

Carnegie Mellon University

Database Systems

15-445/645 SPRING 2026

ANDY PAVLO

JIGNESH PATEL

Lecture #09

Indexes + Filters:
Bloom Filters, Skip Lists,
Tries, Inverted Indexes,
Vector Indexes



ADMINISTRIVIA



Project #1 is due Sunday Feb 15th @ 11:59pm

→ Recitation Video + Slides ([@64](#))

→ Perf Recitation Video + Slides ([@79](#))

→ Special OH on Saturday Feb 14th @ 3:00-5:00pm (GHC 5207)

Homework #3 is due Sunday Feb 22nd @ 11:59pm

Mid-Term Exam is on Wednesday Feb 25th @ 2:00pm

→ Lectures #01–11 (inclusive)

→ Study guide will be released early next week.

INDEXES VS. FILTERS

An **index** data structure of a subset of a table's attributes that are organized and/or sorted to the location of specific tuples using those attributes.

→ Example: B+Tree

A **filter** is a data structure that answers set membership queries; it tells you whether a key (likely) exists in a set but not where it is located.

→ Example: Bloom Filter

TODAY'S AGENDA

Bloom Filters

Skip Lists

Tries / Radix Trees

Inverted Indexes

Vector Indexes



BLOOM FILTERS



Probabilistic data structure (bitmap) that answers set membership queries.

- False negatives will never occur.
- False positives can sometimes occur.
- See [Bloom Filter Calculator](#).

Insert(x):

- Use k hash functions to set bits in the filter to 1.

Lookup(x):

- Check whether the bits are 1 for each hash function.

BLOOM FILTERS

Insert 'RZA'

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

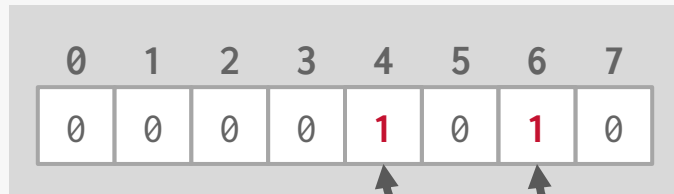
$$\text{hash}_1('RZA') = 2222 \% 8 = 6$$

$$\text{hash}_2('RZA') = 4444 \% 8 = 4$$

BLOOM FILTERS

Insert 'RZA'

Bloom Filter



$$\text{hash}_1('RZA') = 2222 \% 8 = 6$$

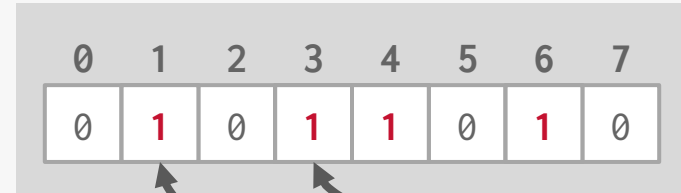
$$\text{hash}_2('RZA') = 4444 \% 8 = 4$$

BLOOM FILTERS

Insert 'RZA'

Insert 'GZA'

Bloom Filter



$$\text{hash}_1('GZA') = 5555 \% 8 = 3$$

$$\text{hash}_2('GZA') = 7777 \% 8 = 1$$

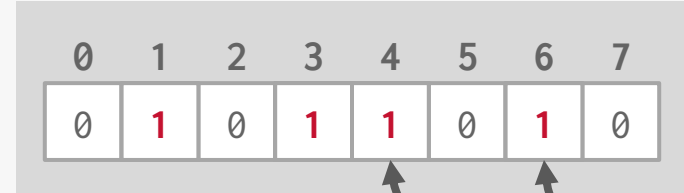
BLOOM FILTERS

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Bloom Filter



$$\text{hash}_1('RZA') = 2222 \% 8 = 6$$

$$\text{hash}_2('RZA') = 4444 \% 8 = 4$$

BLOOM FILTERS

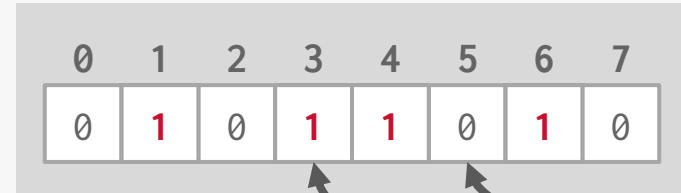
Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Bloom Filter



$$\text{hash}_1(\text{'Raekwon'}) = 3333 \% 8 = 5$$
$$\text{hash}_2(\text{'Raekwon'}) = 8899 \% 8 = 3$$

BLOOM FILTERS

Insert 'RZA'

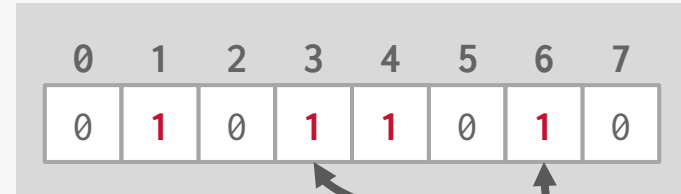
Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Lookup 'ODB' → *TRUE*

Bloom Filter



$$\begin{aligned} \text{hash}_1('ODB') &= 6699 \% 8 = 3 \\ \text{hash}_2('ODB') &= 9966 \% 8 = 6 \end{aligned}$$

OTHER FILTERS



Counting Bloom Filter

- Supports dynamically adding and removing keys.
- Uses integers instead of bits to count the number of occurrences of a key in a set.

Carnegie
Mellon
University



Cuckoo Filter

- Also supports dynamically adding and removing keys.
- Uses a Cuckoo Hash Table but stores fingerprints instead of full keys.

Carnegie
Mellon
University



Succinct Range Filter (SuRF)

- Immutable compact trie that supports approximate exact matches and range filtering.

Counting Bloom Filter

- Supports dynamically adding and removing elements
- Uses integers instead of bits to count occurrences of a key in a set.

Carnegie
Mellon
University



Cuckoo Filter

- Also supports dynamically adding and removing elements
- Uses a Cuckoo Hash Table but not bloom keys.

Carnegie
Mellon
University



Succinct Range Filter (SuRF)

- Immutable compact trie that supports fast membership matches and range filtering.

The screenshot shows the Redis documentation page for the Cuckoo filter. The page is titled "Cuckoo filter" and is part of the "Develop with Redis" section. It includes a navigation menu on the left with categories like "Quick starts", "Connect", "Understand data types", "Probabilistic", "Interact with data", "Libraries and tools", "Redis products", and "Commands". The "Cuckoo filter" page content includes an introduction, a comparison with Bloom filters, and use cases such as "Targeted ad campaigns (advertising, retail)" and "Discount code/coupon validation (retail, online shops)".

Redis

Docs → Develop with Redis → Understand Redis data types → Probabilistic → Cuckoo filter

Cuckoo filter

Cuckoo filters are a probabilistic data structure that checks for presence of an element in a set

A Cuckoo filter, just like a Bloom filter, is a probabilistic data structure in Redis Stack that enables you to check if an element is present in a set in a very fast and space efficient way, while also allowing for deletions and showing better performance than Bloom in some scenarios.

While the Bloom filter is a bit array with flipped bits at positions decided by the hash function, a Cuckoo filter is an array of buckets, storing fingerprints of the values in one of the buckets at positions decided by the two hash functions. A membership query for item x searches the possible buckets for the fingerprint of x , and returns true if an identical fingerprint is found. A cuckoo filter's fingerprint size will directly determine the false positive rate.

Use cases

Targeted ad campaigns (advertising, retail)

This application answers this question: Has the user signed up for this campaign yet?

Use a Cuckoo filter for every campaign, populated with targeted users' ids. On every visit, the user id is checked against one of the Cuckoo filters.

- If yes, the user has not signed up for campaign. Show the ad.
- If the user clicks ad and signs up, remove the user id from that Cuckoo filter.
- If no, the user has signed up for that campaign. Try the next ad/Cuckoo filter.

Discount code/coupon validation (retail, online shops)

This application answers this question: Has this discount code/coupon been used yet?

Use a Cuckoo filter populated with all discount codes/coupons. On every try, the entered code is checked against the filter.

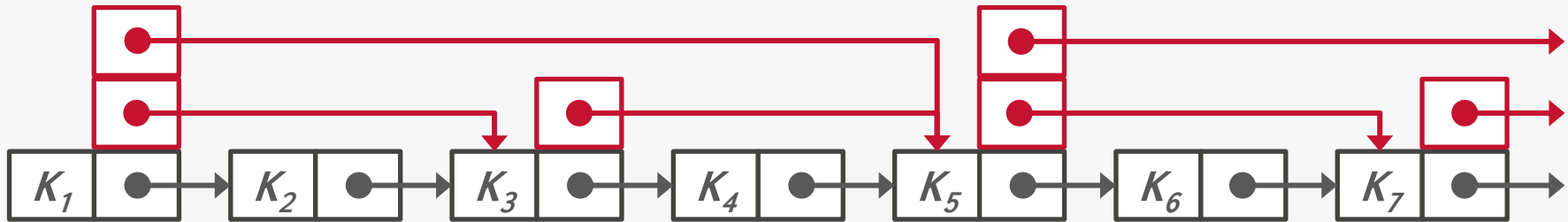
- If no, the coupon is not valid.
- If yes, the coupon can be valid. Check the main database. If valid, remove from Cuckoo filter as used.

OBSERVATION

The easiest way to implement a dynamic order-preserving index is to use a sorted linked list.

All operations have to linear search.

→ Average Cost: $O(n)$



SKIP LISTS

Multiple levels of linked lists with extra pointers to skip over entries.

- 1st level is a sorted list of all keys.
- 2nd level links every other key
- 3rd level links every fourth key
- Each level has 1/2 the keys of one below it

Maintains keys in sorted order without requiring global rebalancing.

- Approximate **O(log n)** search times.

Mostly for in-memory data structures.

- Example: LSM MemTable

Skip Lists: A Probabilistic Alternative to Balanced Trees

Skip lists are a data structure that can be used in place of balanced trees. Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

William Pugh

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that give very poor performance. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input sequence. In most cases queries must be answered on-line, so randomly permuting the input is impractical. Balanced tree algorithms re-arrange the tree as operations are performed to maintain certain balance conditions and assure good performance.

Skip lists are a probabilistic alternative to balanced trees. Skip lists are balanced by simulating a random number generator. Although skip lists have had worse case performance, no input sequence consistently produces the worst case performance (much like quicksort when the pivot element is chosen randomly). It is very unlikely a skip list data structure will be significantly unbalanced (e.g., for a dictionary of more than 250 elements, the chance that a search will take more than 3 times the expected time is less than one in a million). Skip lists have balance properties similar to that of search trees built by random insertions, yet do not require insertions to be random.

Balancing a data structure probabilistically is easier than explicitly maintaining the balance. For many applications, skip lists are a more natural representation than trees, also leading to simpler algorithms. The simplicity of skip list algorithms makes them easier to implement and provides significant constant factor speed improvements over balanced tree and self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of $1 \frac{1}{2}$ pointers per element (or even less) and do not require balance or priority information to be stored with each node.

SKIP LISTS

We might need to examine every node of the list when searching a linked list (Figure 1a). If the list is stored in sorted order and every other node of the list also has a pointer to the node two ahead in the list (Figure 1b), we have to examine no more than $\lceil n/2 \rceil + 1$ nodes (where n is the length of the list).

Also giving every fourth node a pointer four ahead (Figure 1c) requires that no more than $\lceil n/4 \rceil + 2$ nodes be examined. If every 2^k node has a pointer 2^k nodes ahead (Figure 1d), the number of nodes that must be examined can be reduced to $\lceil \log_2 n \rceil + 1$ while only doubling the number of pointers. This data structure could be used for fast searching, but insertion and deletion would be impractical.

