.predict_location(15:30:00)` accesses some public repository of wind currents (a *tensor field* **F** in $\mathbf{R}^3$). For the present position **r** of each aloft balloon, compute the time evolution of **r** in **F** to get expected position **p** of the balloon at 3:30 PM. **(A linear-algebraic computation on GPUs!)**
2. The `NEAR 'Paris'` operator checks if the expected position **p** is near the position of Paris.

“Pre-” Vector Database Example.

```
SELECT balloon_ID
FROM Balloons
WHERE balloon_status = aloft AND position.predict_location(15:30:00) NEAR 'Paris';
```

How query execution might work here:

1. `position.predict_location(15:30:00)` accesses some public repository of wind currents (a *tensor field* **F** in $\mathbf{R}^3$). For the present position **r** of each aloft balloon, compute the time evolution of **r** in **F** to get expected position **p** of the balloon at 3:30 PM. **(A linear-algebraic computation on GPUs!)**
2. The `NEAR 'Paris'` operator checks if the expected position **p** is near the position of Paris. **(A linear-algebraic computation on GPUs!)**

“Pre-” Vector Database Example.

“Pre-” Vector Database Example.

We’ve used the geometric interpretation of linear-algebraic computations to

“Pre-” Vector Database Example.

We’ve used the geometric interpretation of linear-algebraic computations to endow our query

“Pre-” Vector Database Example.

We’ve used the geometric interpretation of linear-algebraic computations to endow our query, our query processor

“Pre-” Vector Database Example.

We’ve used the geometric interpretation of linear-algebraic computations to endow our query, our query processor, and our address reader

“Pre-” Vector Database Example.

We’ve used the geometric interpretation of linear-algebraic computations to endow our query, our query processor, and our address reader with the ability execute queries that are more physically realistic.

RECALL:

High-level idea for a solution.

Database records should include extra fields that encode more of the real-life-meanings attached to records.

AND

Query-to-key processing and the key's address-reads should utilize the extra fields to enable more realistic querying.

More-or-less obvious to ML Engineers:

More-or-less obvious to ML Engineers:

Why not use a good *universal encoder* to produce encodings of either the record itself or data associated to the record?

More-or-less obvious to ML Engineers:

Why not use a good *universal encoder* to produce encodings of either the record itself or data associated to the record (PDFs, BMPs, etc.)?

More-or-less obvious to ML Engineers:

Why not use a good *universal encoder* to produce encodings of either the record itself or data associated to the record (PDFs, BMPs, etc.)?

The proposed extra fields in any record become the vector's embeddings.

More-or-less obvious to ML Engineers:

Why not use a good *universal encoder* to produce encodings of either the record itself or data associated to the record (PDFs, BMPs, etc.)?

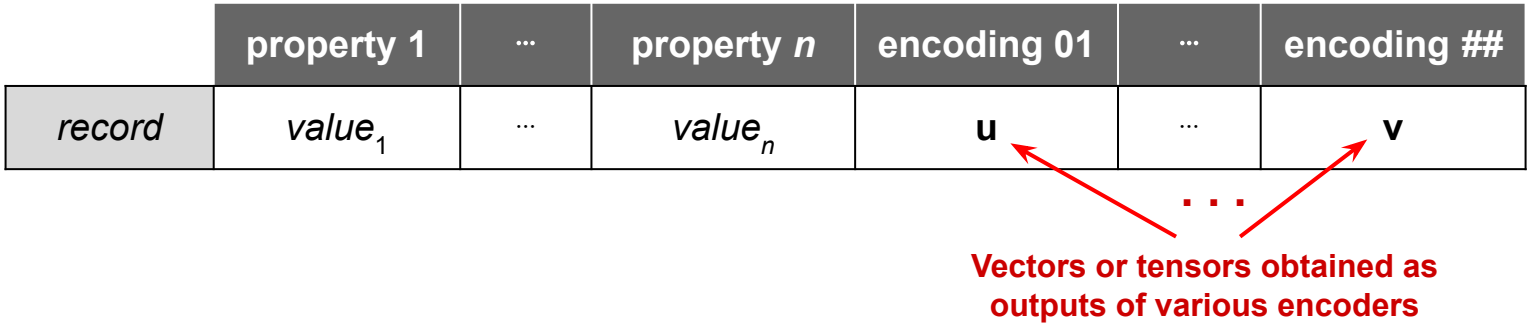
The proposed extra fields in any record become the vector's embeddings.

	property 1	···	property *n*	encoding 01	···	encoding ##
record	$value_1$	···	$value_n$	**u**	···	**v**

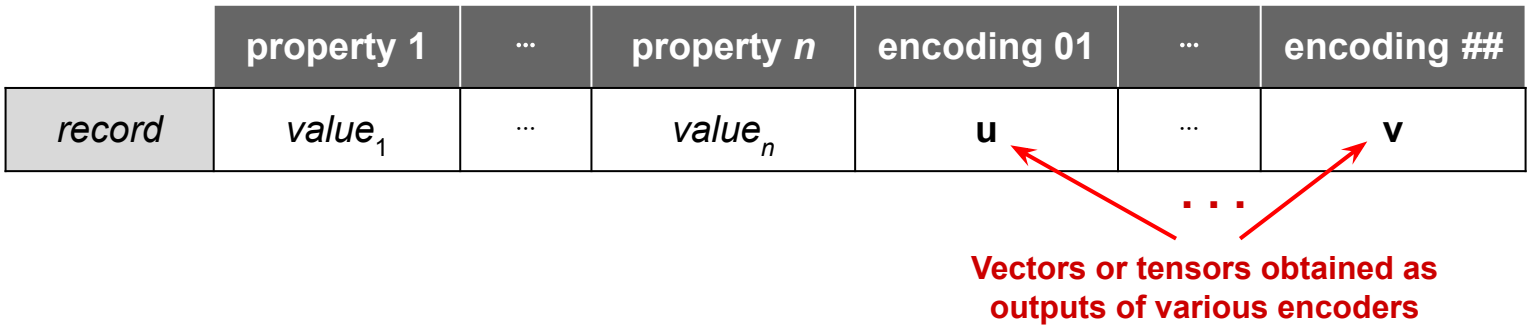
More-or-less obvious to ML Engineers:

Why not use a good *universal encoder* to produce encodings of either the record itself or data associated to the record (PDFs, BMPs, etc.)?

The proposed extra fields in any record become the vector's embeddings.

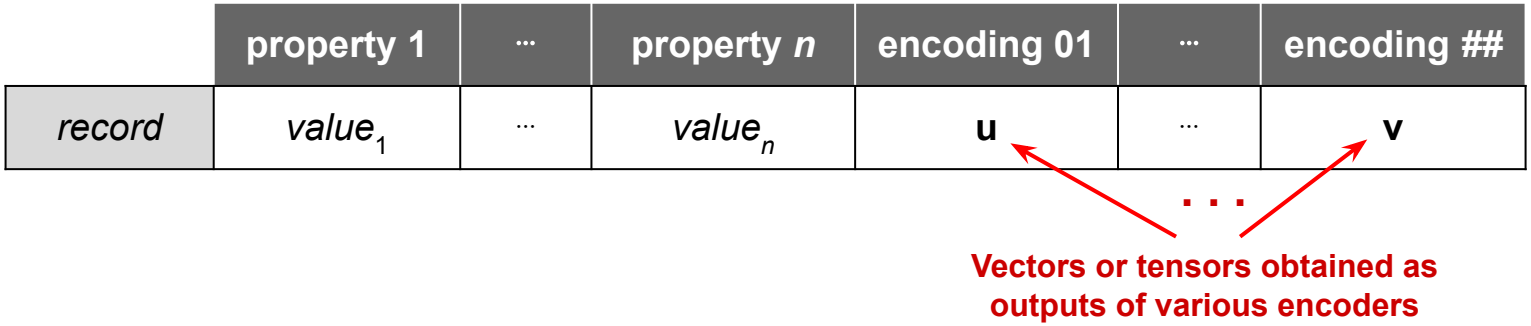


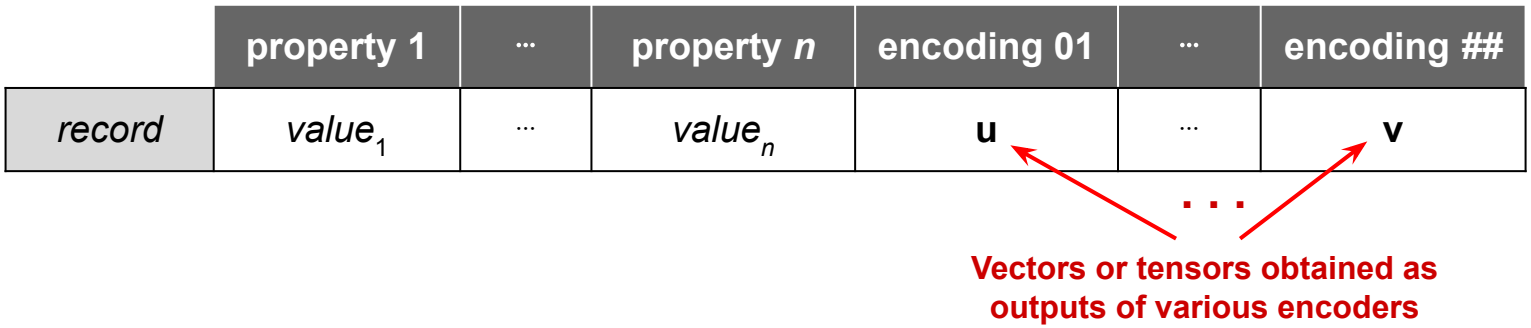
More-or-less obvious to ML Engineers:



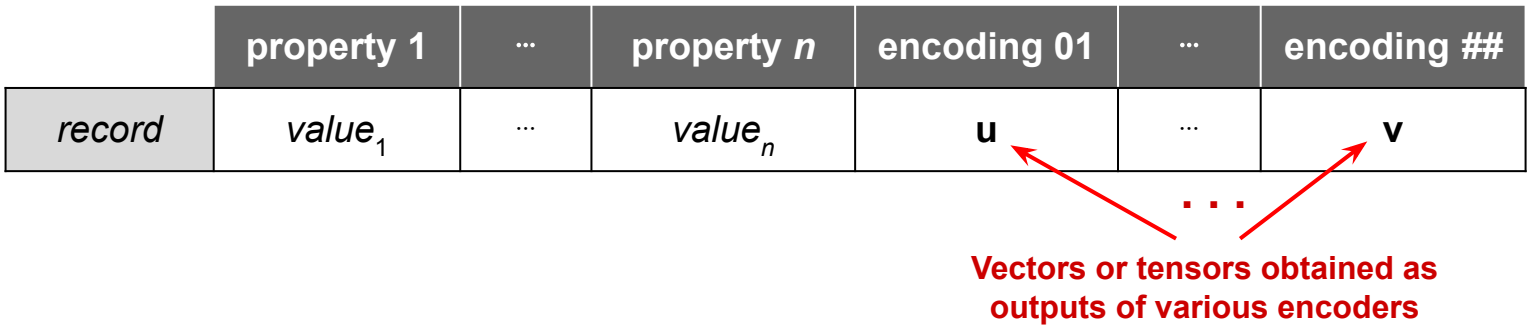
More-or-less obvious to ML Engineers:

Any trainable PyTorch model with “tensor → tensor” forward pass is a reasonable candidate for an operator we might use as part of executing queries in a database with records that look like this.



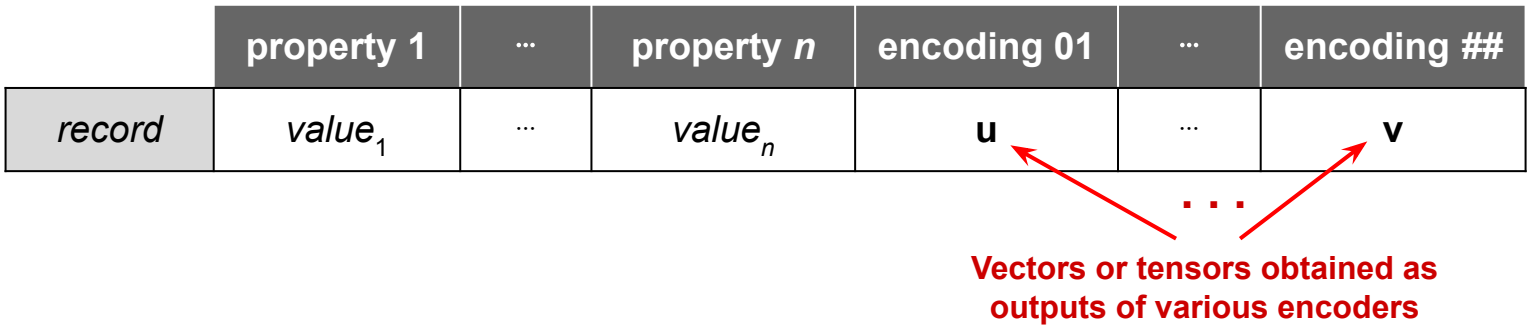

property 1
...
property n
encoding 01
...
encoding ##
record
value1
...
valuen
u
...
v
. . .
Vectors or tensors obtained as
outputs of various encoders

A ***Vector Database*** is a Database Management System (DBMS) where:



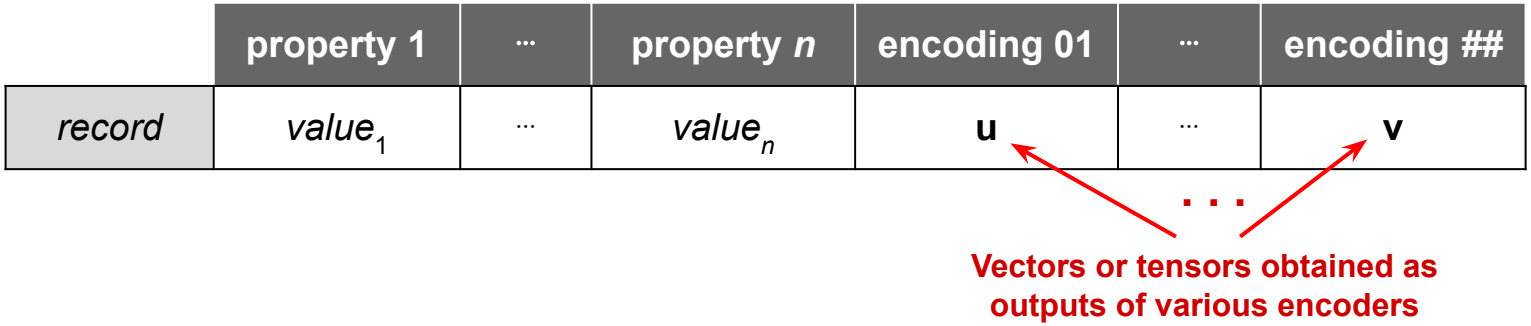
A ***Vector Database*** is a Database Management System (DBMS) where:

1. Stored records include columns whose values are vectors/tensors.



A ***Vector Database*** is a Database Management System (DBMS) where:

1. Stored records include columns whose values are vectors/tensors.
2. Query execution exploits modern tensor-based computational tools (GPUs, ML, etc.) when working with vectors/tensors in these columns.