A node that has k forward pointers is called a level k node. If every 2^k node has a pointer 2^k nodes ahead, then levels of nodes are distributed in a simple pattern: 50% are level 1, 25% are level 2, 12.5% are level 3 and so on. What would happen if the levels of nodes were chosen randomly, but in the same proportions (e.g., as in Figure 1e)? A node's k^{th} forward pointer, instead of pointing 2^k nodes ahead, points to the next node of level k or higher. Insertions or deletions would require only local modifications: the level of a node, chosen randomly when the node is inserted, need never change. Some arrangements of levels would give poor execution times, but we will see that such arrangements are rare. Because these data structures are linked lists with extra pointers that skip over intermediate nodes, I named them *skip lists*.

SKIP LIST ALGORITHMS

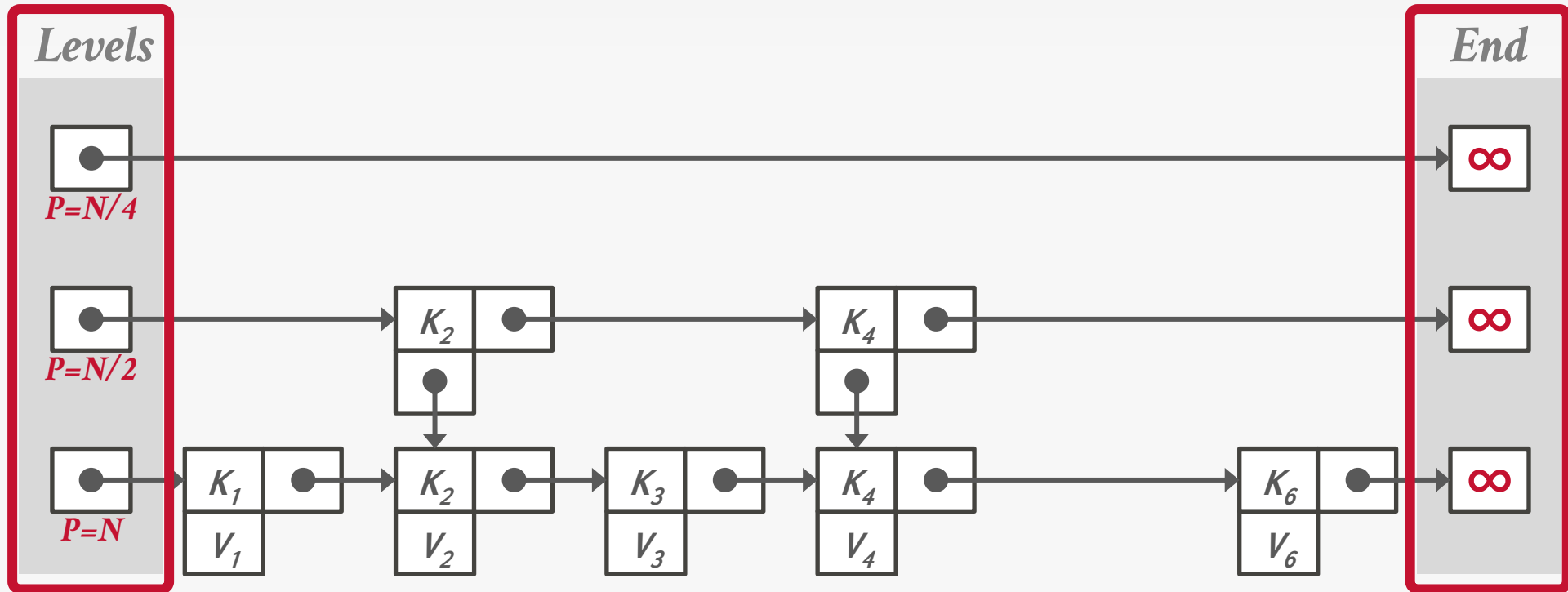
This section gives algorithms to search for, insert and delete elements in a dictionary or symbol table. The *Search* operation returns the contents of the value associated with the desired key or *failure* if the key is not present. The *Insert* operation associates a specified key with a new value (inserting the key if it had not already been present). The *Delete* operation deletes the specified key. It is easy to support additional operations such as "find the minimum key" or "find the next key".

Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A level k node has k forward pointers, indexed 1 through k . We do not need to store the level of a node in the node. Levels are capped at some appropriate constant *MaxLevel*. The level of a list is the maximum level currently in the list (or 1 if the list is empty). The header of a list has forward pointers at levels one through *MaxLevel*. The forward pointers of the header at levels higher than the current maximum level of the list point to NIL.