Time/good place for Jupyter notebook example?

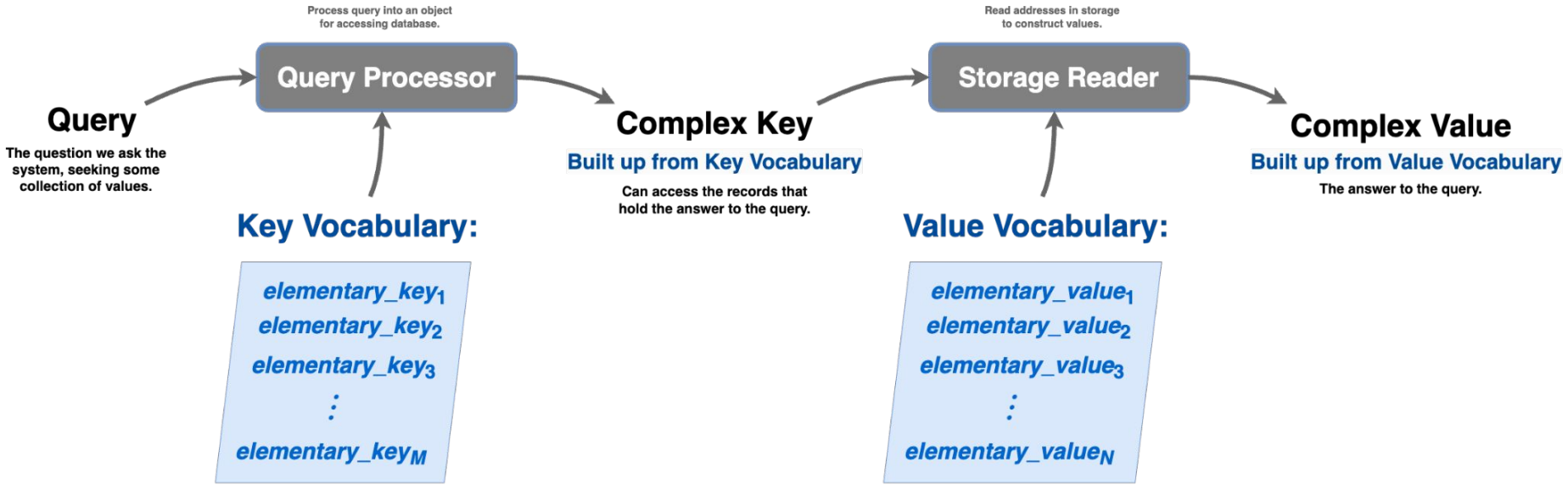
Relationship to Transformers.

Relationship to Transformers.

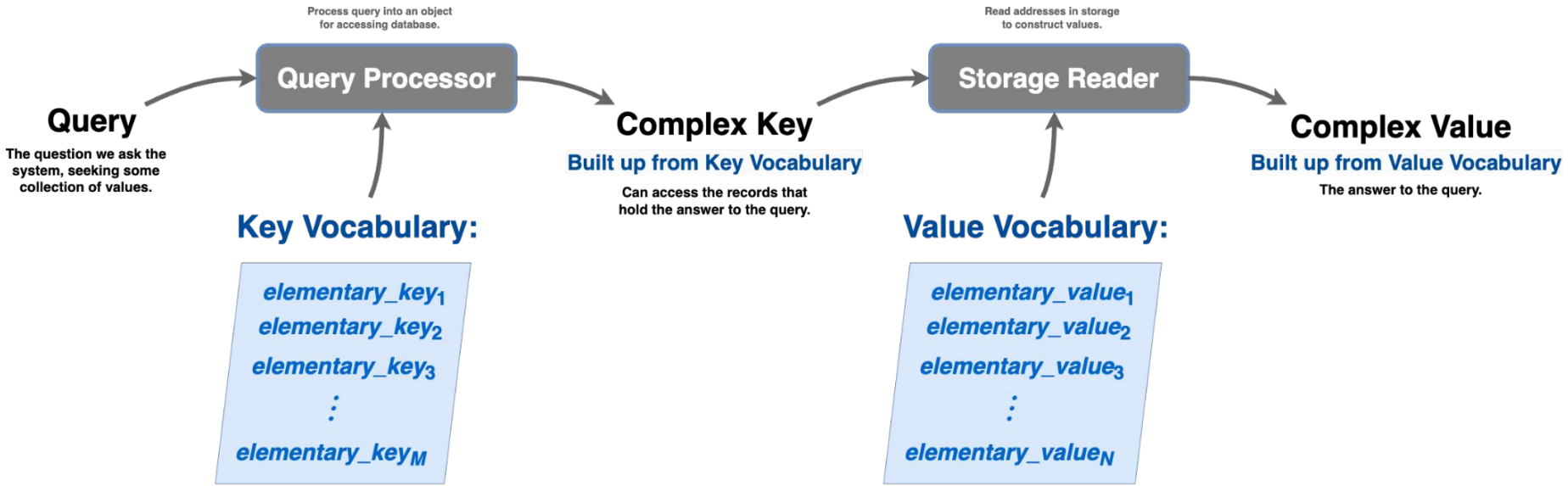
We can add some details to our abstract picture of a DBMS without hurting its level of abstraction too much:

Relationship to Transformers.

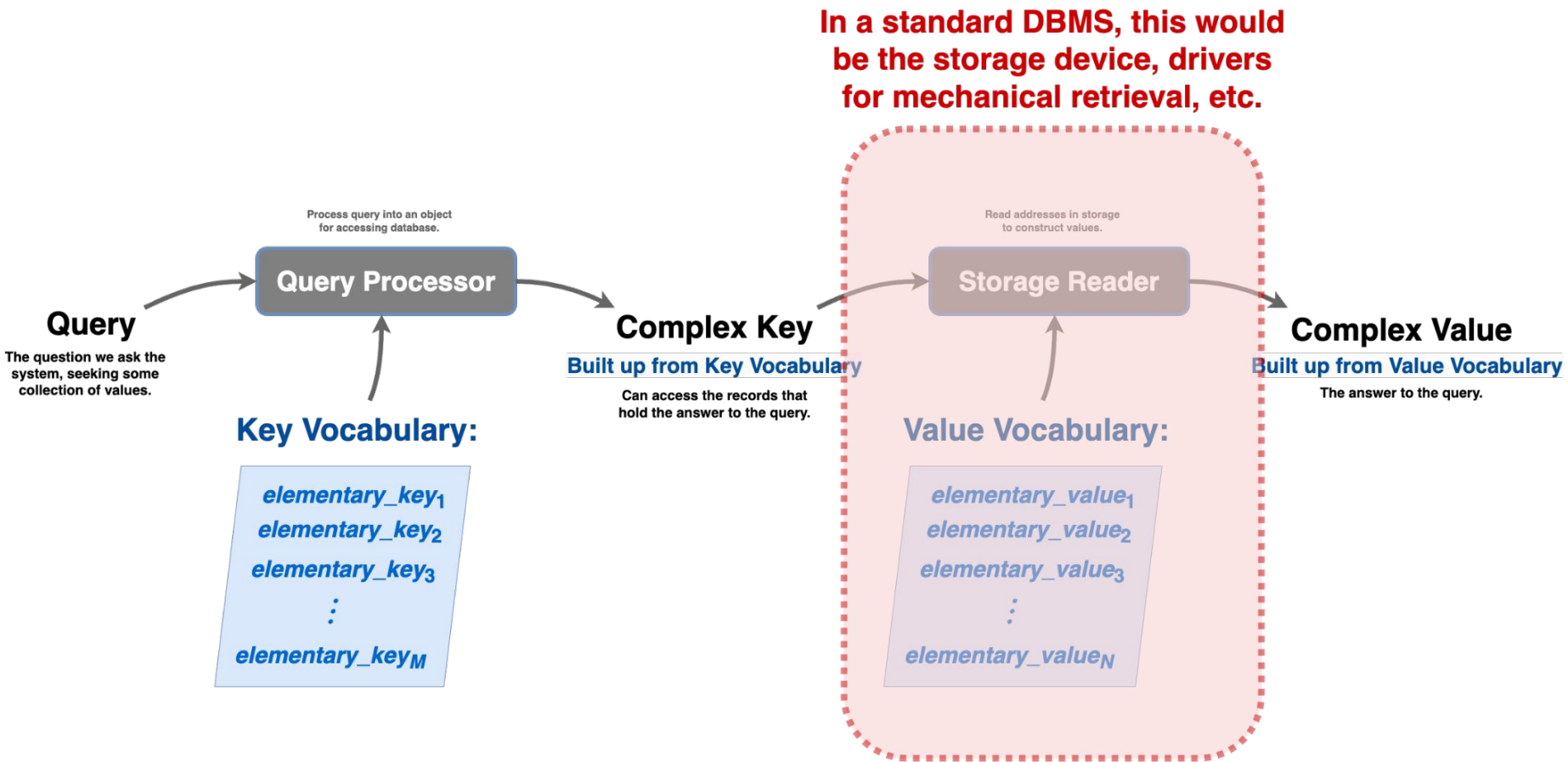
We can add some details to our abstract picture of a DBMS without hurting its level of abstraction too much:



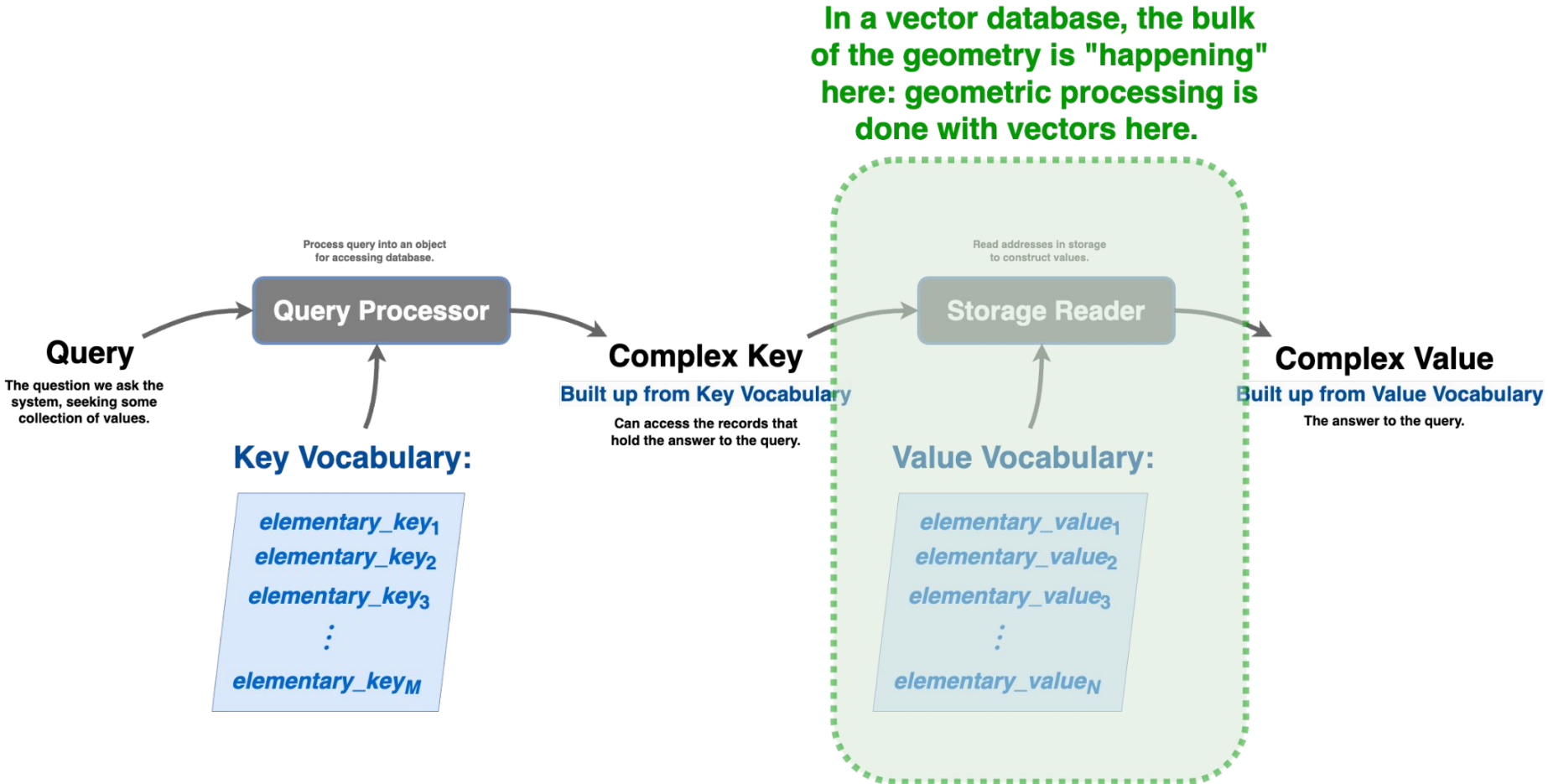
Relationship to Transformers.



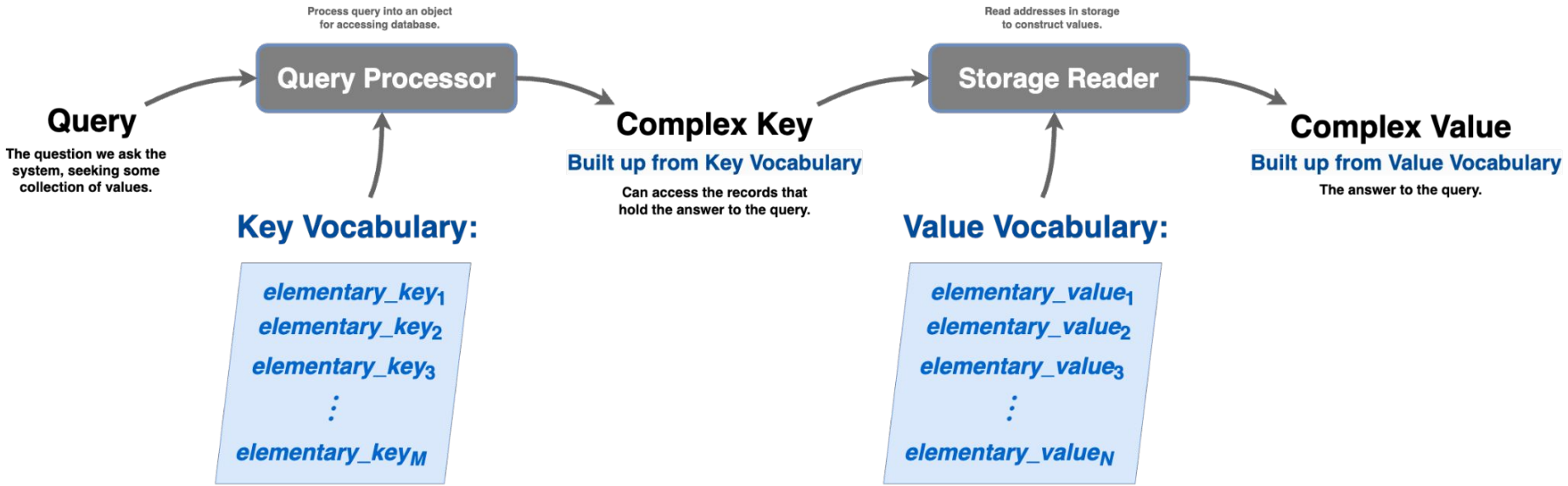
Relationship to Transformers.



Relationship to Transformers.

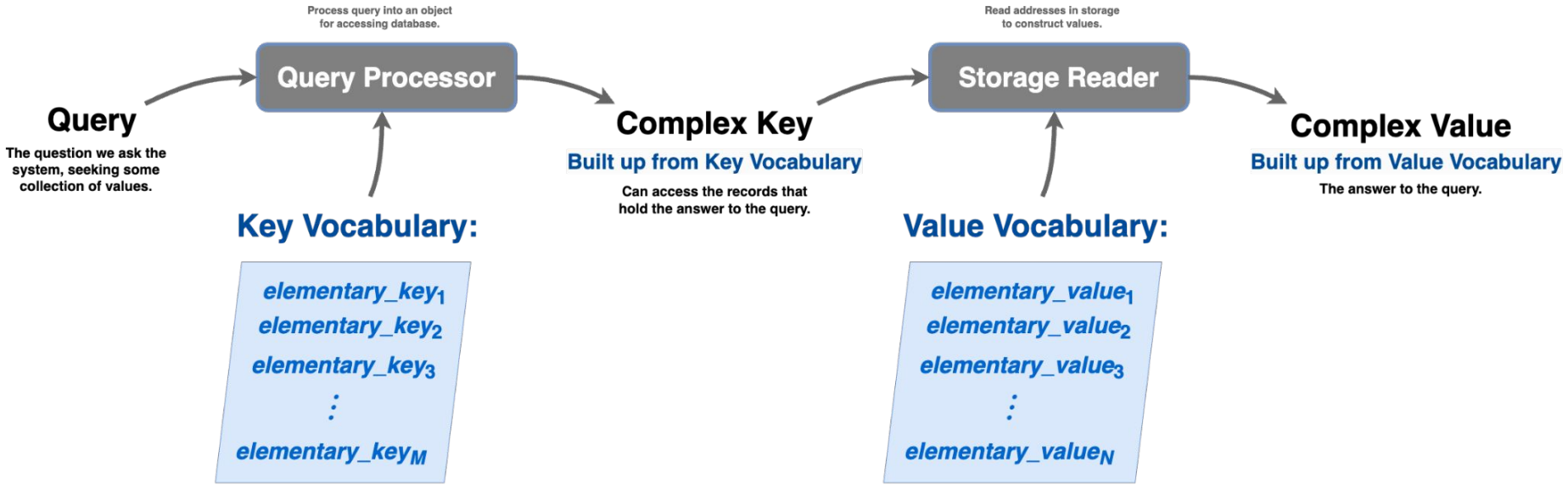


Relationship to Transformers.



Relationship to Transformers.

The cross-attention head in a transformer also follows this pattern.

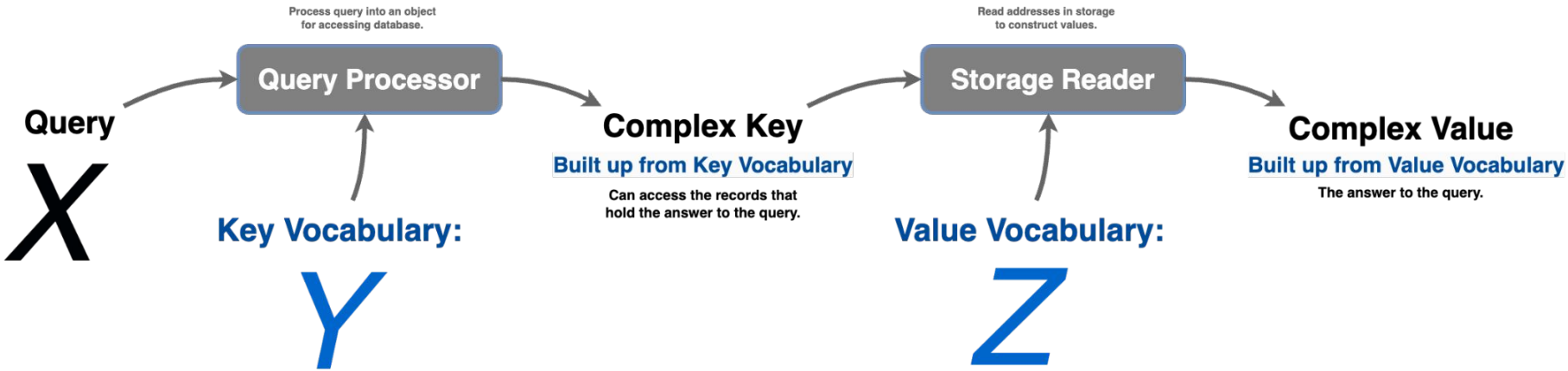


Relationship to Transformers.

The cross-attention head in a transformer also follows this pattern.

Relationship to Transformers.

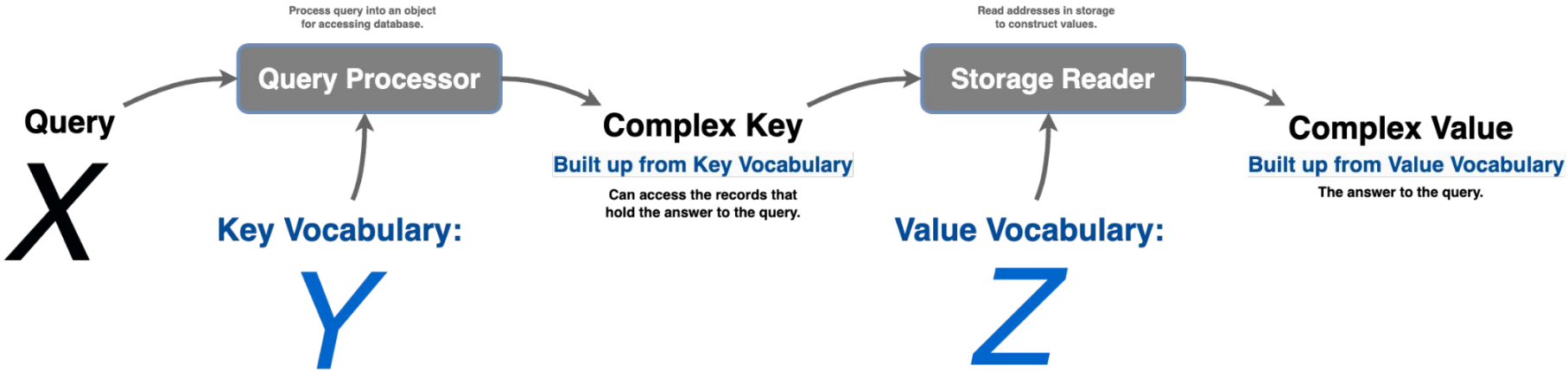
The cross-attention head in a transformer also follows this pattern.



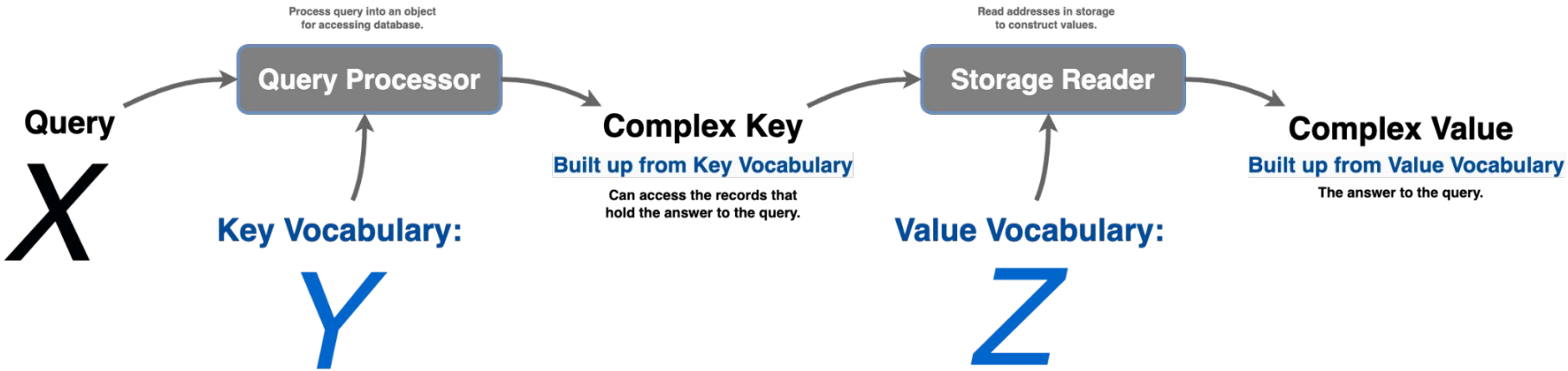
Relationship to Transformers.

The cross-attention head in a transformer also follows this pattern.

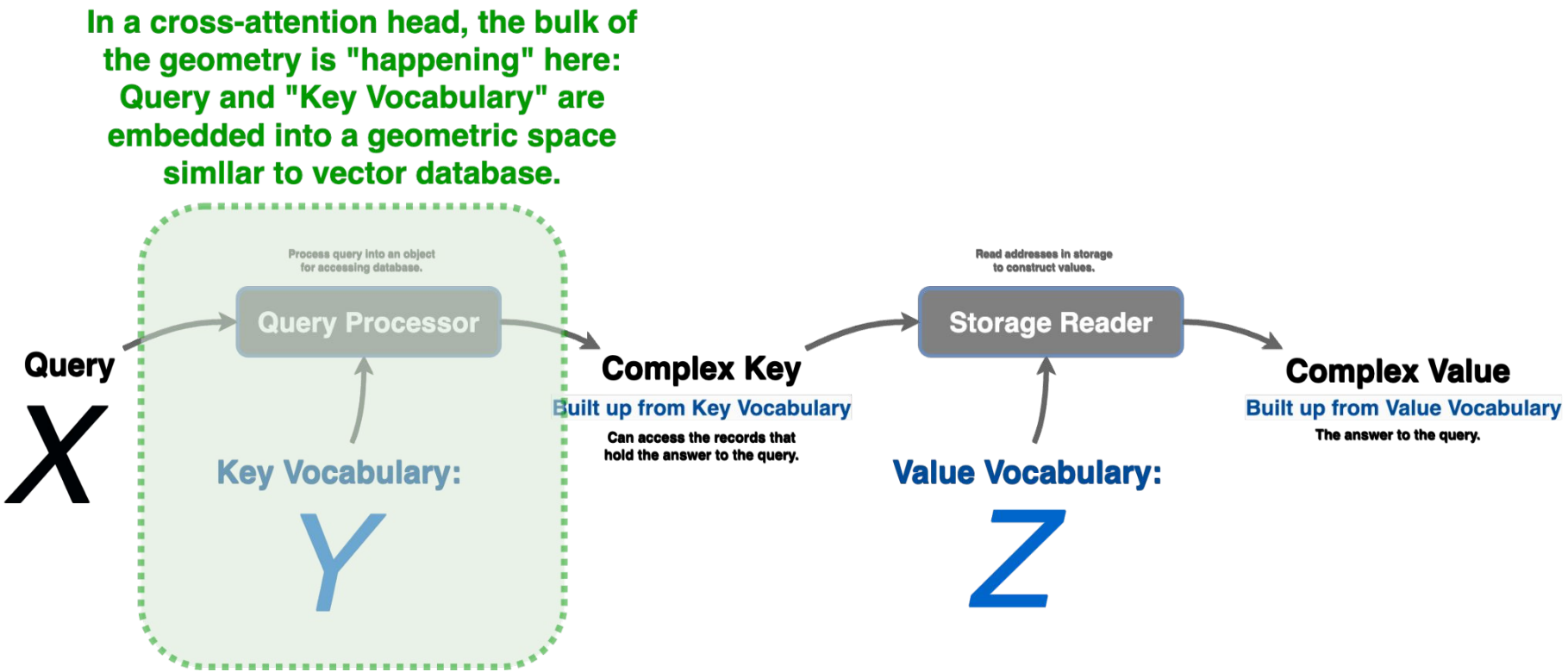
A cross-attention head is like a DBMS ***so robust*** that you can load it with any (correctly formatted) key and value vocabulary you like, and it will be able to do "closest match" query execution:



Relationship to Transformers.



Relationship to Transformers.



Speeding up searches in a vector database.

Speeding up searches in a vector database.

Even with GPUs, it’s not efficient to check records one-by-one when the database executes a query.

Speeding up searches in a vector database.

Even with GPUs, it’s not efficient to check records one-by-one when the database executes a query.

How can we execute queries more efficiently in a vector database?

Product Quantization (PQ).

Product Quantization (PQ).

Product quantization is a compression technique (requiring an associated search procedure) that can speed up query execution time.

Product Quantization (PQ).

Product quantization is a compression technique (requiring an associated search procedure) that can speed up query execution time.

In a vector database, we compress the vectors in each record via “quantization”:

Product Quantization (PQ).