SKIP LISTS: INSERT

Insert K_5

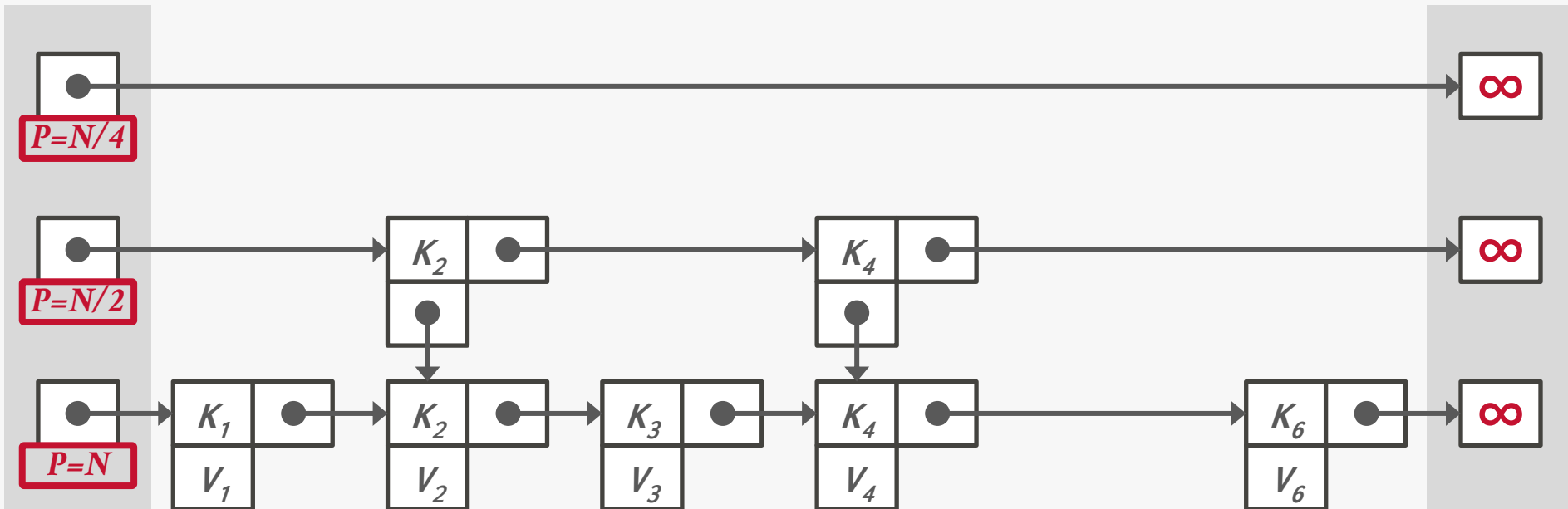


SKIP LISTS: INSERT

Insert K_5

Levels

End

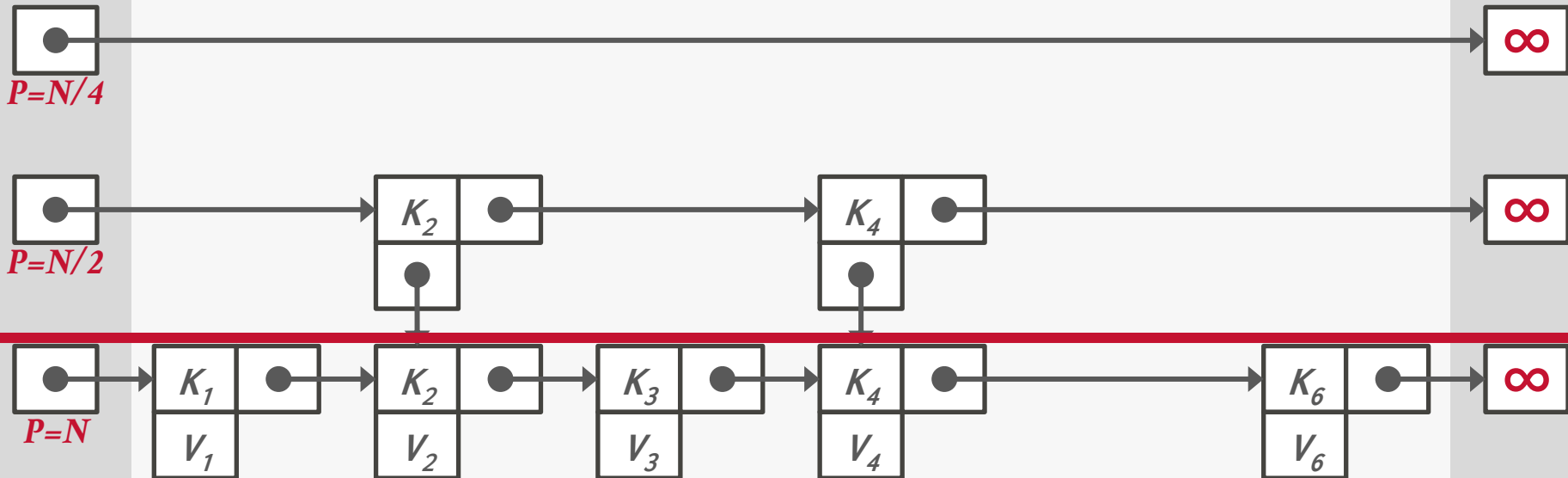


SKIP LISTS: INSERT

Insert K_5

Levels

End

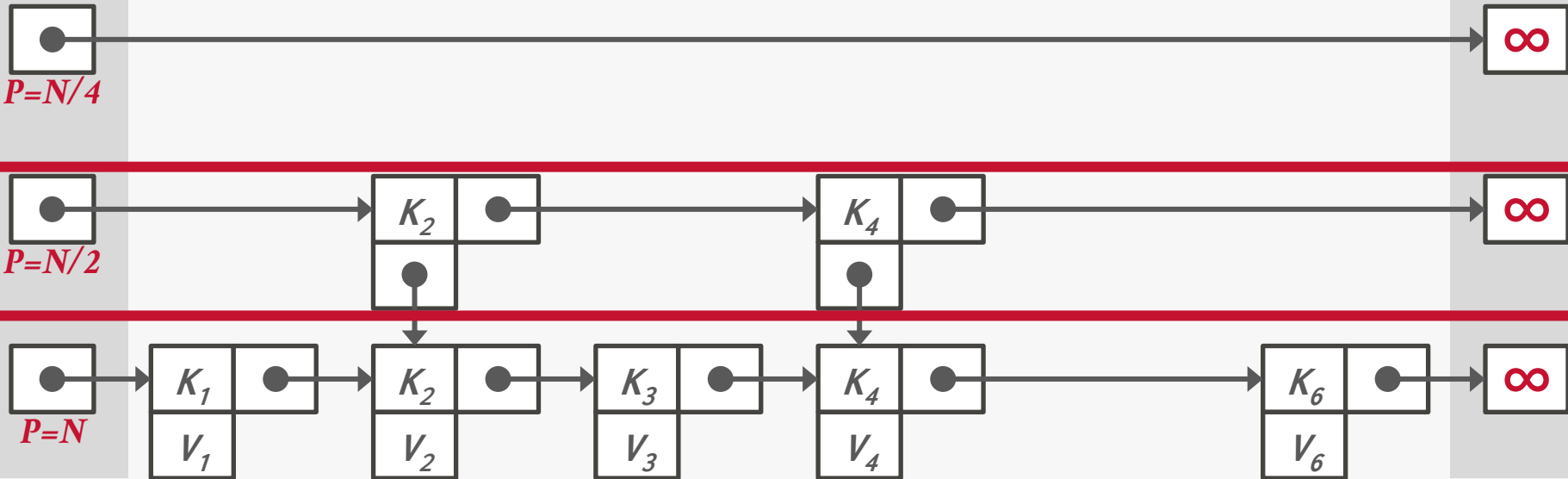


SKIP LISTS: INSERT

Insert K_5

Levels

End



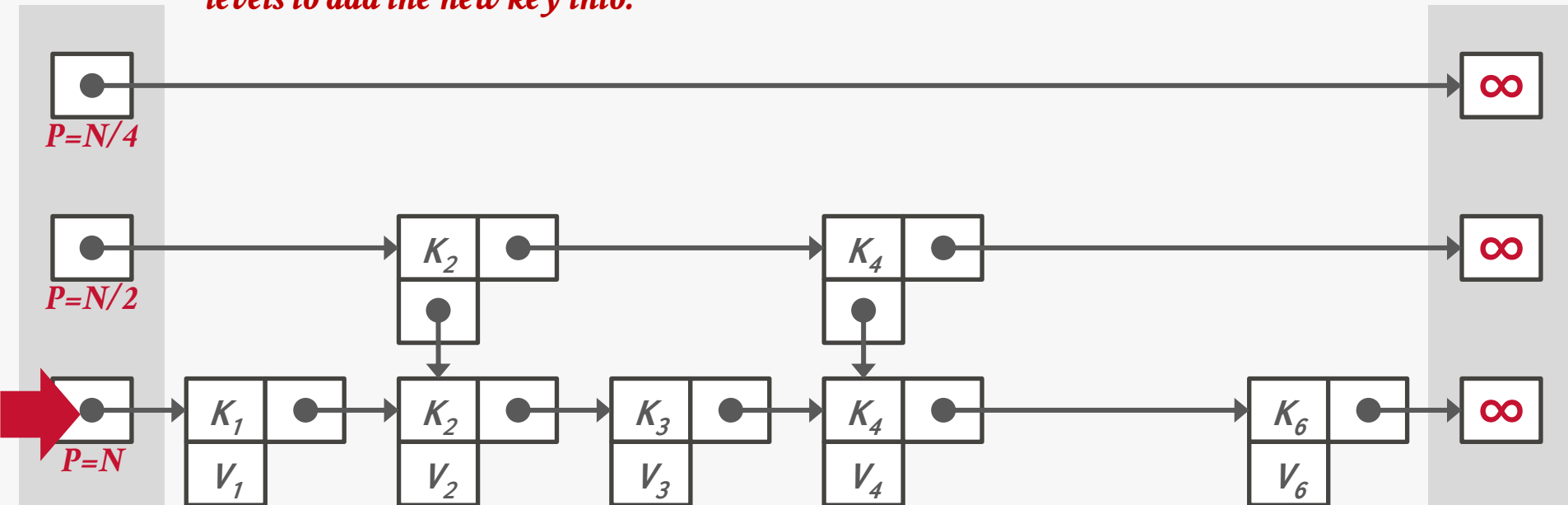
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End



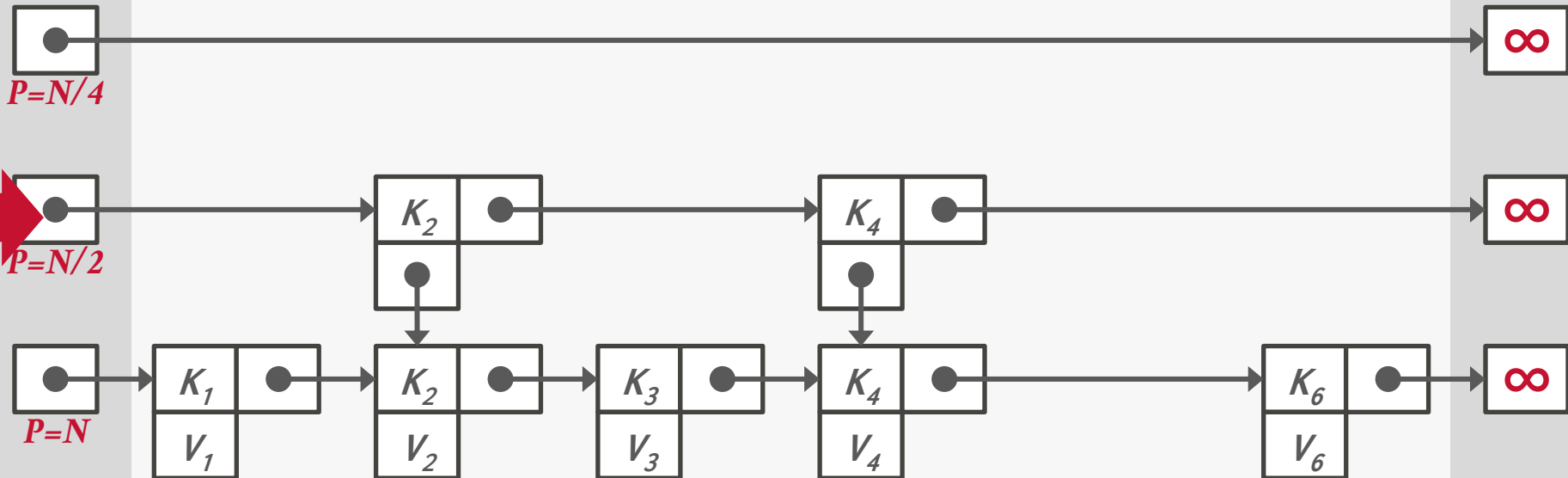
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End



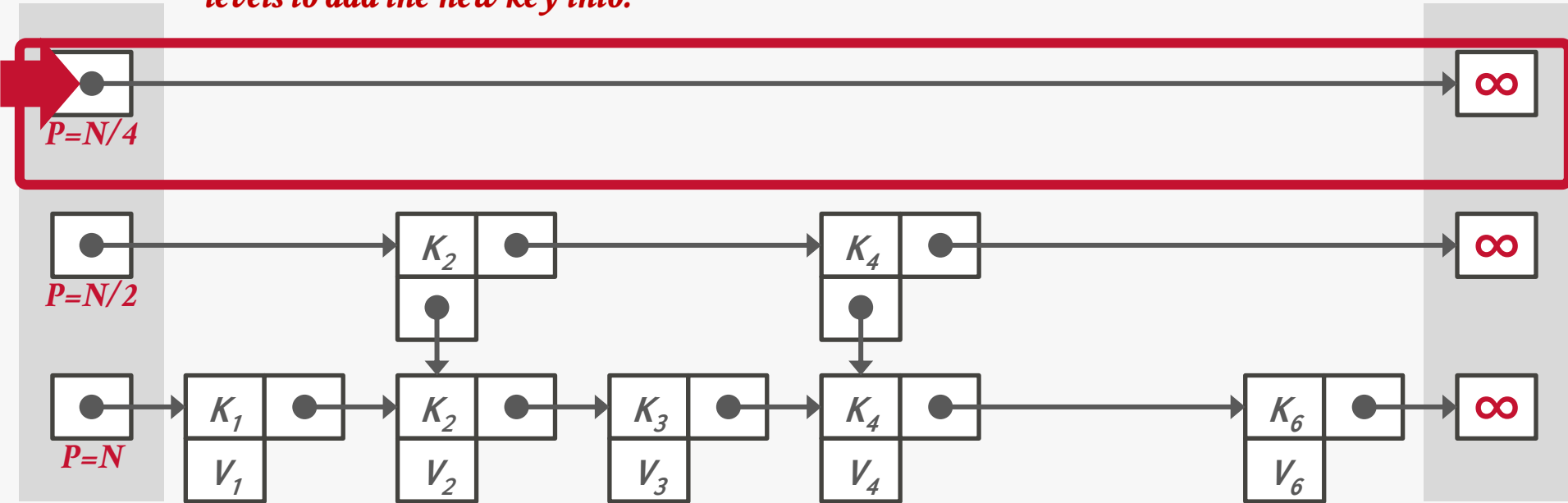
SKIP LISTS: INSERT

Insert K_5

Levels

Flip a coin to decide how many levels to add the new key into.

End



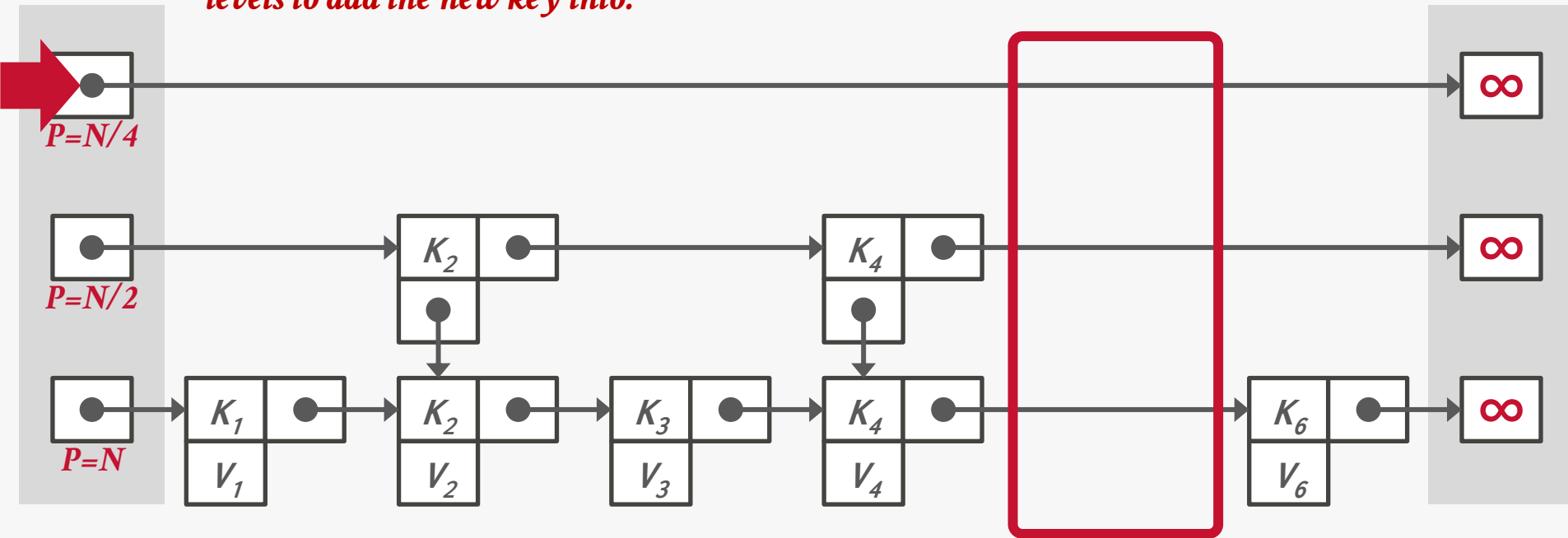
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End



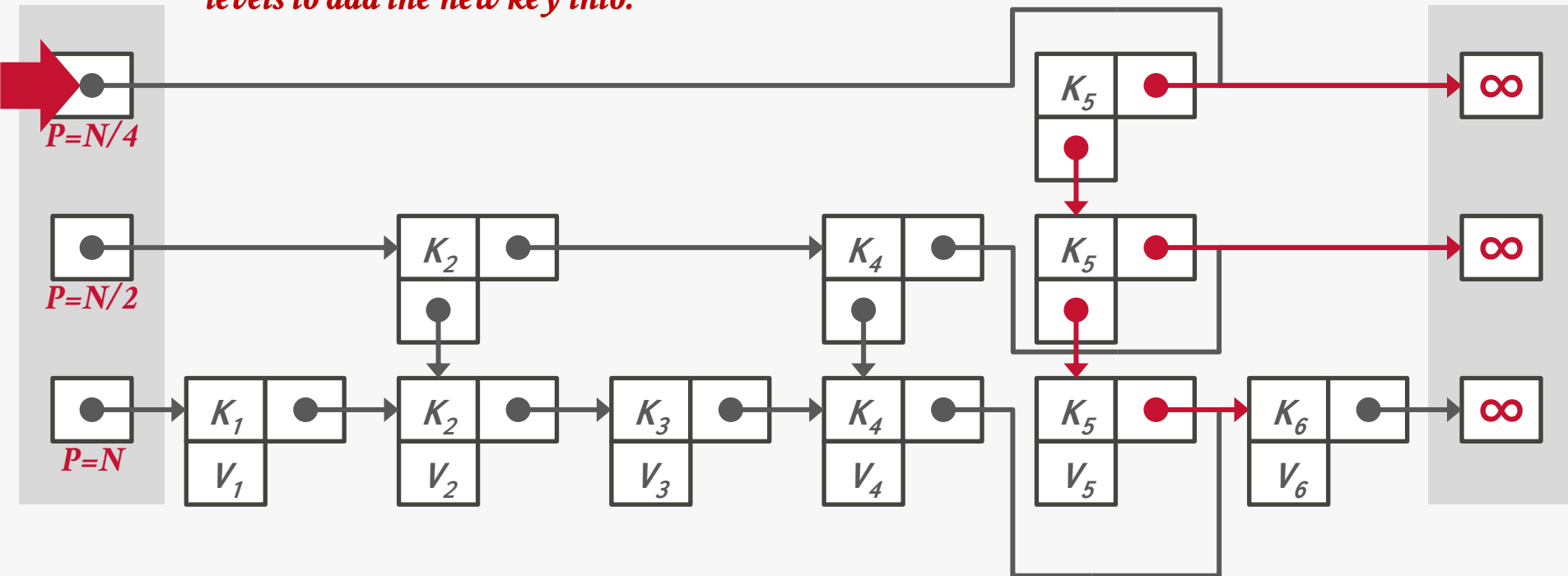
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End