Product quantization is a compression technique (requiring an associated search procedure) that can speed up query execution time.

In a vector database, we compress the vectors in each record via “quantization”:

$$\text{continuous space } \mathbf{R}^d \quad \longmapsto \quad \text{finite set } \{C_1, C_2, \ldots, C_k\}$$

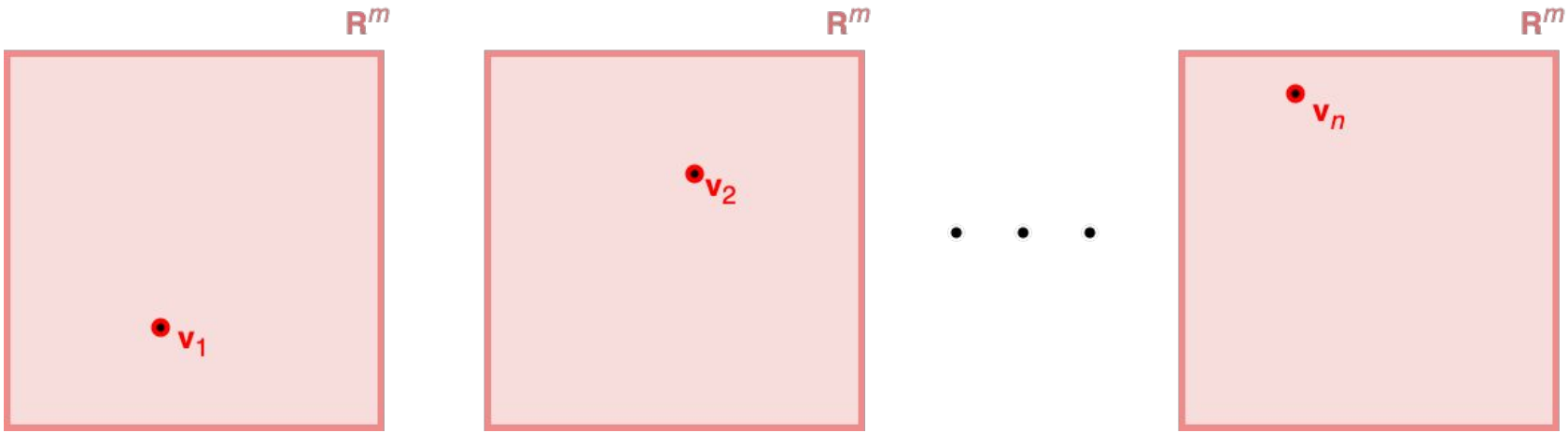
Product Quantization (PQ).

Product Quantization (PQ).

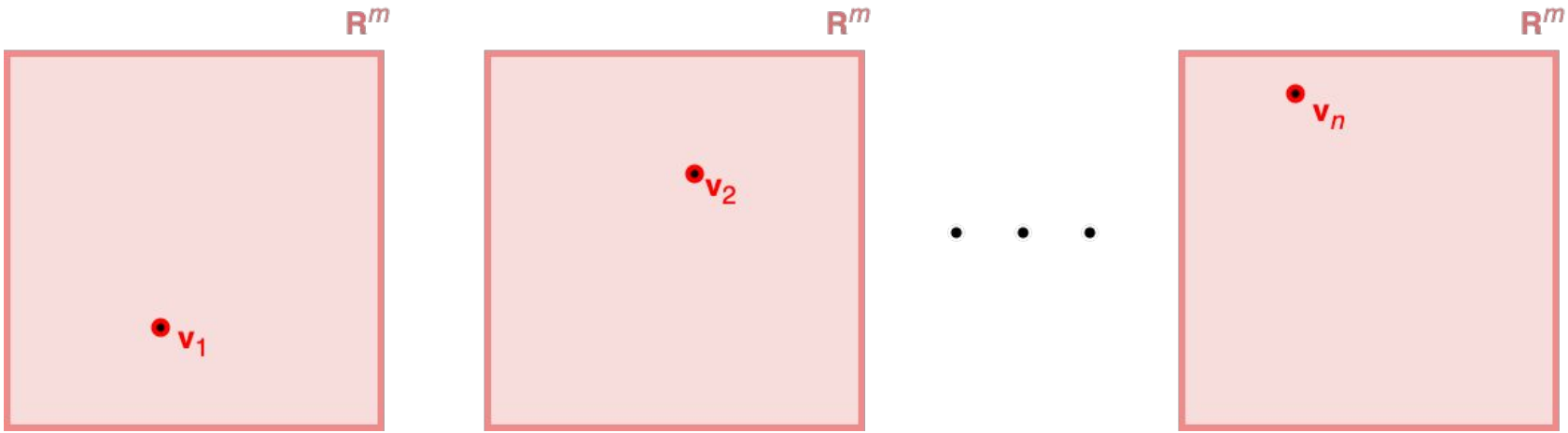
1. Break vector **v** up into lower-dimensional components: $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$, where **v** is $m \cdot n$-dimensional, and each $\mathbf{v}_i$ is m-dimensional.

Product Quantization (PQ).

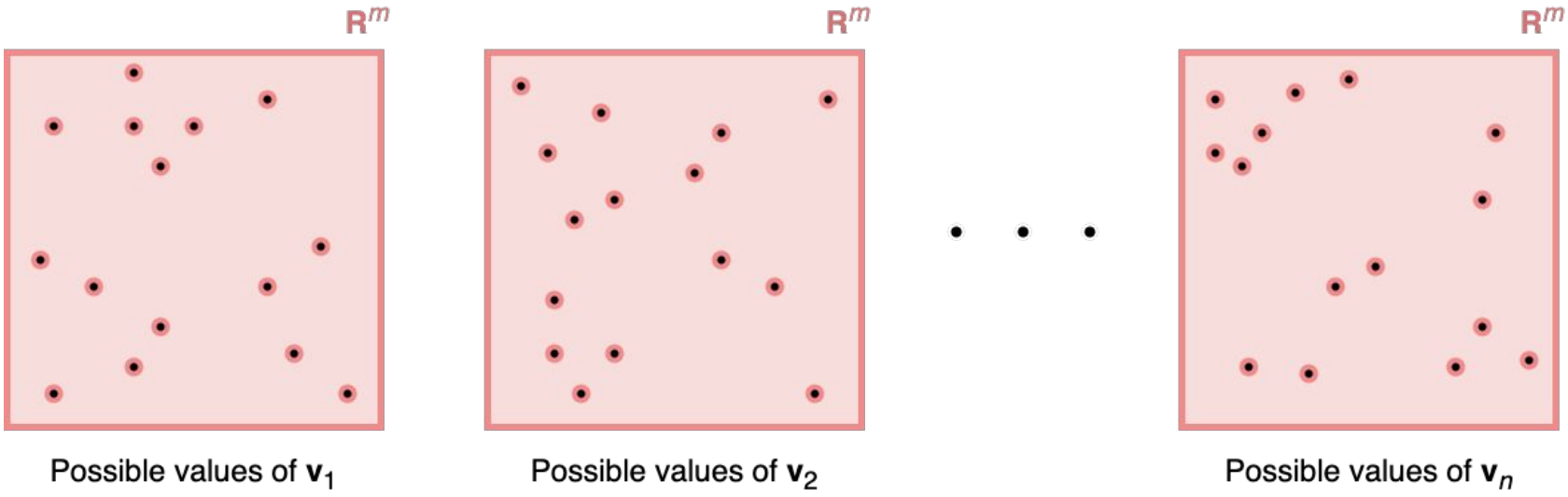
1. Break vector **v** up into lower-dimensional components: $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$, where **v** is $m \cdot n$-dimensional, and each $\mathbf{v}_i$ is m-dimensional.



Product Quantization (PQ).

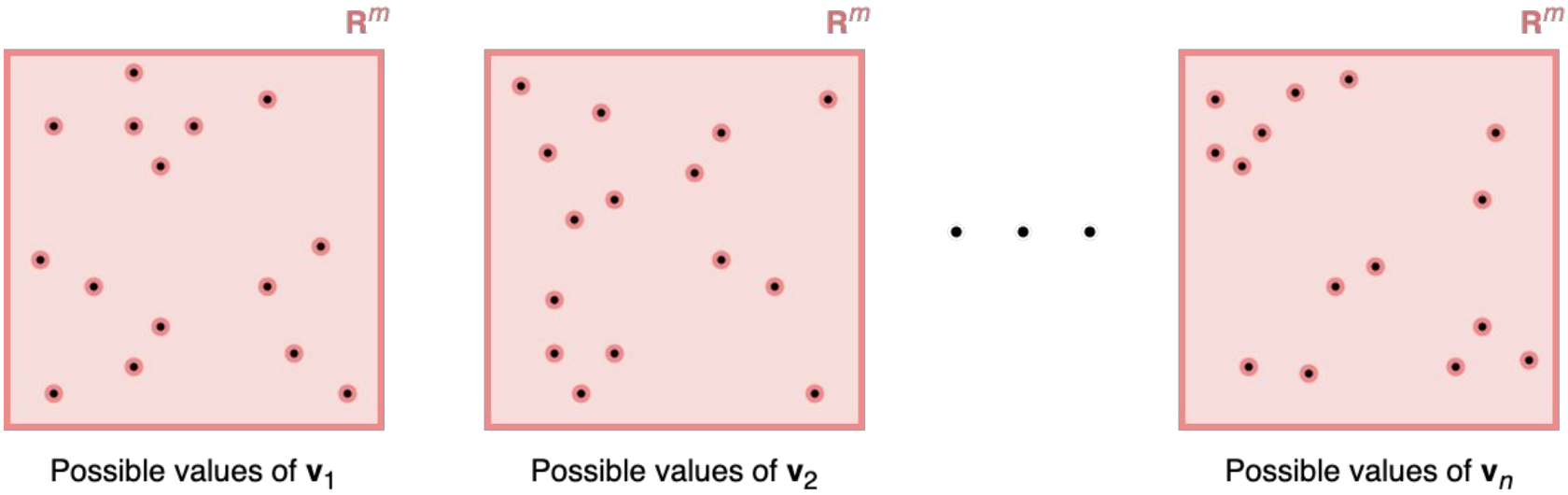


Product Quantization (PQ).



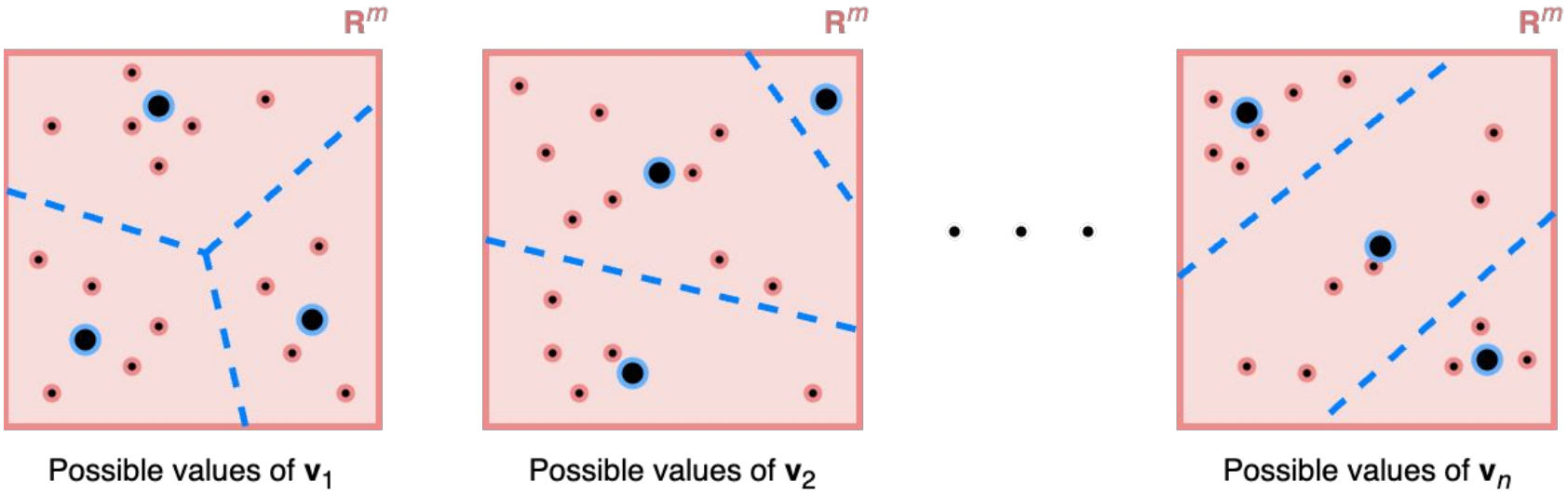
Product Quantization (PQ).

2. Use something like *k*-means clustering to partition sets of possible components into nearest-centroid classes.



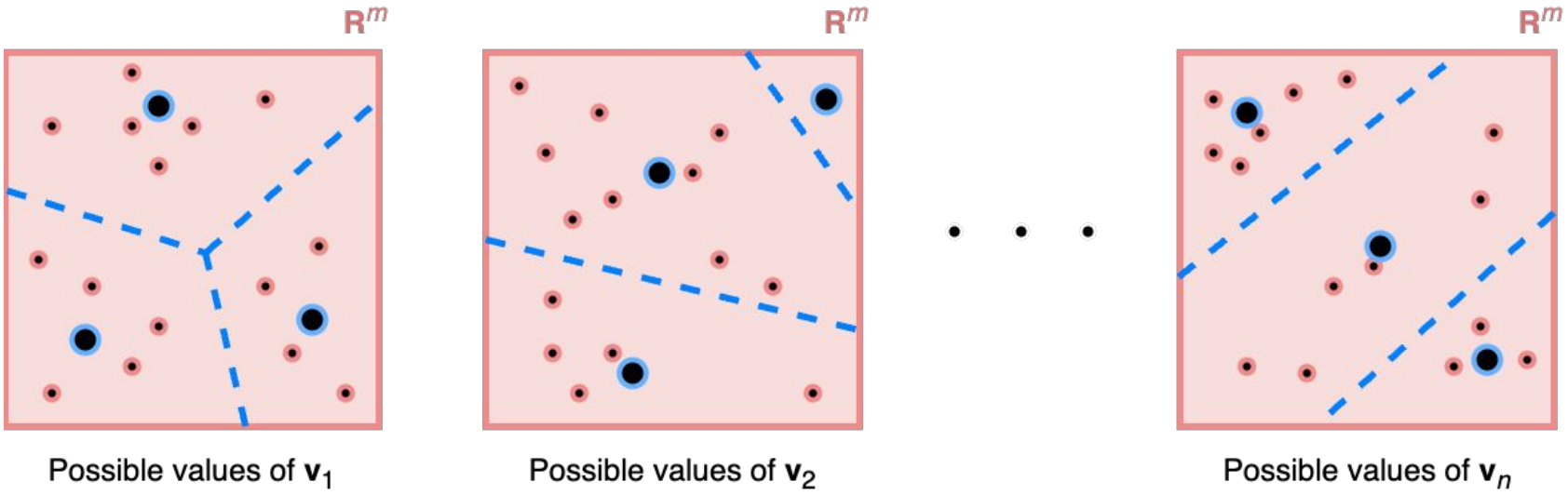
Product Quantization (PQ).