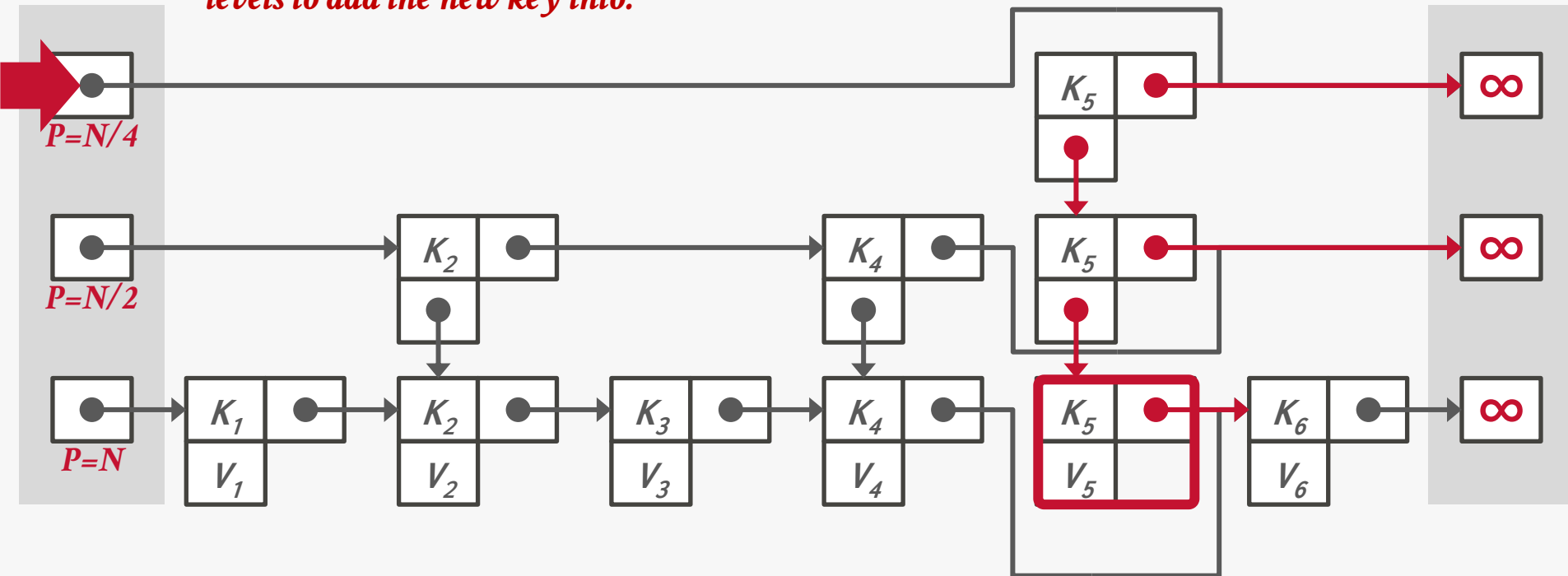
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End



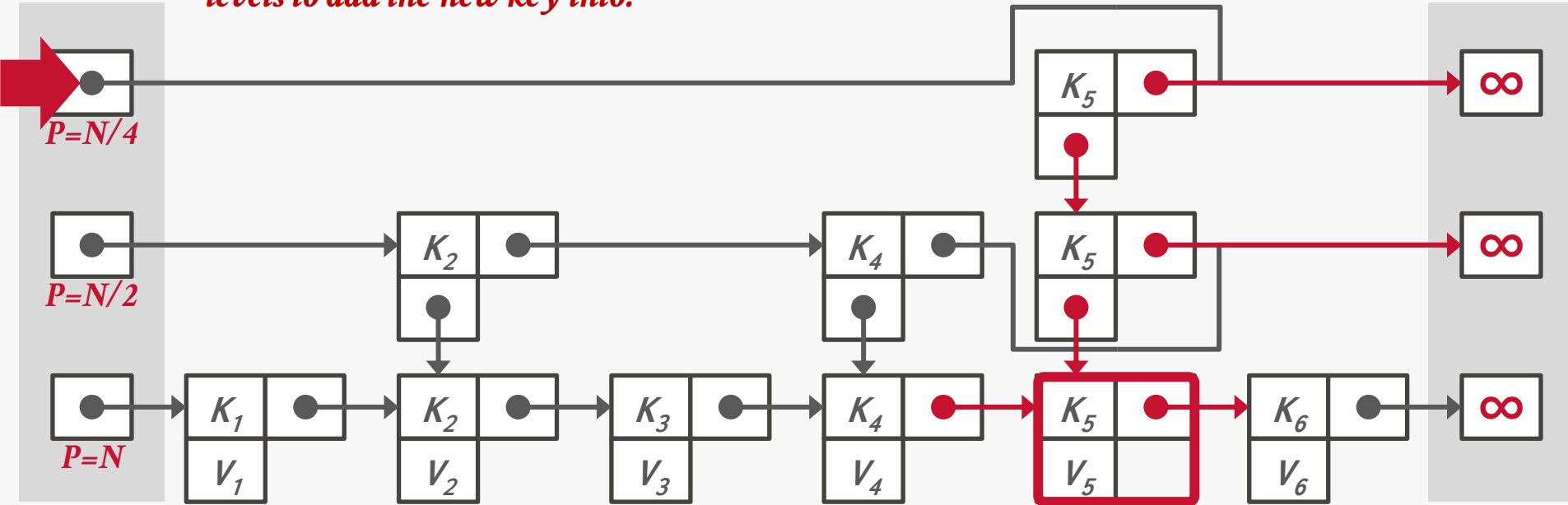
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End



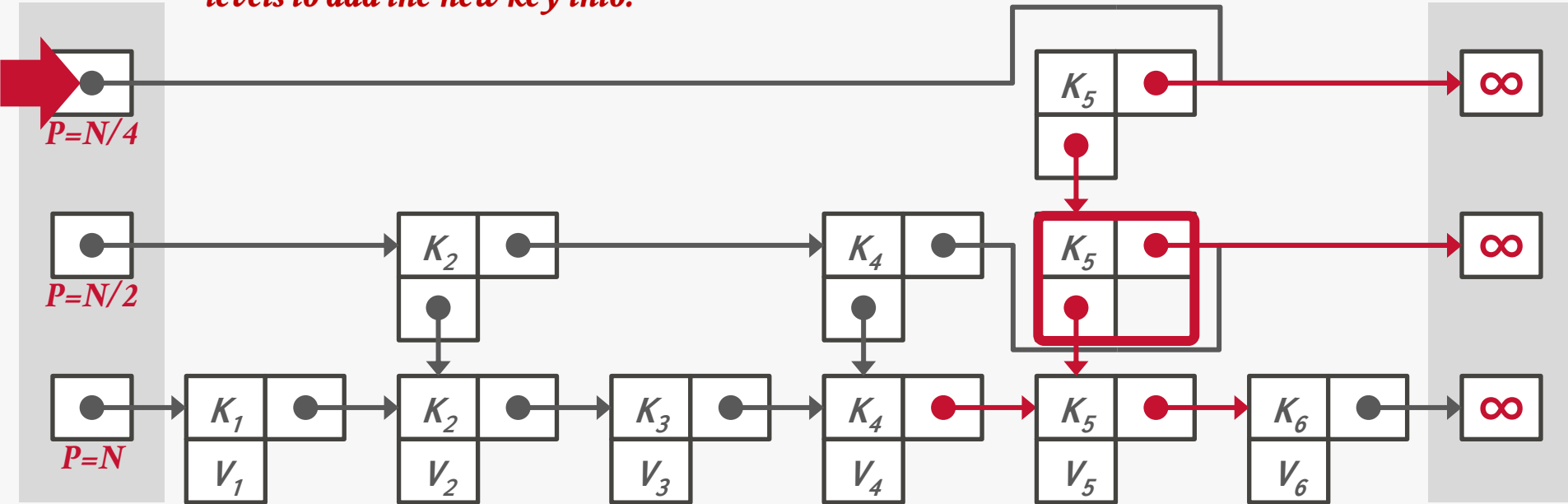
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End



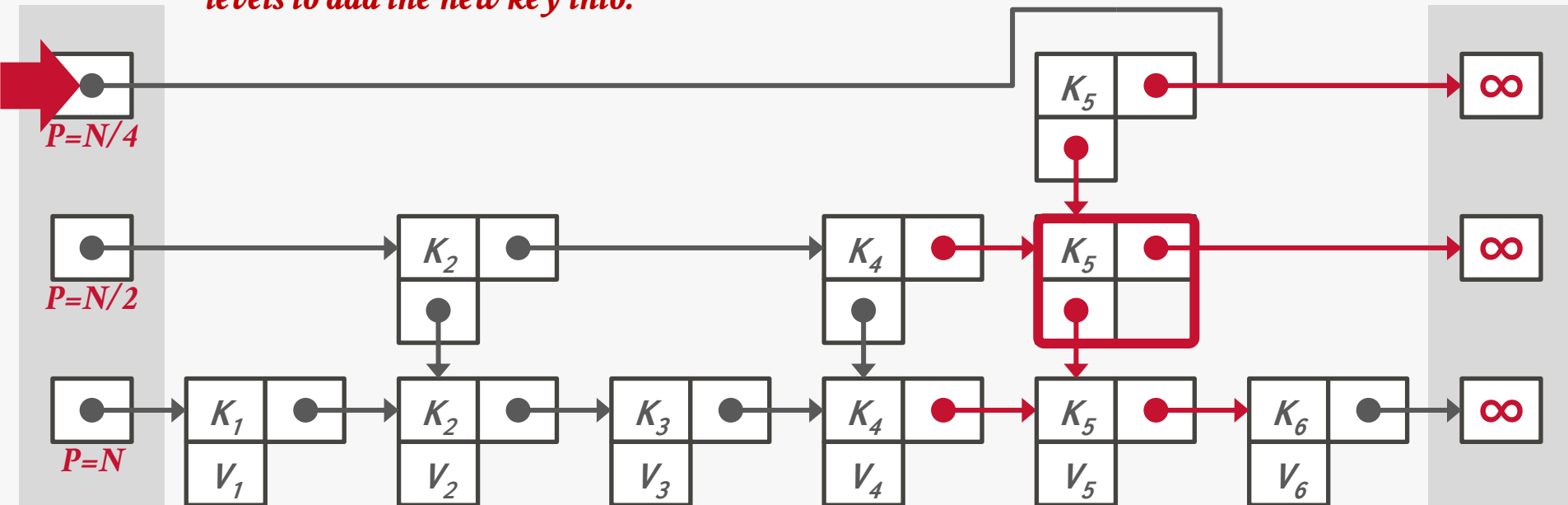
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End