2. Use something like *k*-means clustering to partition sets of possible components into nearest-centroid classes.



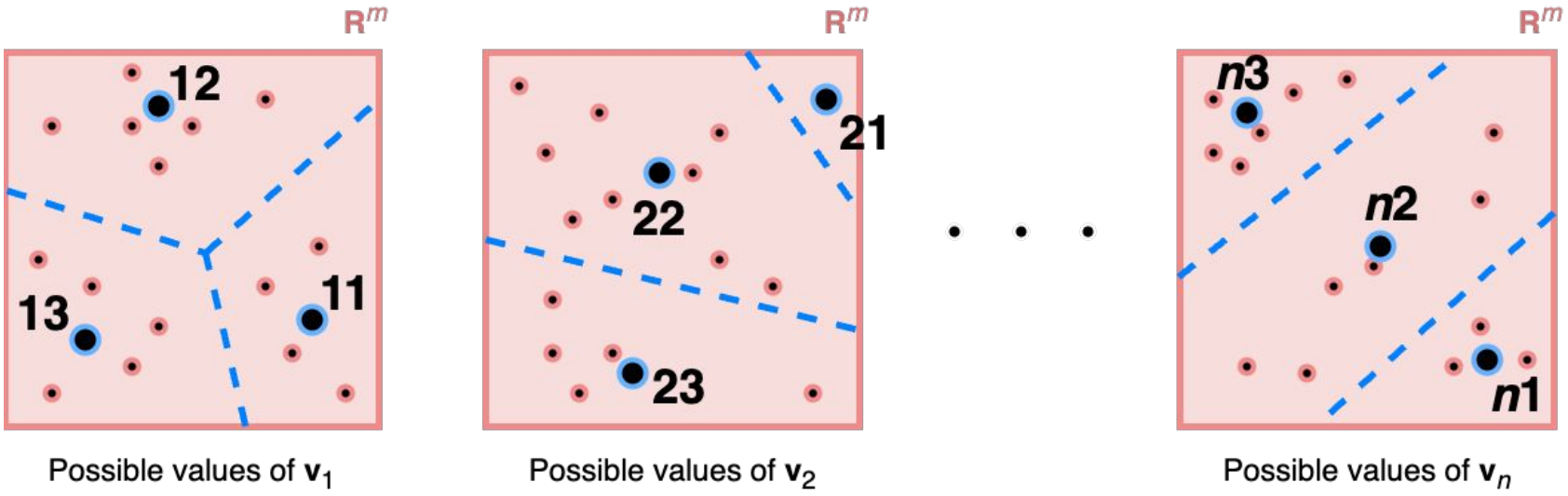
Product Quantization (PQ).

2. Use something like *k*-means clustering to partition sets of possible components into nearest-centroid classes. **Index the centroid classes.**



Product Quantization (PQ).

2. Use something like *k*-means clustering to partition sets of possible components into nearest-centroid classes. **Index the centroid classes.**



Product Quantization (PQ).

Product Quantization (PQ).

Assume that $k \ll m$.

Product Quantization (PQ).

Assume that $k \ll m$. We have the following compression:

Product Quantization (PQ).

Assume that $k \ll m$. We have the following compression:

$m \cdot n$-dimensional vector $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$

Product Quantization (PQ).

Assume that $k \ll m$. We have the following compression:

$m \cdot n$-dimensional vector $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$

to

Product Quantization (PQ).

Assume that $k \ll m$. We have the following compression:

$m \cdot n$-dimensional vector $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$

to

length-n tuple $\text{quant}(\mathbf{v}) = \big(\text{index}(\mathbf{v}_1), \text{index}(\mathbf{v}_2), \ldots, \text{index}(\mathbf{v}_n) \big)$

Product Quantization (PQ).

Assume that $k \ll m$. We have the following compression:

$m \cdot n$-dimensional vector $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$

to

length-n tuple $\text{quant}(\mathbf{v}) = \big(\text{index}(\mathbf{v}_1), \text{index}(\mathbf{v}_2), \ldots, \text{index}(\mathbf{v}_n) \big)$

Called a *quantization* of **v**.

Product Quantization (PQ).

Product Quantization (PQ).

Start with an $m \cdot n$-dimensional vector $\mathbf{v}$.

Product Quantization (PQ).

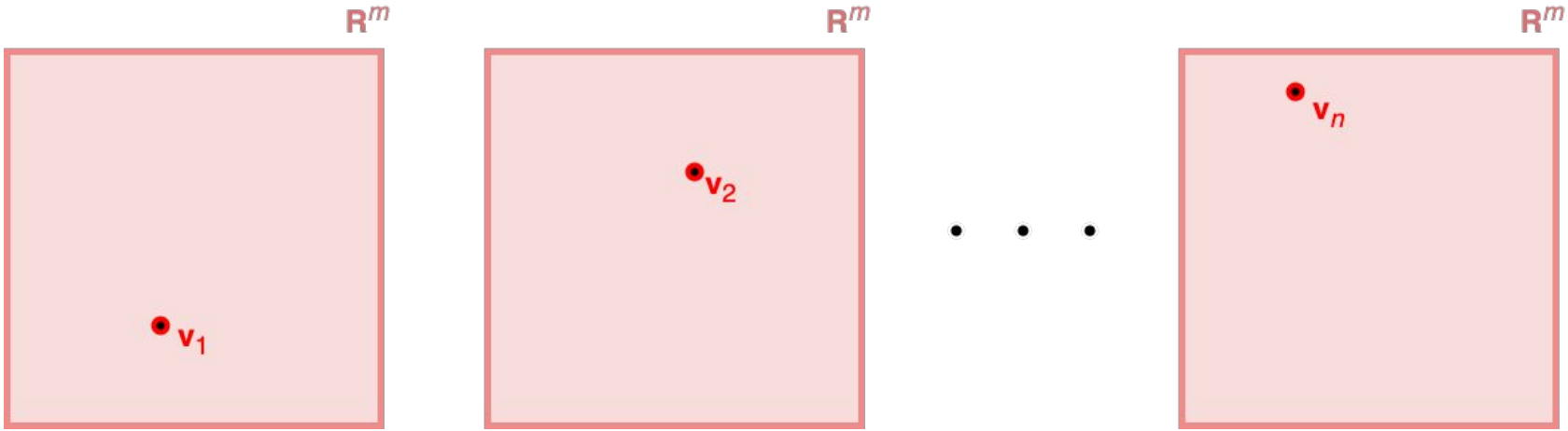
Start with an $m \cdot n$-dimensional vector $\mathbf{v}$.

Decompose it into n vectors of dimension m: $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$

Product Quantization (PQ).

Start with an $m \cdot n$-dimensional vector $\mathbf{v}$.

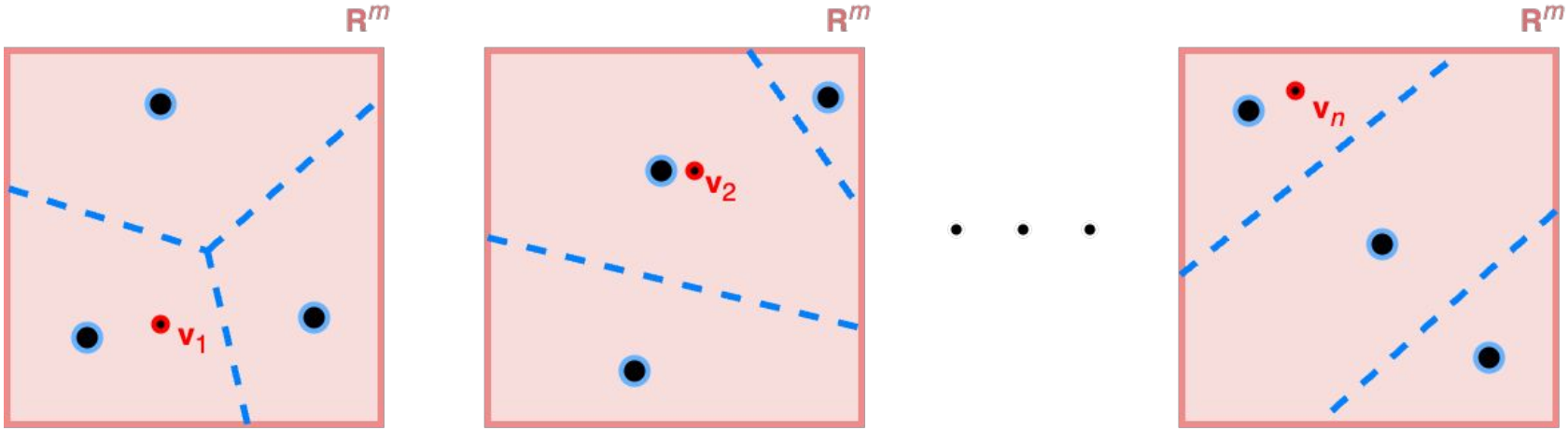
Decompose it into n vectors of dimension m: $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$



Product Quantization (PQ).

Start with an $m \cdot n$-dimensional vector $\mathbf{v}$.

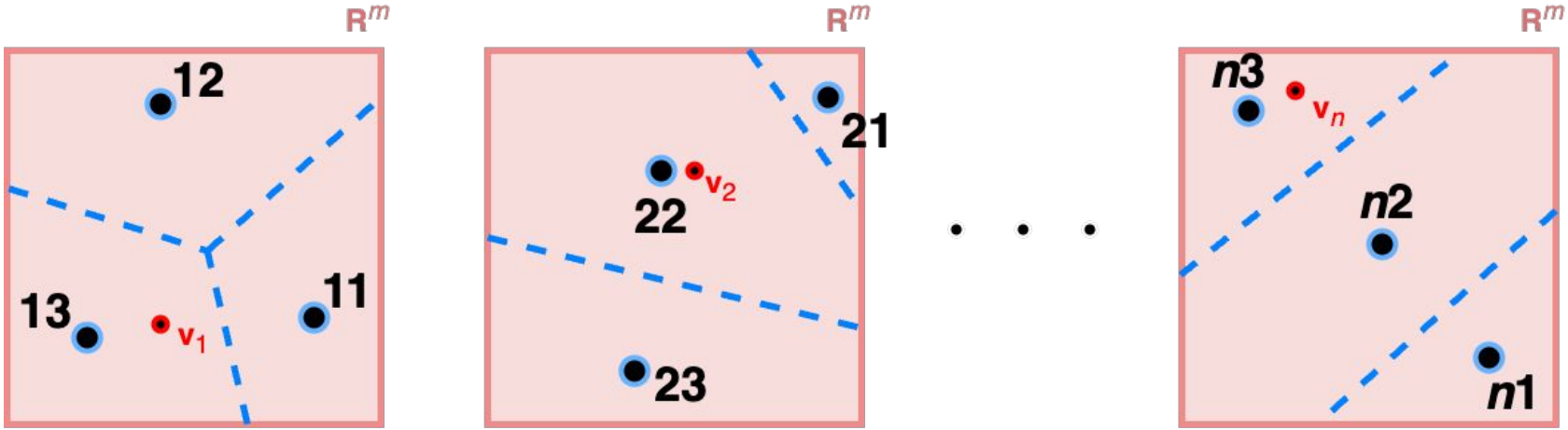
Decompose it into n vectors of dimension m: $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$



Product Quantization (PQ).

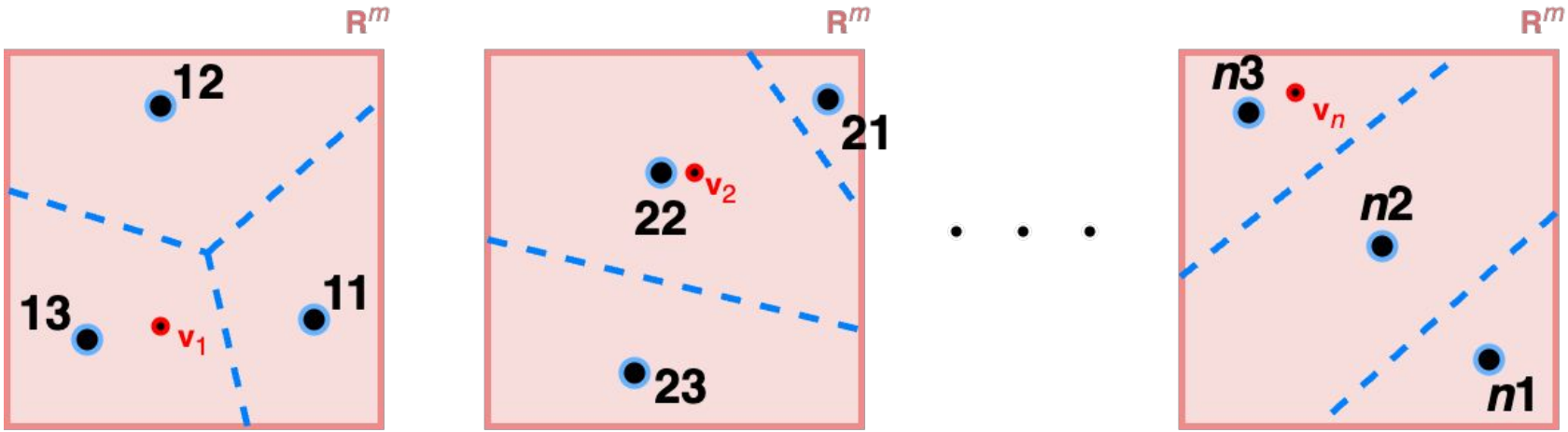
Start with an $m \cdot n$-dimensional vector **v**.

Decompose it into n vectors of dimension m: $\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$



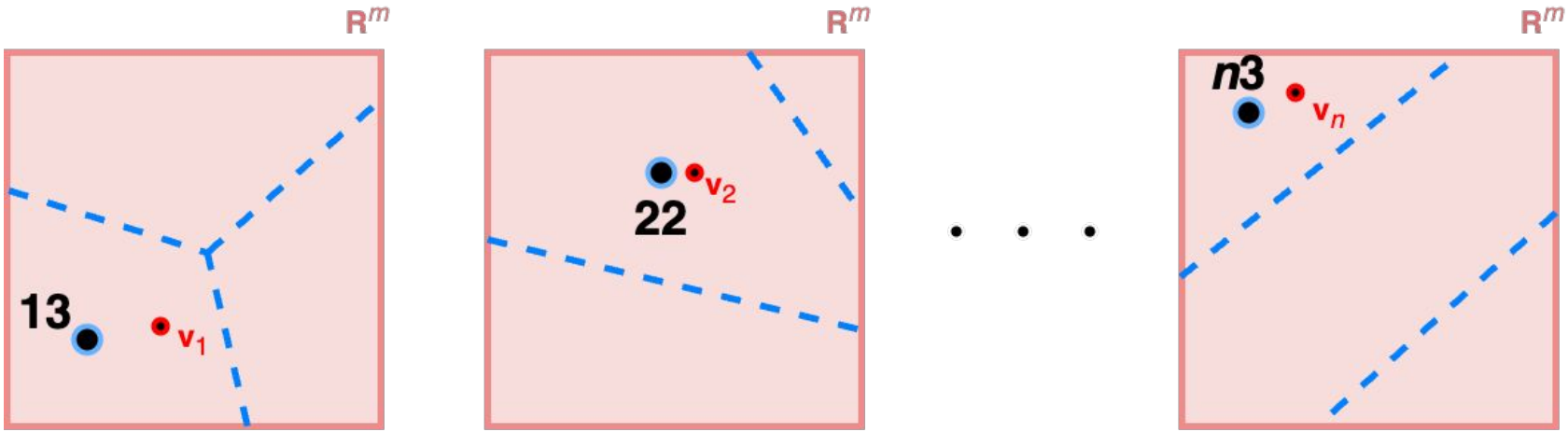
Product Quantization (PQ).

Find nearest centroid of each component vector:



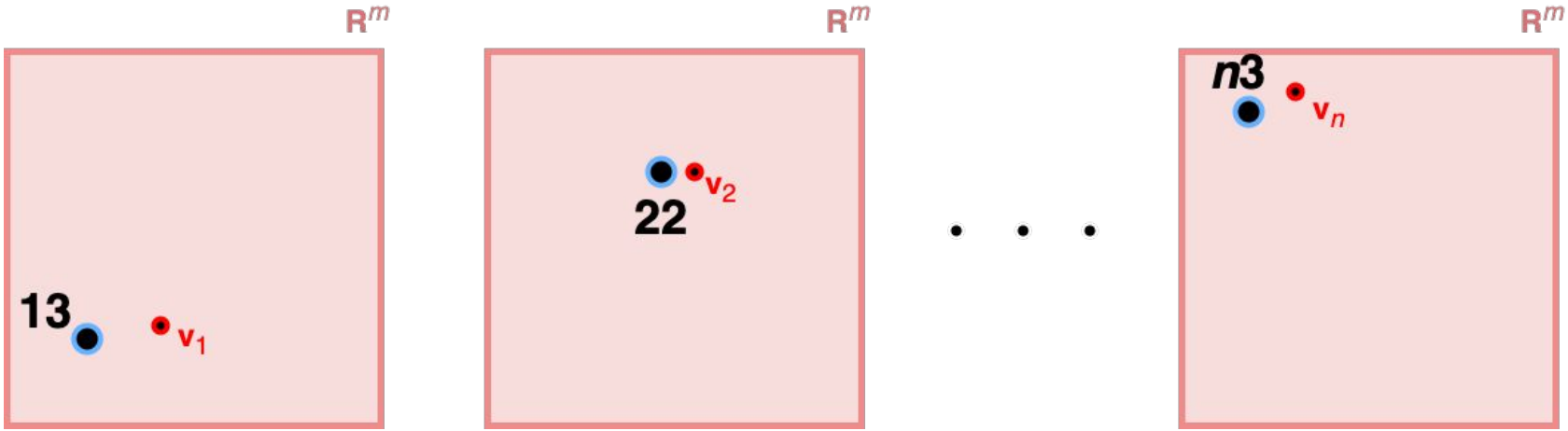
Product Quantization (PQ).

Find nearest centroid of each component vector:



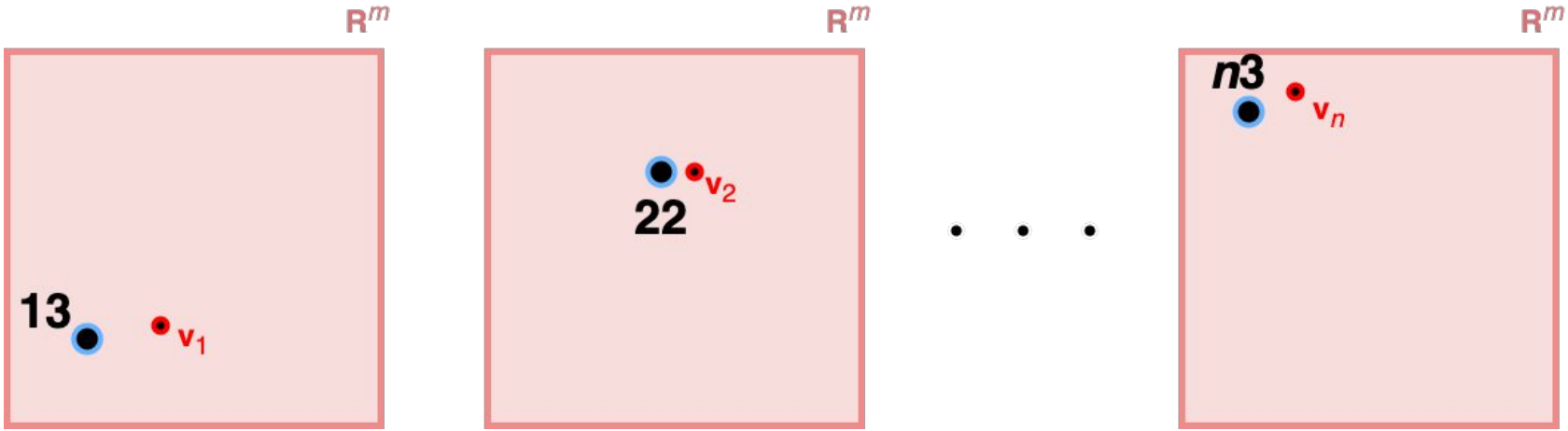
Product Quantization (PQ).

Find nearest centroid of each component vector:



Product Quantization (PQ).

$\mathbf{v} = \mathbf{v}_1 \times \mathbf{v}_2 \times \cdots \times \mathbf{v}_n$ **is compressed to** quant(**v**) = (**13**, **22**, …, ***n*3**)



Product Quantization (PQ).

Product Quantization (PQ).

How do we use this to speed up query execution?

Inverted Vector File (IVF).

Inverted Vector File (IVF).

Example of an arbitrary record in “main collection” in our vector database, without product quantization:

Inverted Vector File (IVF).

Example of an arbitrary record in “main collection” in our vector database, without product quantization:

	record _ID	property_1	property_2	···	property_*n*	encoding
record	*serial number*	$value_1$	$value_2$	···	$value_n$	**v**

Inverted Vector File (IVF).

Example of an arbitrary record in “main collection” in our vector database, without product quantization:

	record _ID	property_1	property_2	···	property_*n*	encoding
record	*serial number*	$value_1$	$value_2$	···	$value_n$	**v**

Example of an “inverted record” in auxiliary collection, making use of product quantization:

Inverted Vector File (IVF).

Example of an arbitrary record in “main collection” in our vector database, without product quantization:

	record _ID	**property_1**	**property_2**	**…**	**property_*n***	**encoding**
record	*serial number*	$value_1$	$value_2$	…	$value_n$	**v**

Example of an “inverted record” in auxiliary collection, making use of product quantization:

	quantized_vector	**SELECT record_ID WHERE** quant(**v**) = *centroid tuple*
“inverted record”	*centroid tuple*	*list* : [$record\_ID_1$, $record\_ID_2$, …,$record\_ID_L$]

Inverted Vector File (IVF).

Inverted Vector File (IVF).

	quantized_vector	**SELECT record_ID WHERE** quant(**v**) = *centroid tuple*
“inverted record”	*centroid tuple*	*list* : [$record\_ID_1$, $record\_ID_2$, …,$record\_ID_L$]

Inverted Vector File (IVF).

	quantized_vector	**SELECT record_ID WHERE** quant(**v**) = *centroid tuple*
“inverted record”	*centroid tuple*	*list* : [$record\_ID_1$, $record\_ID_2$, …,$record\_ID_L$]

This inverted record satisfies two nice conditions:

Inverted Vector File (IVF).

	quantized_vector	**SELECT record_ID WHERE** quant(**v**) = *centroid tuple*
“inverted record”	*centroid tuple*	*list* : [$record\_ID_1$, $record\_ID_2$, …,$record\_ID_L$]

This inverted record satisfies two nice conditions:

1. There are far fewer inverted records than normal records, since each inverted record collects many record_IDs.

Inverted Vector File (IVF).

	quantized_vector	**SELECT record_ID WHERE** quant(**v**) = *centroid tuple*
"inverted record"	*centroid tuple*	*list* : [$record\_ID_1$, $record\_ID_2$, …,$record\_ID_L$]

This inverted record satisfies two nice conditions:

1. There are far fewer inverted records than normal records, since each inverted record collects many record_IDs.
2. It requires less compute to perform linear-algebraic calculations with centroid tuples than with the original vectors **v**.

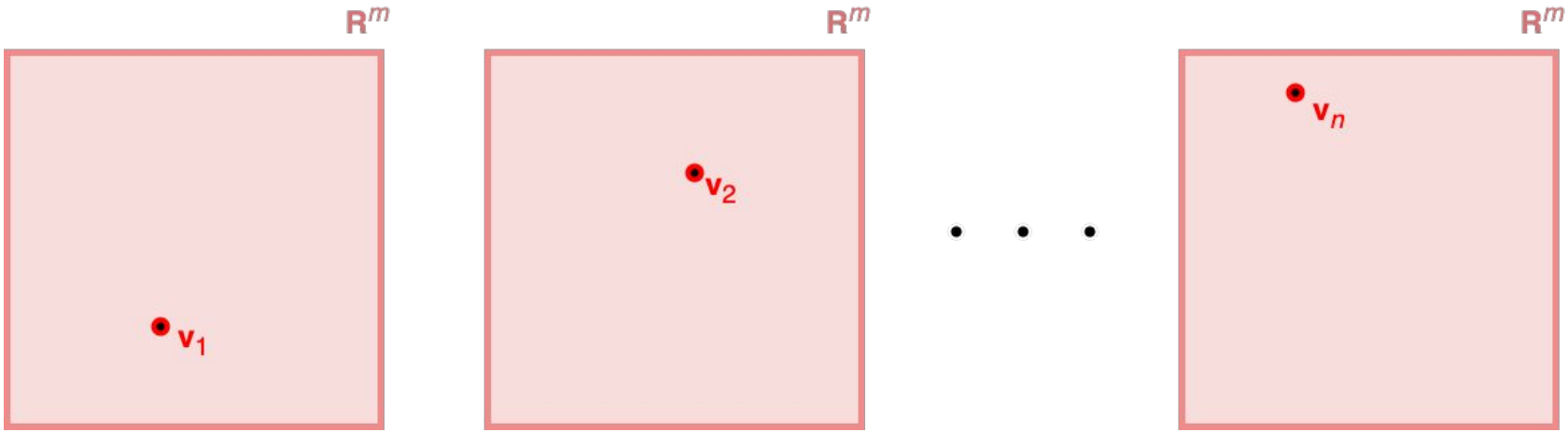
Inverted Vector File (IVF).

	quantized_vector	**SELECT record_ID WHERE** quant(**v**) = *centroid tuple*
"inverted record"	*centroid tuple*	*list* : [$record\_ID_1$, $record\_ID_2$, …,$record\_ID_L$]

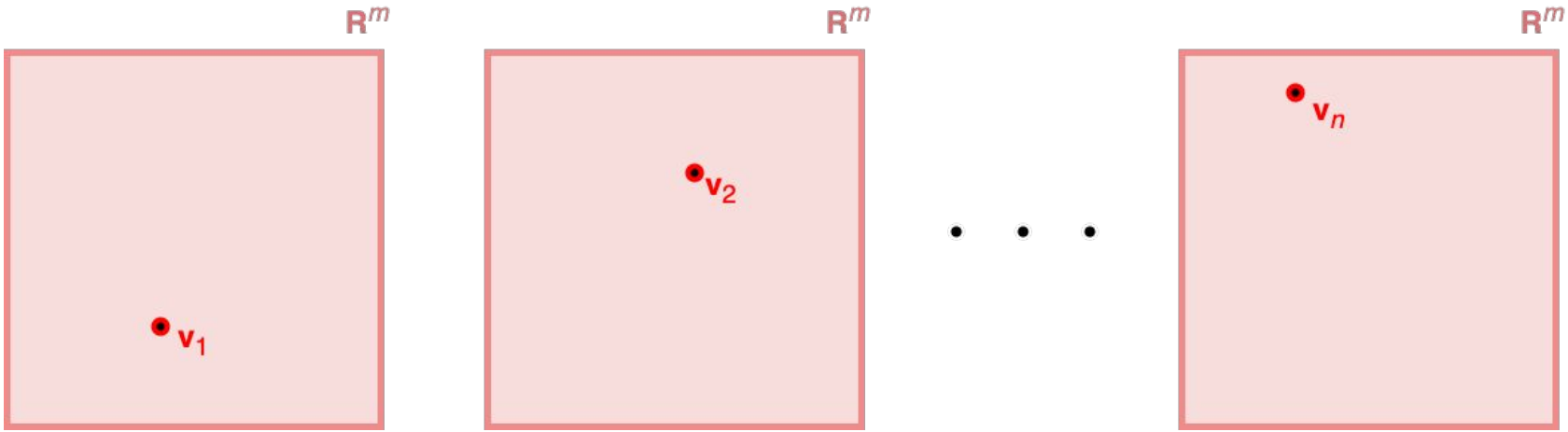
This inverted record satisfies two nice conditions:

1. There are far fewer inverted records than normal records, since each inverted record collects many record_IDs.
2. It requires less compute to perform linear-algebraic calculations with centroid tuples than with the original vectors **v**. (Think block matrices…)

Transforming each component $\mathbf{v}_i$ separately means fewer arithmetic operations for matrix multiplication:



$$\begin{pmatrix} \Phi_1(\mathbf{v}_1) \\ \Phi_2(\mathbf{v}_2) \\ \vdots \\ \Phi_n(\mathbf{v}_n) \end{pmatrix} = \begin{pmatrix} \boxed{\Phi_1} & & 0 \\ & \boxed{\Phi_2} & \\ 0 & & \ddots \\ & & \boxed{\Phi_n} \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \vdots \\ \mathbf{v}_n \end{pmatrix}$$



RECALL:

Inverted Vector File (IVF).

	quantized_vector	**SELECT record_ID WHERE** quant(**v**) = *centroid tuple*
"inverted record"	*centroid tuple*	*list* : [$record\_ID_1$, $record\_ID_2$, …,$record\_ID_L$]

This inverted record satisfies two nice conditions:

1. There are far fewer inverted records than normal records, since each inverted record collects many record_IDs.
2. It requires less compute to perform linear-algebraic calculations with centroid tuples than with the original vectors **v**.