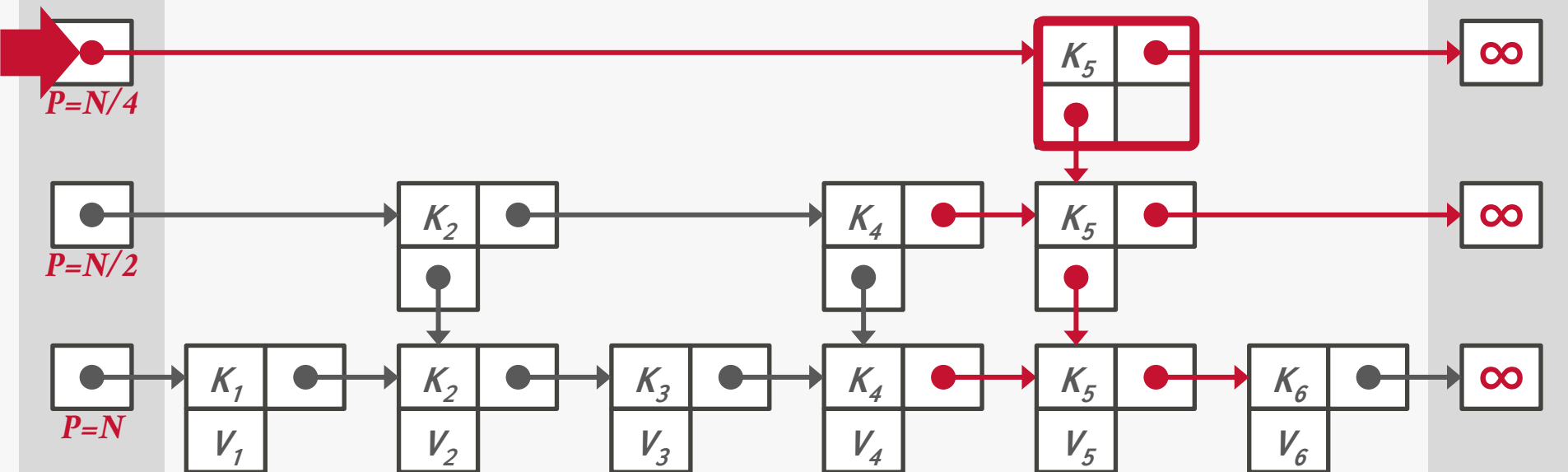
SKIP LISTS: INSERT

Insert K_5

Flip a coin to decide how many levels to add the new key into.

Levels

End

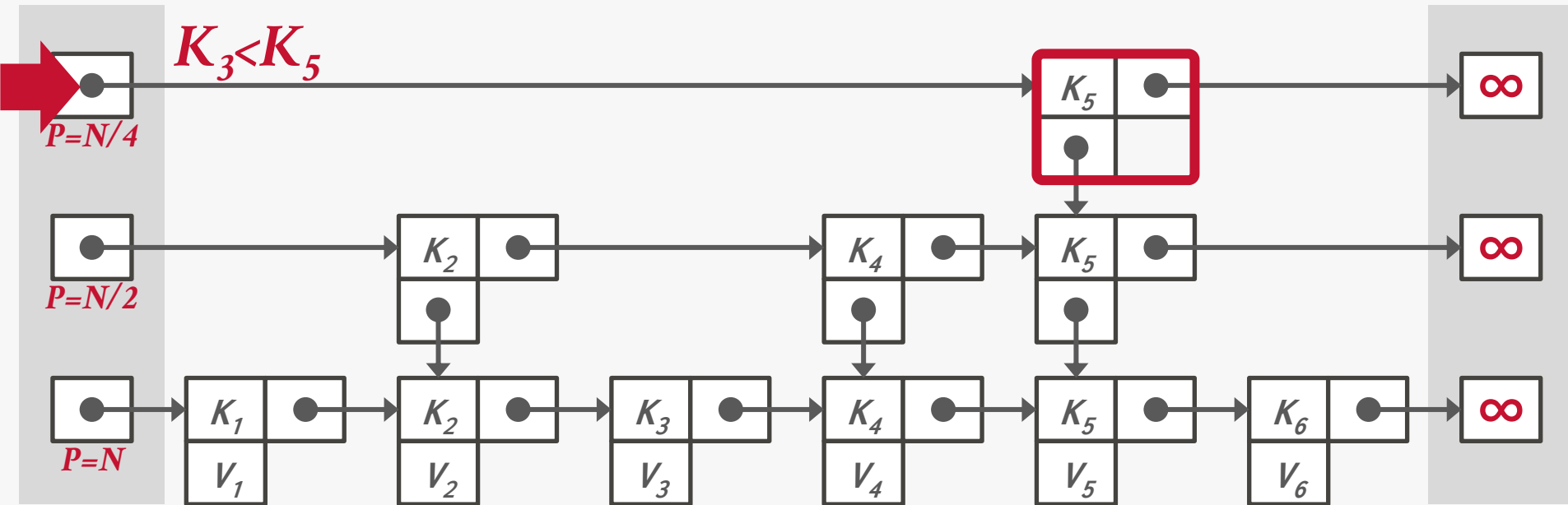


SKIP LISTS: SEARCH

Find K_3

Levels

End

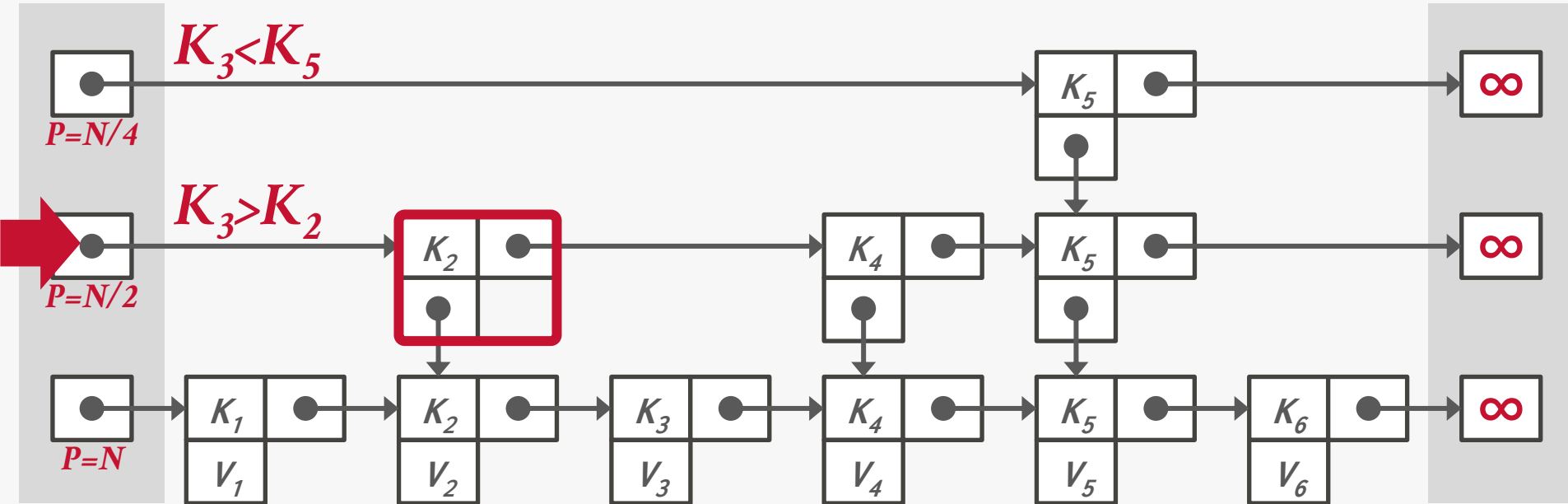


SKIP LISTS: SEARCH

Find K_3

Levels

End

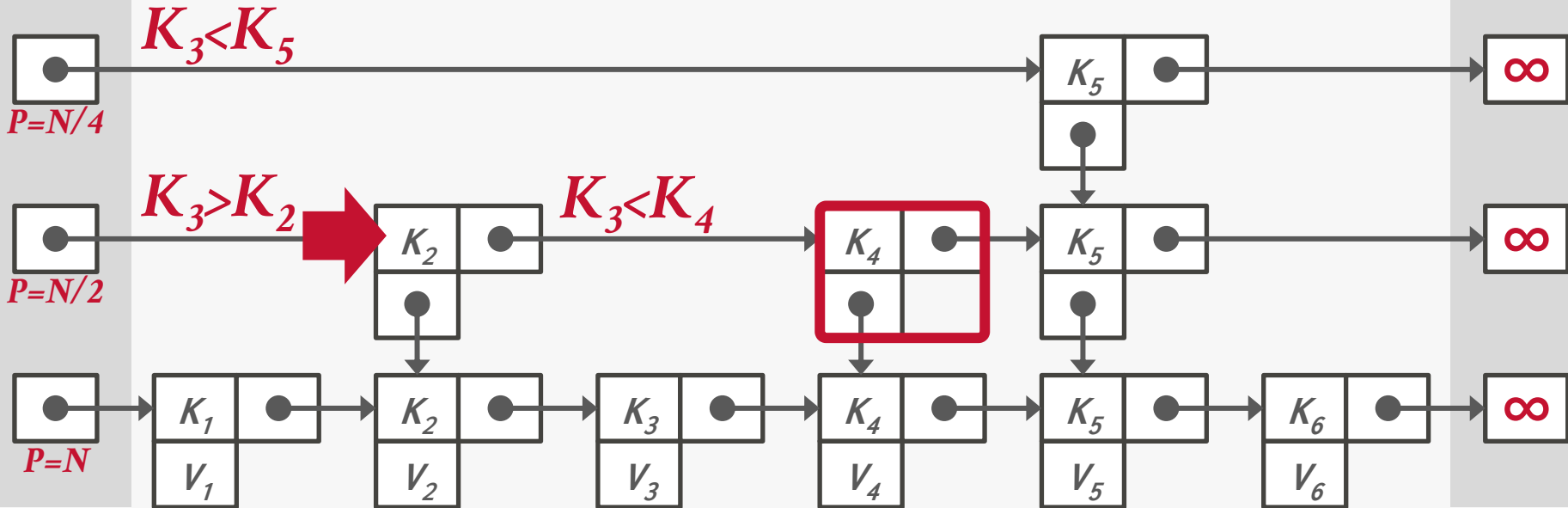


SKIP LISTS: SEARCH

Find K_3

Levels

End

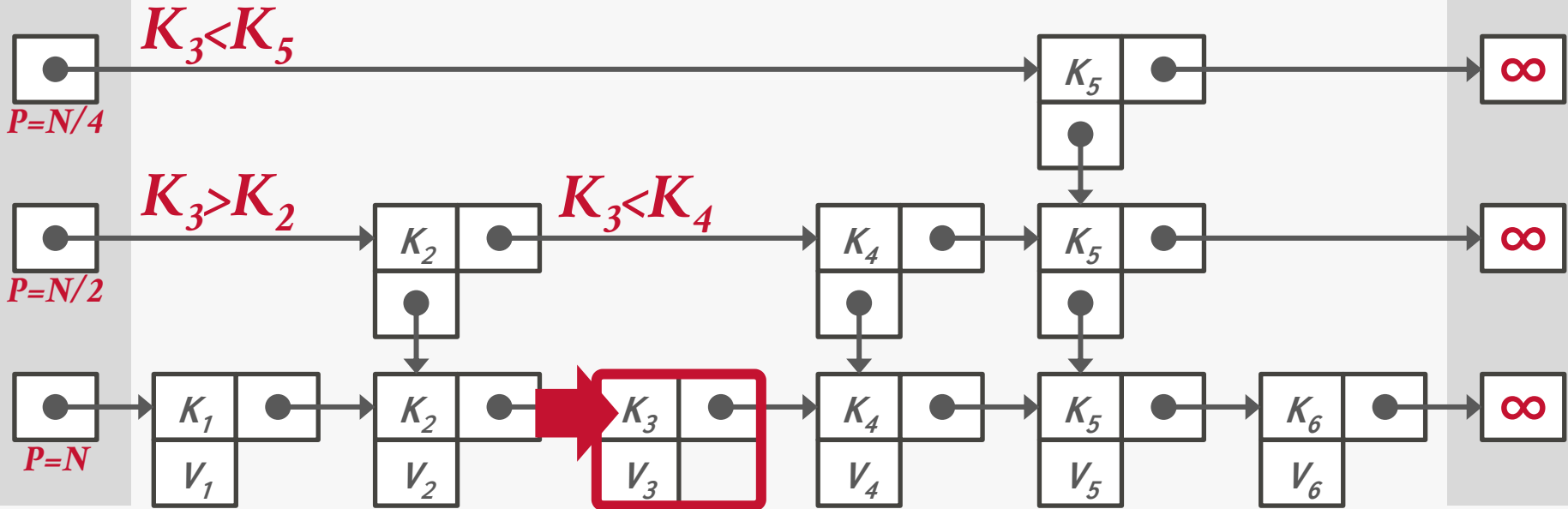


SKIP LISTS: SEARCH

Find K_3

Levels

End



SKIP LISTS: DELETE

First **logically** remove a key from the index by setting a flag to tell threads to ignore.

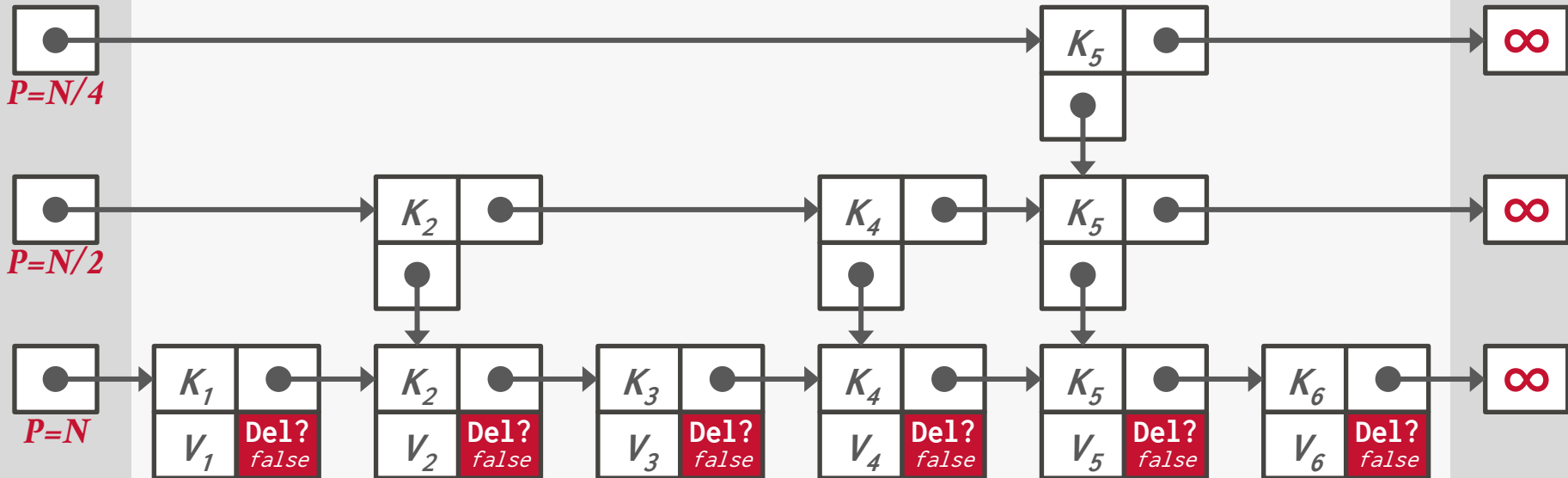
Then **physically** remove the key once we know that no other thread is holding the reference.