RECALL:

Inverted Vector File (IVF).

	quantized_vector	**SELECT record_ID WHERE** quant(**v**) = *centroid tuple*
"inverted record"	*centroid tuple*	*list* : [$record\_ID_1$, $record\_ID_2$, …,$record\_ID_L$]

This inverted record satisfies two nice conditions:

1. There are far fewer inverted records than normal records, since each inverted record collects many record_IDs.
2. It requires less compute to perform linear-algebraic calculations with centroid tuples than with the original vectors **v**.

⇒ substantial speed-up for query execution

Hierarchical Navigable Small World (HNSW).

Hierarchical Navigable Small World (HNSW).

Think about how you actually look up words in a physical dictionary (a book).

Hierarchical Navigable Small World (HNSW).

Think about how you actually look up words in a physical dictionary (a book).

Having a **key** (the word you're looking up), you…

Hierarchical Navigable Small World (HNSW).

Think about how you actually look up words in a physical dictionary (a book).

Having a **key** (the word you're looking up), you…

1. …Randomly open dictionary to **page**$_1$.

Hierarchical Navigable Small World (HNSW).

Think about how you actually look up words in a physical dictionary (a book).

Having a **key** (the word you're looking up), you…

1. …Randomly open dictionary to **page**$_1$. Randomly select **word**$_1$ on **page**$_1$.

Hierarchical Navigable Small World (HNSW).

Think about how you actually look up words in a physical dictionary (a book).

Having a **key** (the word you're looking up), you…

1. …Randomly open dictionary to **page**$_1$. Randomly select **word**$_1$ on **page**$_1$. Depending on alphabetical relation between **key** and **word**$_1$, you…

Hierarchical Navigable Small World (HNSW).

Think about how you actually look up words in a physical dictionary (a book).

Having a **key** (the word you're looking up), you…

1. …Randomly open dictionary to **page**$_1$. Randomly select **word**$_1$ on **page**$_1$. Depending on alphabetical relation between **key** and **word**$_1$, you…
2. …Open dictionary to random new **page**$_2$ occurring before or after **page**$_1$ (depending on alphabetical relation between **key** and **word**$_1$).

Hierarchical Navigable Small World (HNSW).

Think about how you actually look up words in a physical dictionary (a book).

Having a **key** (the word you're looking up), you…

1. …Randomly open dictionary to **page**$_1$. Randomly select **word**$_1$ on **page**$_1$. Depending on alphabetical relation between **key** and **word**$_1$, you…
2. …Open dictionary to random new **page**$_2$ occurring before or after **page**$_1$ (depending on alphabetical relation between **key** and **word**$_1$). Randomly select **word**$_2$ on **page**$_2$,

Hierarchical Navigable Small World (HNSW).

Think about how you actually look up words in a physical dictionary (a book).

Having a **key** (the word you're looking up), you…

1. …Randomly open dictionary to **page**$_1$. Randomly select **word**$_1$ on **page**$_1$. Depending on alphabetical relation between **key** and **word**$_1$, you…
2. …Open dictionary to random new **page**$_2$ occurring before or after **page**$_1$ (depending on alphabetical relation between **key** and **word**$_1$). Randomly select **word**$_2$ on **page**$_2$, compare against **key**, and repeat…

Hierarchical Navigable Small World (HNSW).

Hierarchical Navigable Small World (HNSW).

This example is essentially “binary search” with some random sampling thrown in.

Hierarchical Navigable Small World (HNSW).

This example is essentially “binary search” with some random sampling thrown in.

Results in $O(N) \longrightarrow O(\log N)$ speed-up when searching through a list of N items.

Hierarchical Navigable Small World (HNSW).

This example is essentially “binary search” with some random sampling thrown in.

Results in $O(N) \longrightarrow O(\log N)$ speed-up when searching through a list of N items.

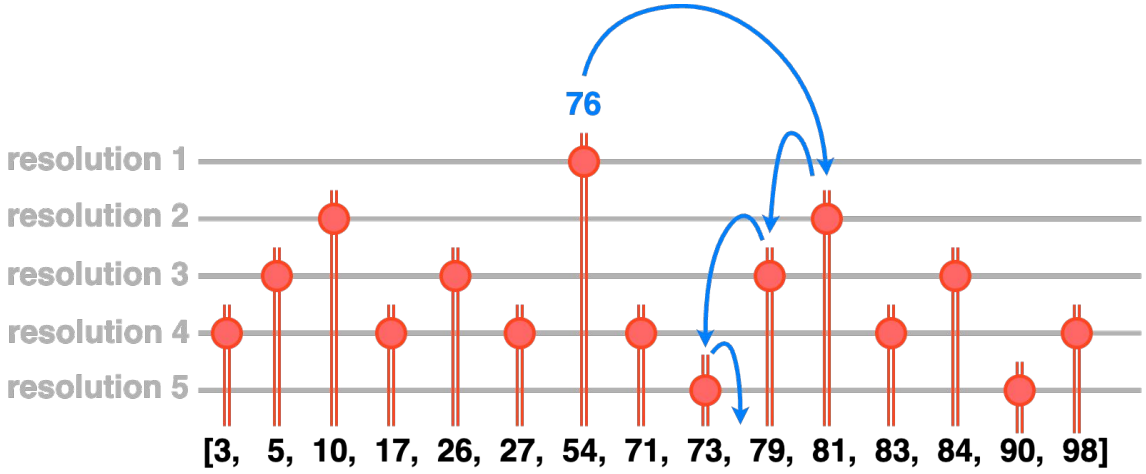
We can think of binary search as a “multi-resolution” comparison algorithm:

Hierarchical Navigable Small World (HNSW).

This example is essentially “binary search” with some random sampling thrown in.

Results in $O(N) \longrightarrow O(\log N)$ speed-up when searching through a list of N items.

We can think of binary search as a “multi-resolution” comparison algorithm:



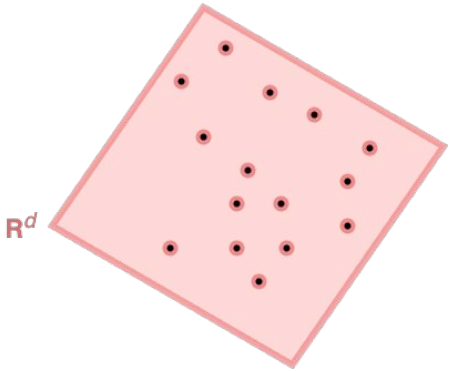
Hierarchical Navigable Small World (HNSW).

Hierarchical Navigable Small World (HNSW).

Try running linear-algebraic operations on vectors at multiple resolutions.

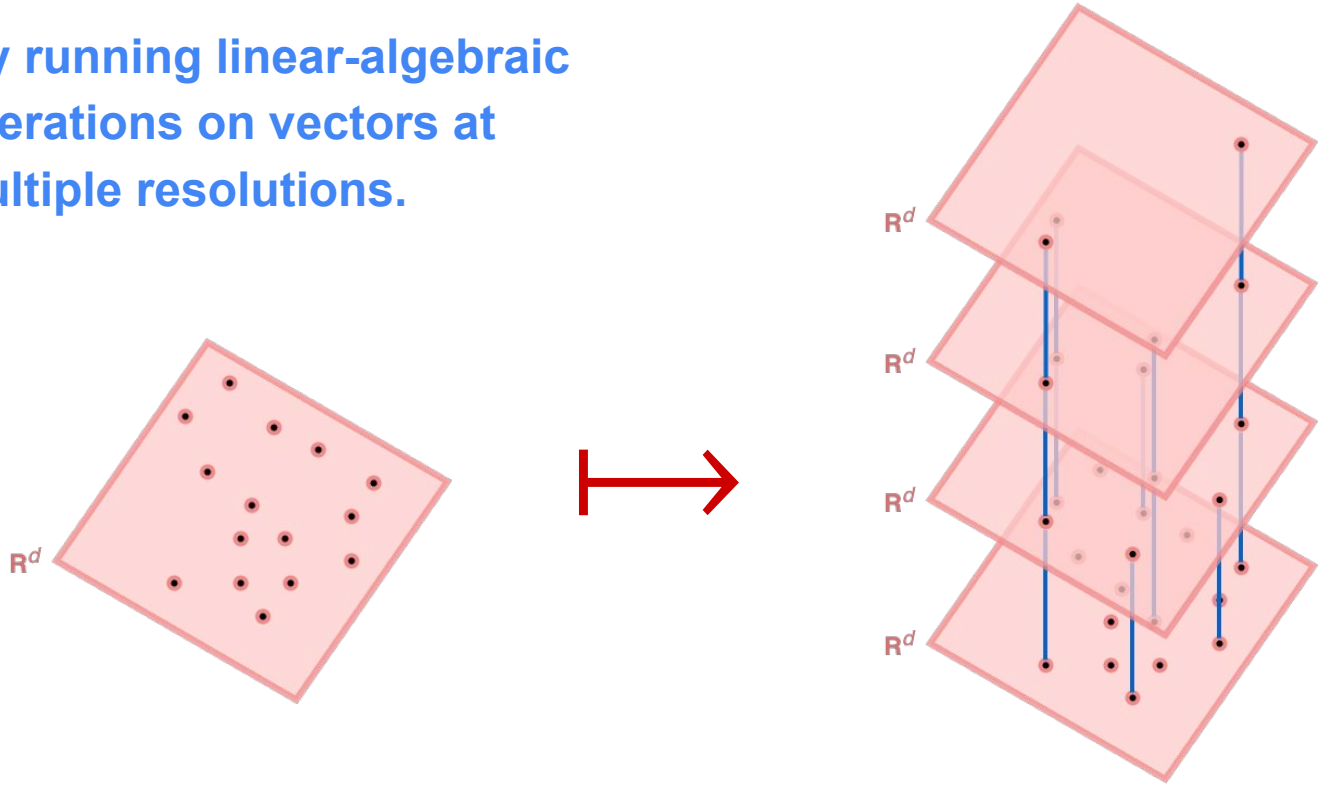
Hierarchical Navigable Small World (HNSW).

Try running linear-algebraic operations on vectors at multiple resolutions.



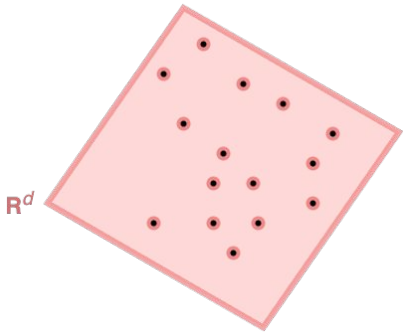
Hierarchical Navigable Small World (HNSW).

Try running linear-algebraic operations on vectors at multiple resolutions.

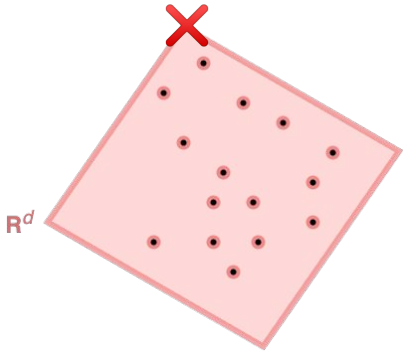


Proceed like binary search, but with **“nearest next centroid”** replacing **“≷ next axis”**

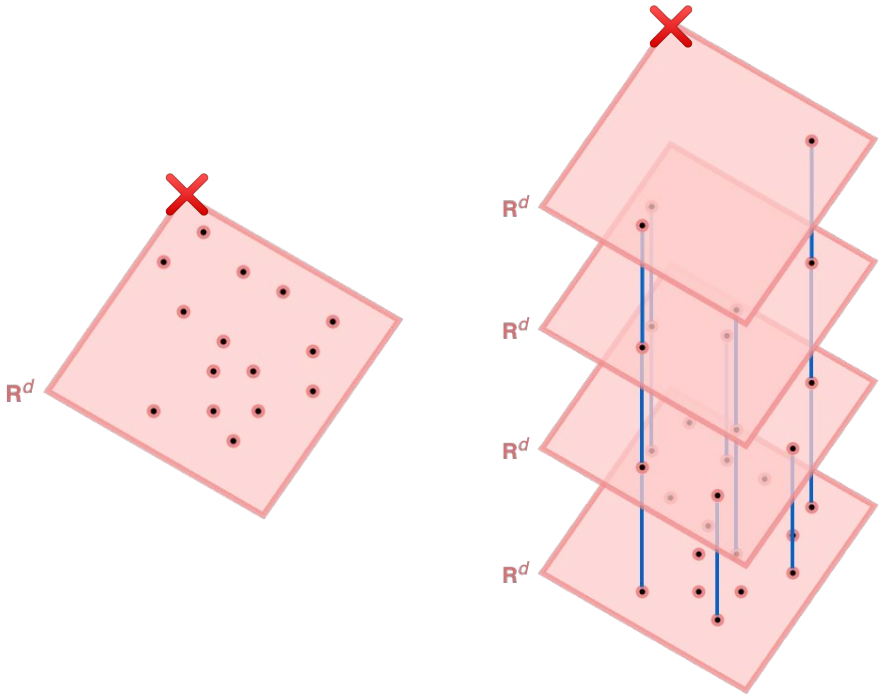
Proceed like binary search, but with **“nearest next centroid”** replacing **“≷ next axis”**