SKIP LISTS: DELETE

Delete K_5

Levels

End

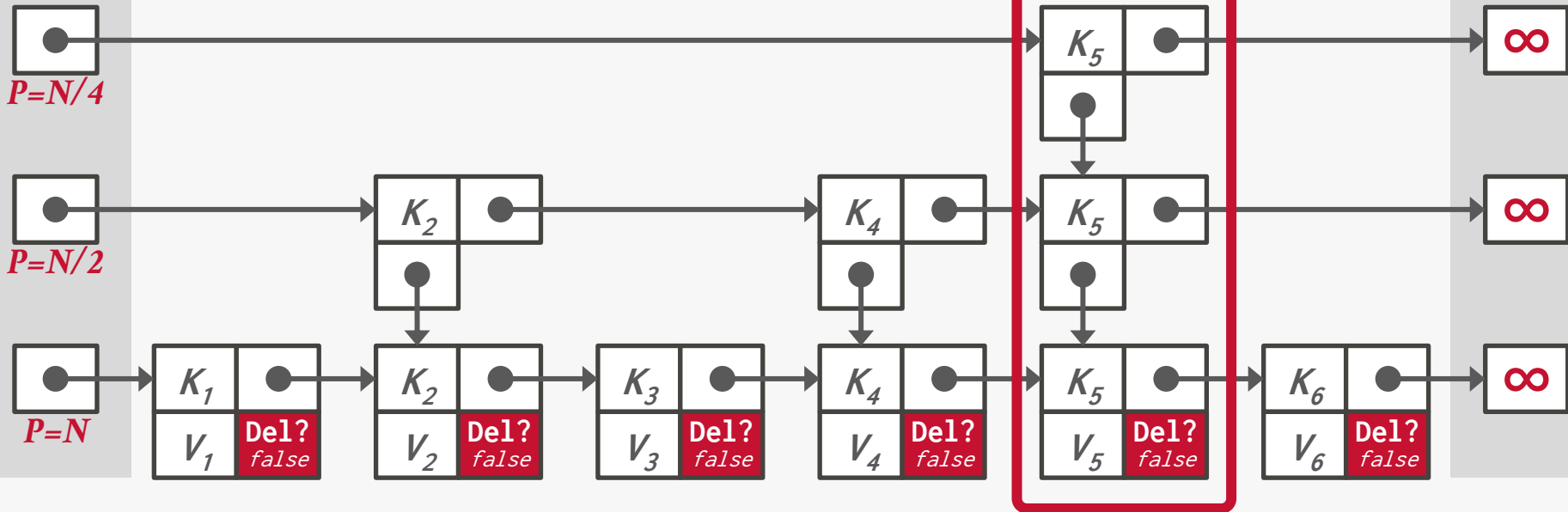


SKIP LISTS: DELETE

Delete K_5

Levels

End

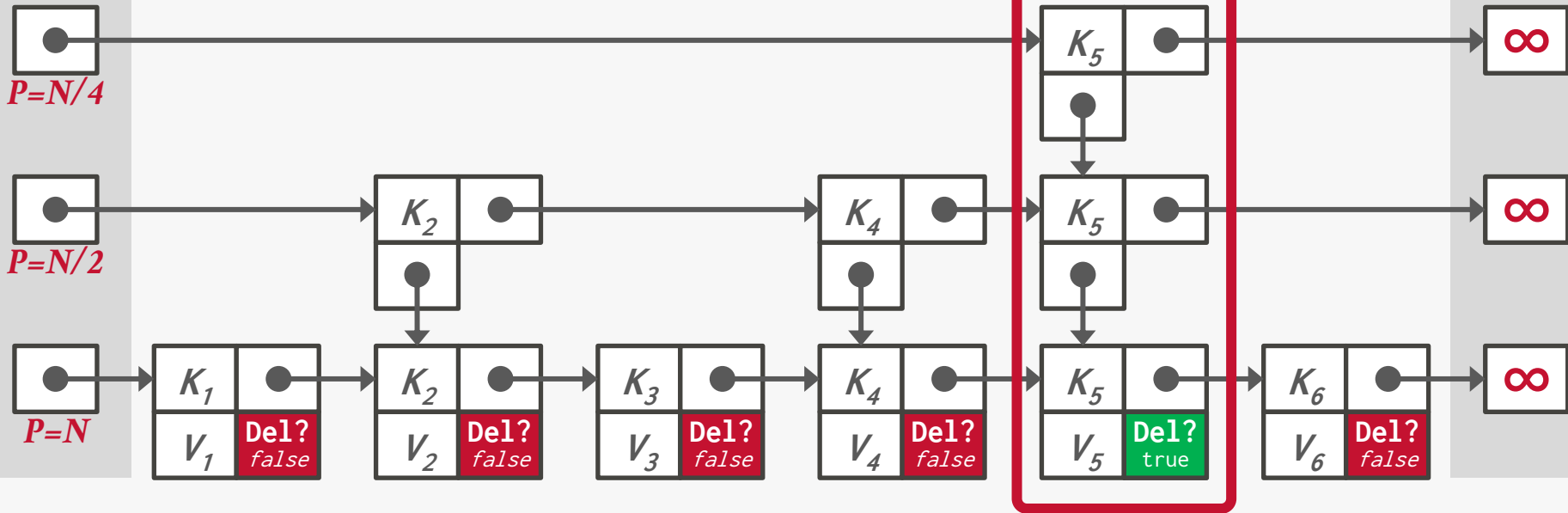


SKIP LISTS: DELETE

Delete K_5

Levels

End

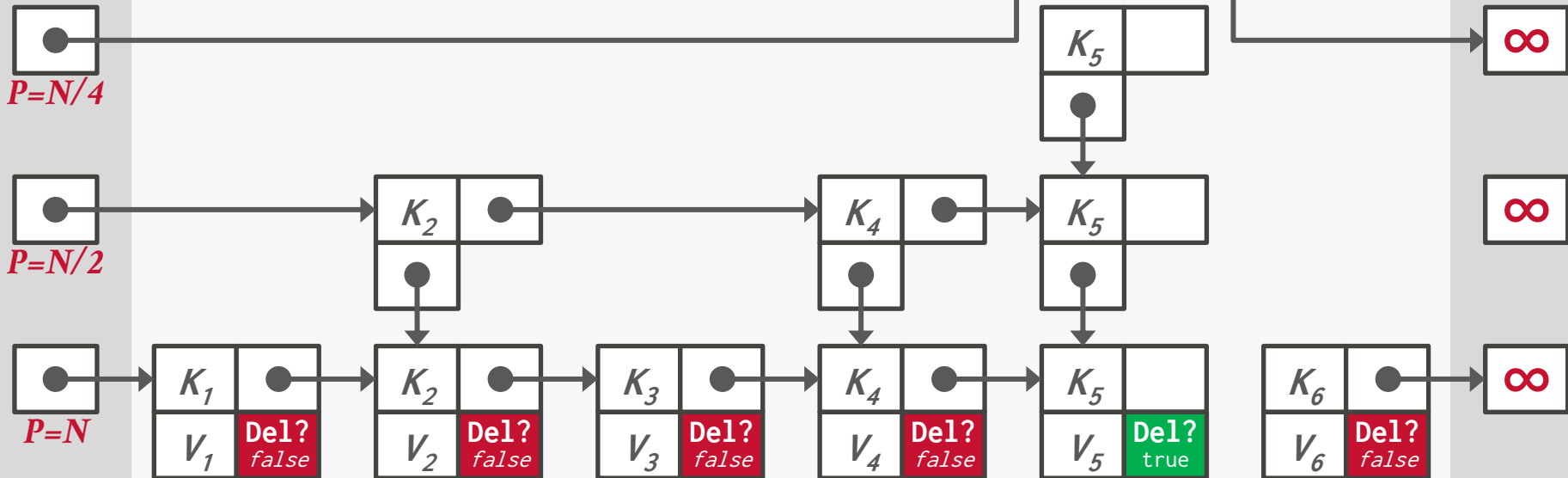


SKIP LISTS: DELETE

Delete K_5

Levels

End

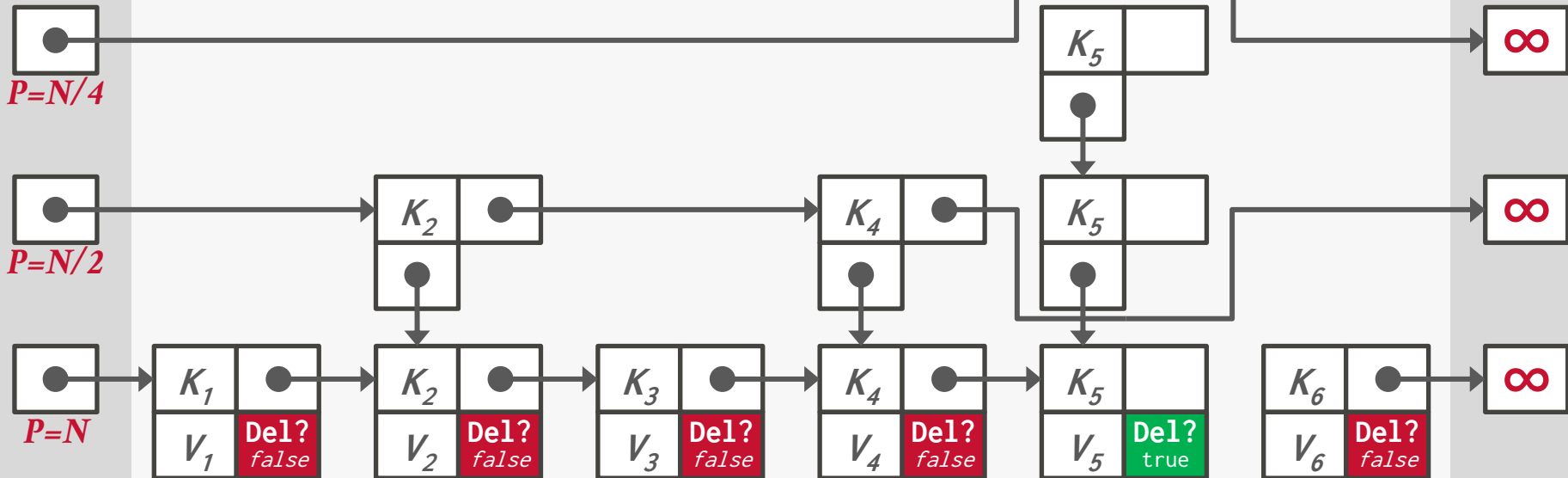


SKIP LISTS: DELETE

Delete K_5

Levels

End

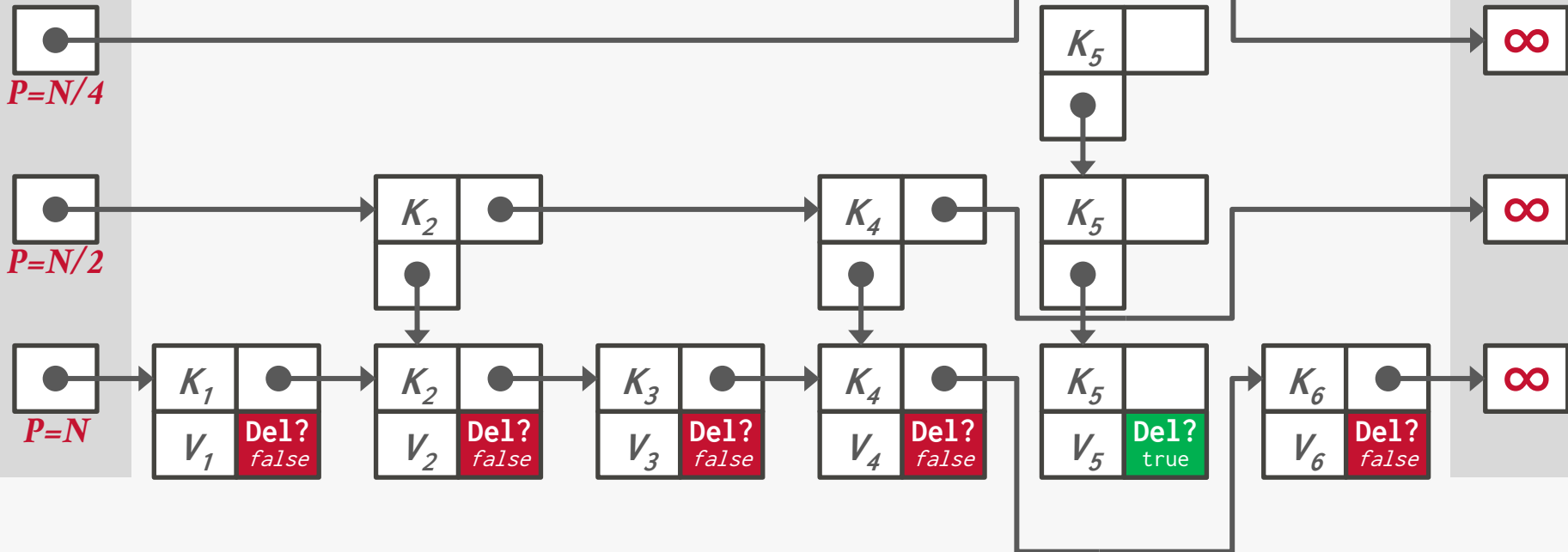


SKIP LISTS: DELETE

Delete K_5

Levels

End

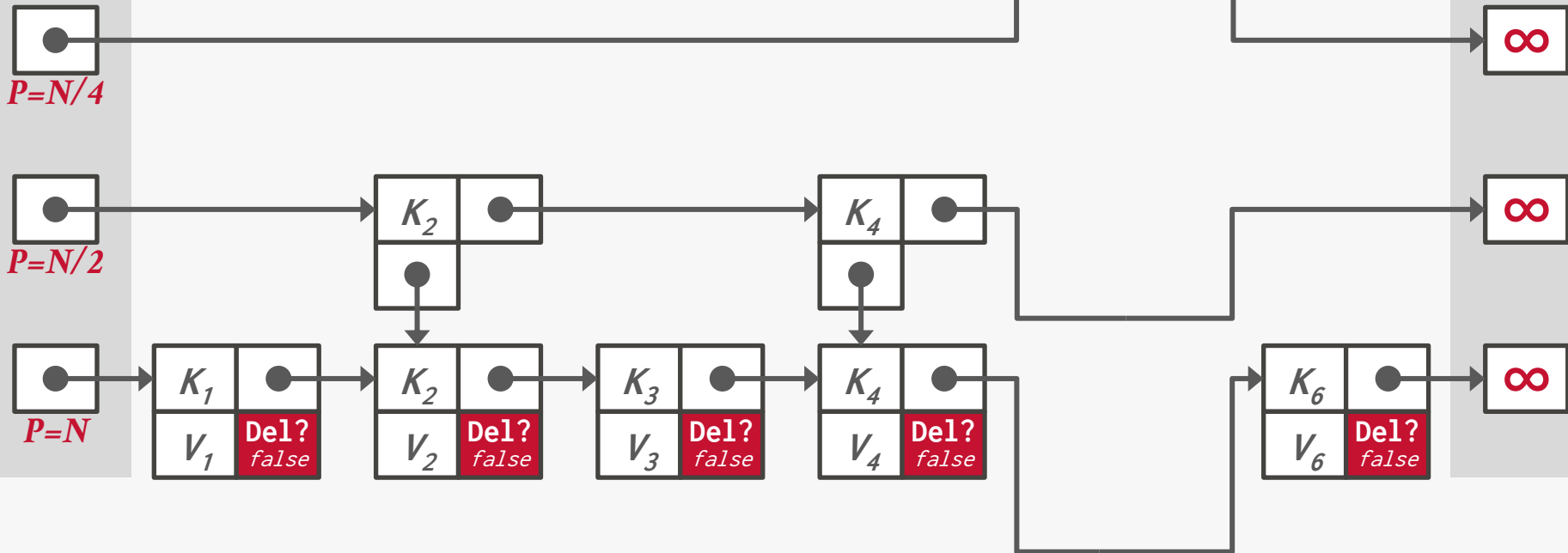


SKIP LISTS: DELETE

Delete K_5

Levels

End



SKIP LISTS

Advantages:

- May use less memory than a B+Tree, if you do not include reverse pointers.
- Insertions and deletions do not require rebalancing.

Disadvantages:

- Not disk/cache friendly because they do not optimize locality of references.
- Reverse search is non-trivial.

OBSERVATION

The inner node keys in a B+Tree cannot tell you whether a key exists in the index. You must always traverse to the leaf node.

This means that you could have (at least) one buffer pool page miss per level in the tree just to find out a key does not exist.

TRIE INDEX

Use a digital representation of keys to examine prefixes one-by-one.

→ aka *Digital Search Tree*, *Prefix Tree*.

Shape depends on keys and lengths.

→ Does not depend on existing keys or insertion order.

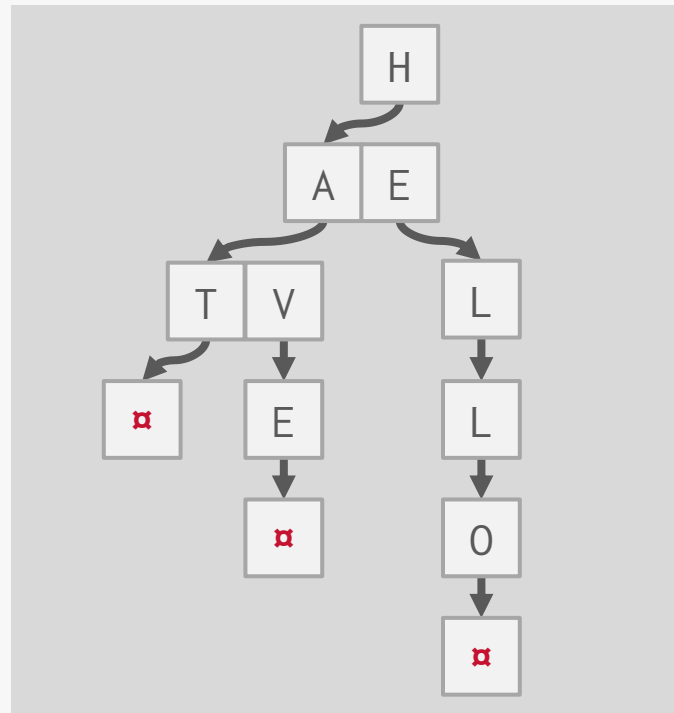
→ Does not require rebalancing operations.

All operations have $O(k)$ complexity where k is the length of the key.

→ Path to a leaf node represents a key.

→ Keys are stored implicitly and can be reconstructed from paths.

Keys: HELLO, HAT, HAVE



TRIE INDEX

Use a digital representation of keys to examine prefixes one-by-one.

→ aka *Digital Search Tree*, *Prefix Tree*.

Shape depends on keys and lengths.

→ Does not depend on existing keys or insertion order.

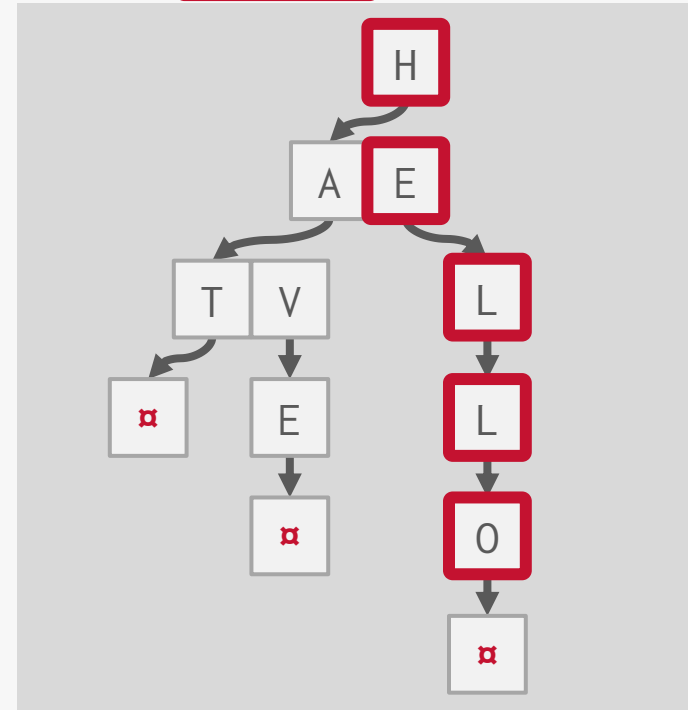
→ Does not require rebalancing operations.