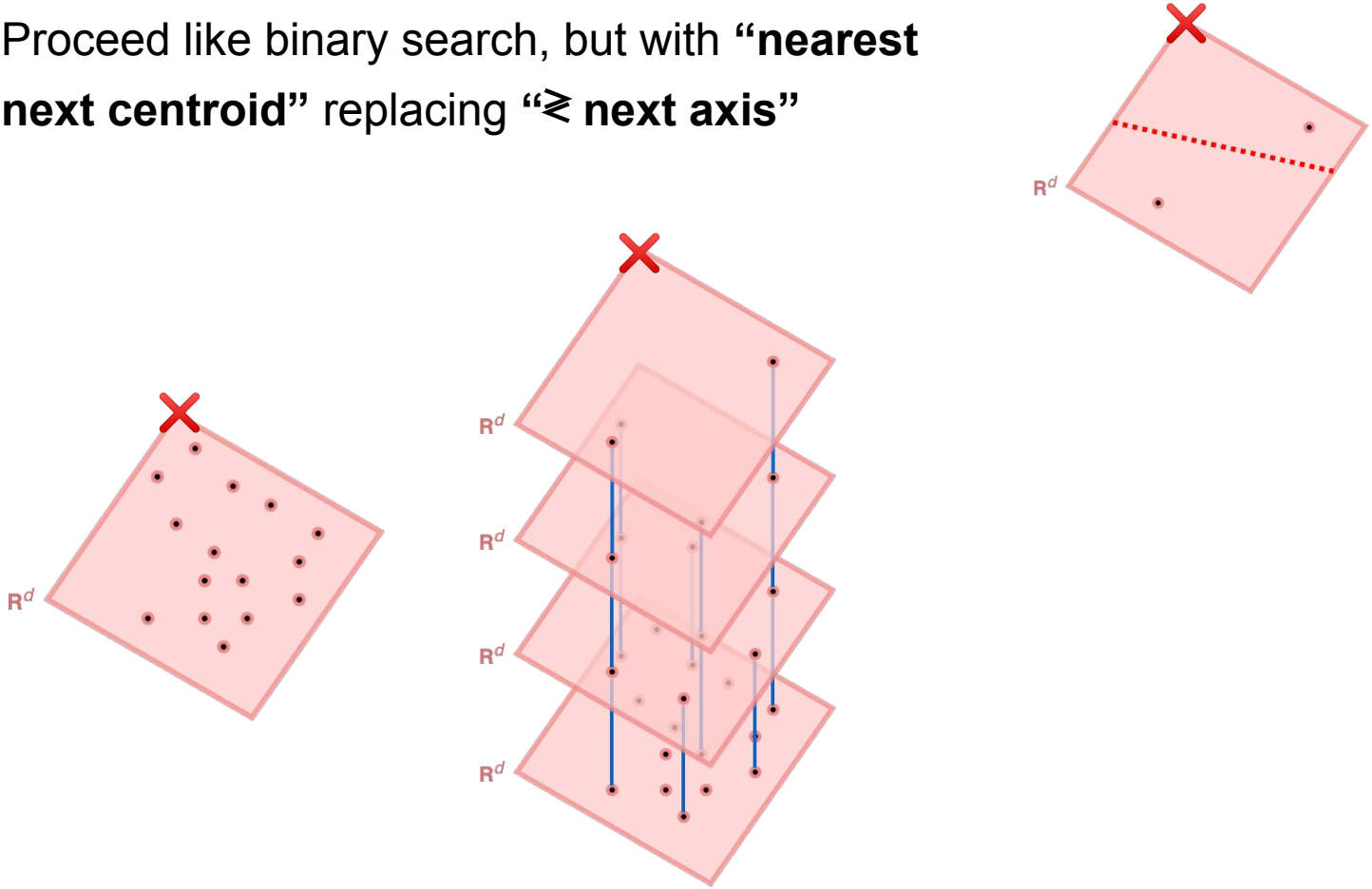
Proceed like binary search, but with **“nearest next centroid”** replacing **“≷ next axis”**



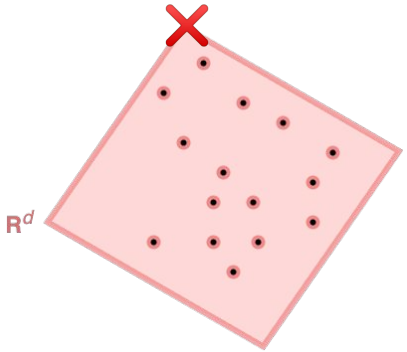
Proceed like binary search, but with **“nearest next centroid”** replacing **“≷ next axis”**

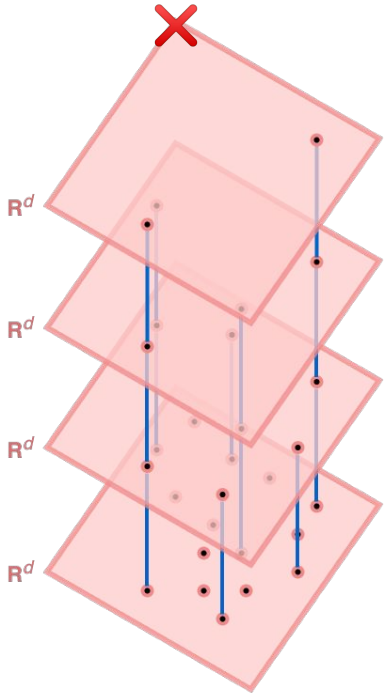


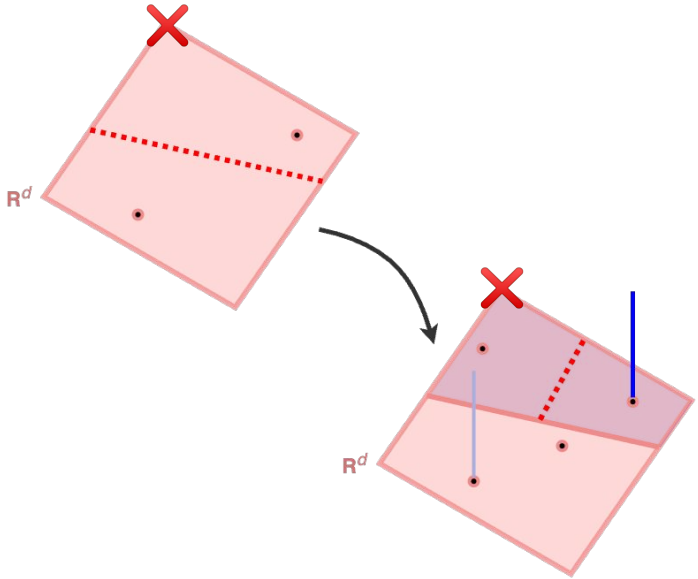
Proceed like binary search, but with **“nearest next centroid”** replacing **“≷ next axis”**



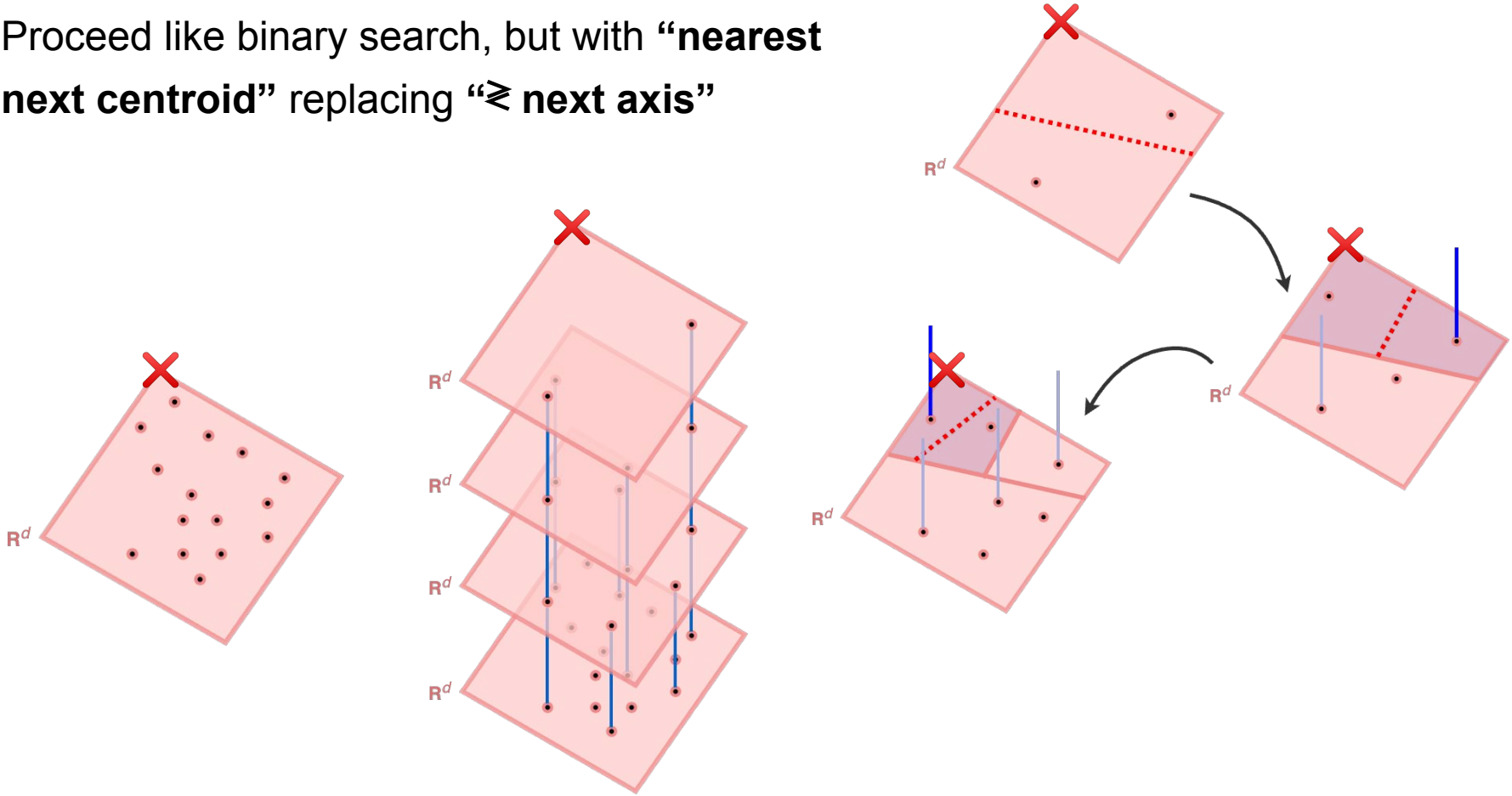
Proceed like binary search, but with **“nearest next centroid”** replacing **“≷ next axis”**



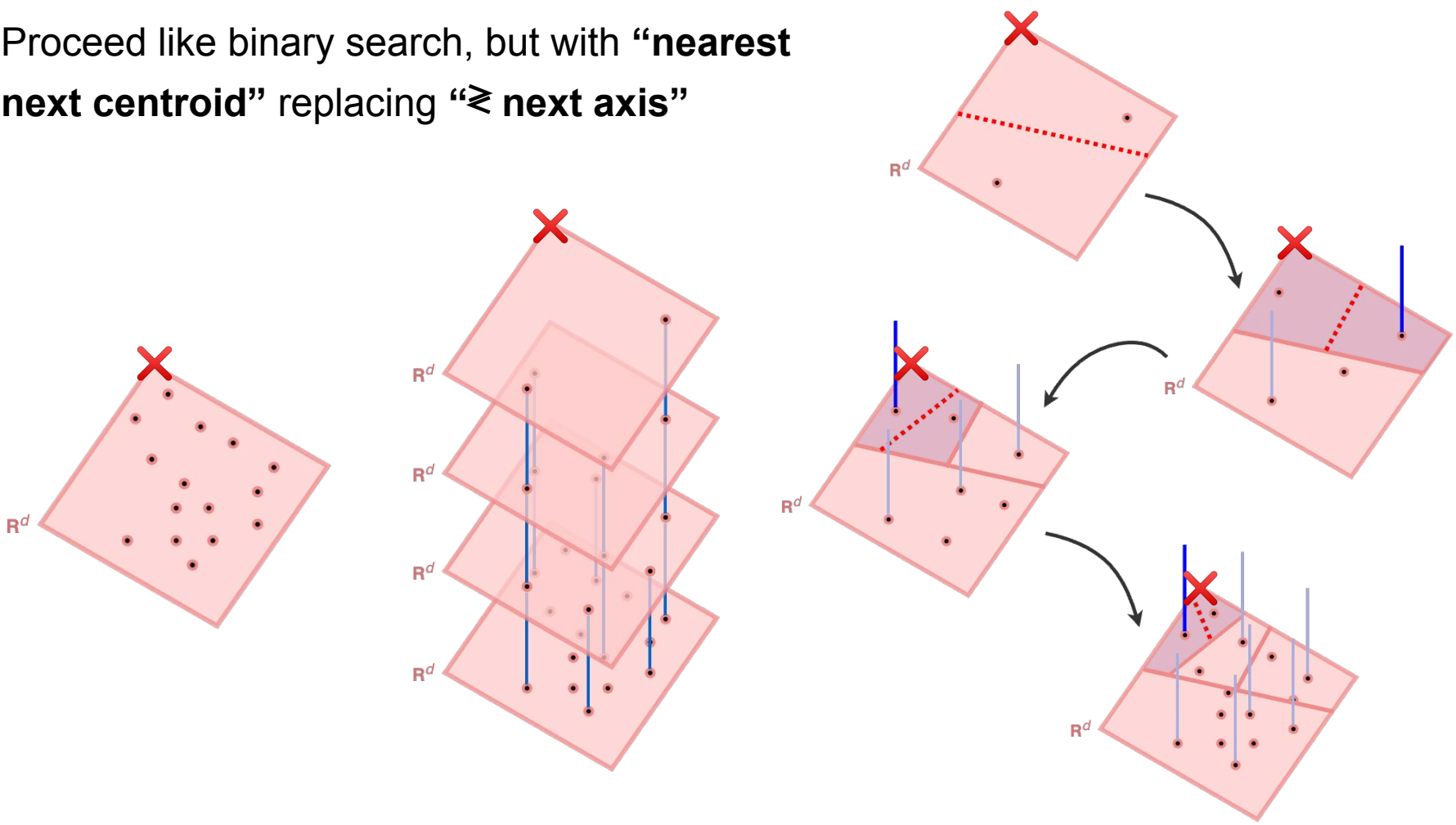




Proceed like binary search, but with **“nearest next centroid”** replacing **“≷ next axis”**



Proceed like binary search, but with **“nearest next centroid”** replacing **“≷ next axis”**



Thank you!