All operations have $O(k)$ complexity where k is the length of the key.

→ Path to a leaf node represents a key.

→ Keys are stored implicitly and can be reconstructed from paths.

Keys: HELLO, HAT, HAVE



TRIE KEY SPAN

The **span** of a trie level is the number of bits that each partial key / digit represents.

→ If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

This determines the **fan-out** of each node and the physical **height** of the tree.

→ n -way Trie = Fan-Out of n

TRIE KEY SPAN

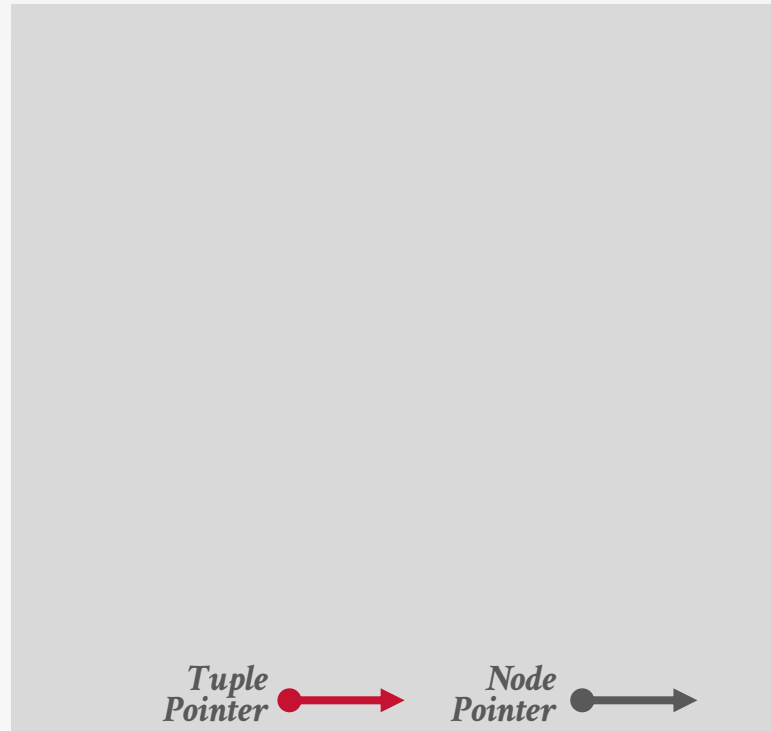
Keys: K10, K25, K31

K10 → 000000000 00001010

K25 → 000000000 00011001

K31 → 000000000 00011111

1-bit Span Trie



TRIE KEY SPAN

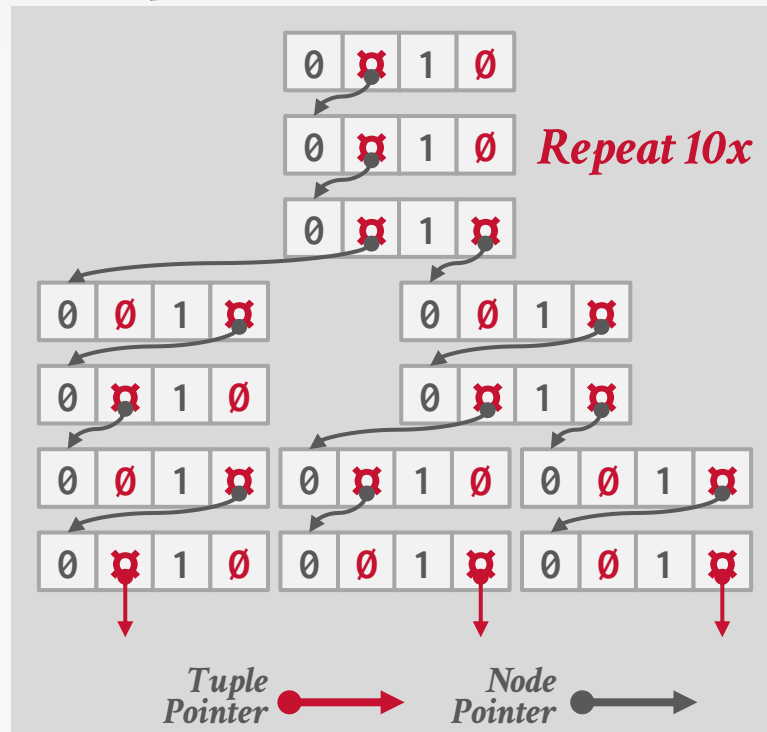
Keys: K10, K25, K31

K10 → 000000000 00001010

K25 → 000000000 00011001

K31 → 000000000 00011111

1-bit Span Trie

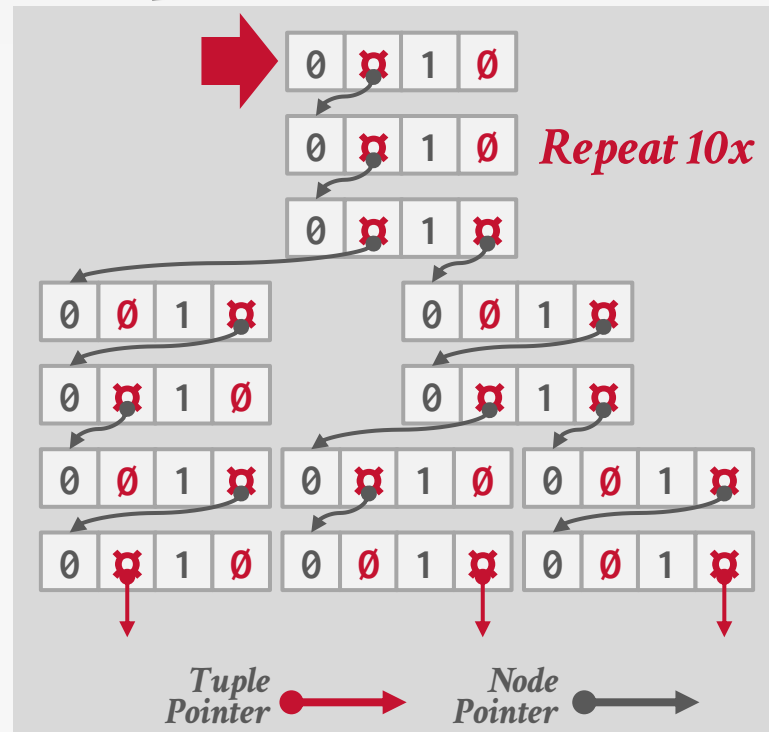


TRIE KEY SPAN

Keys: K10, K25, K31

K10 → 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
 K25 → 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1
 K31 → 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1

1-bit Span Trie



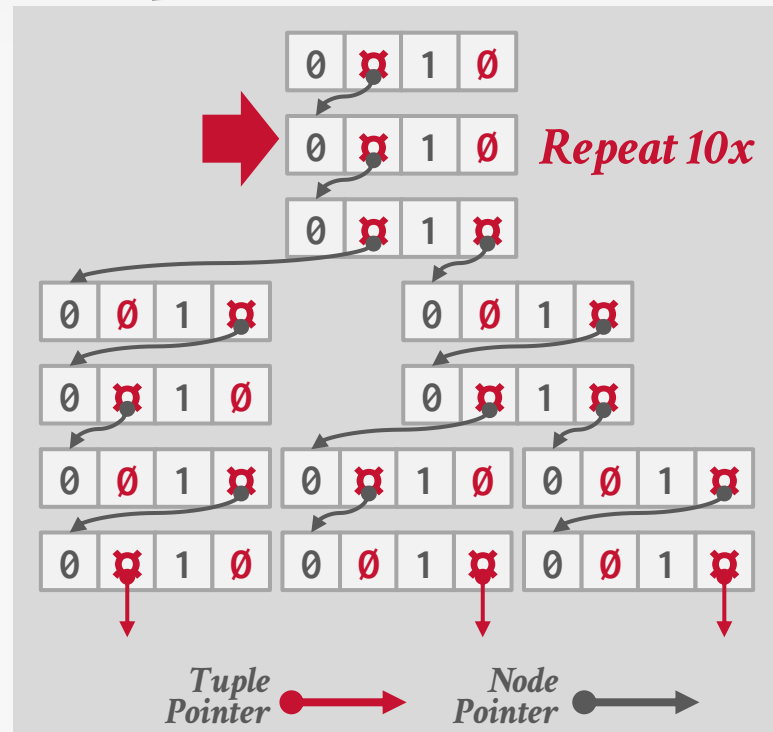
TRIE KEY SPAN

Keys: K10, K25, K31

↓

K10 → 00000000 00001010
 K25 → 00000000 00011001
 K31 → 00000000 00011111

1-bit Span Trie



TRIE KEY SPAN

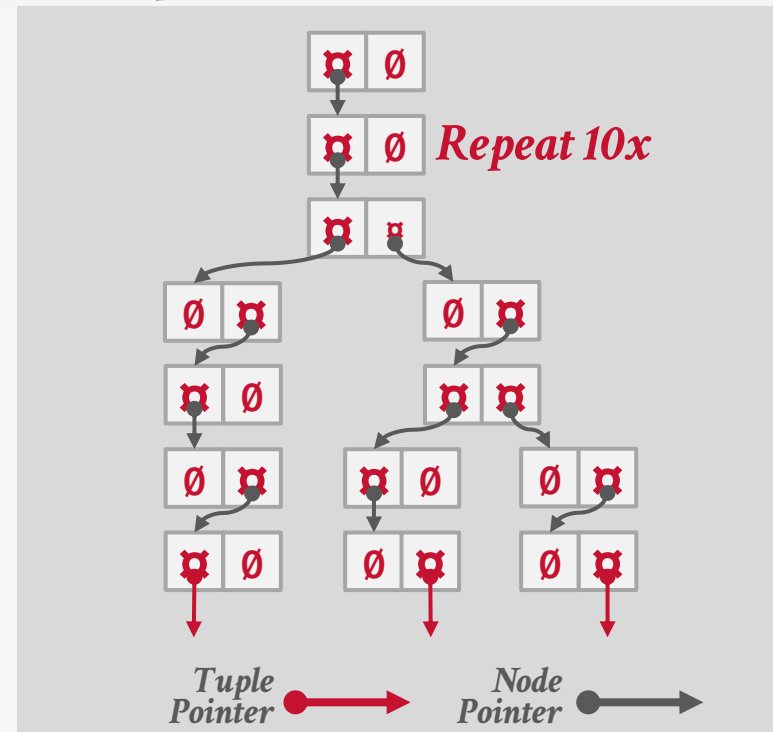
Keys: K10, K25, K31

K10 → 00000000 00001010

K25 → 00000000 00011001

K31 → 00000000 00011111

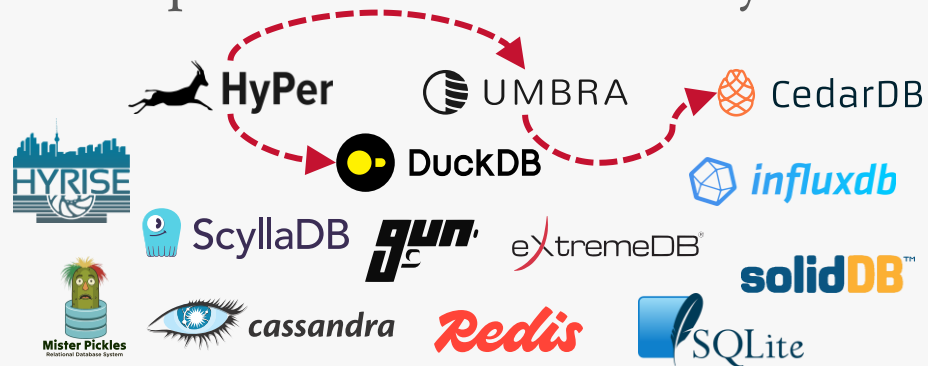
1-bit Span Trie



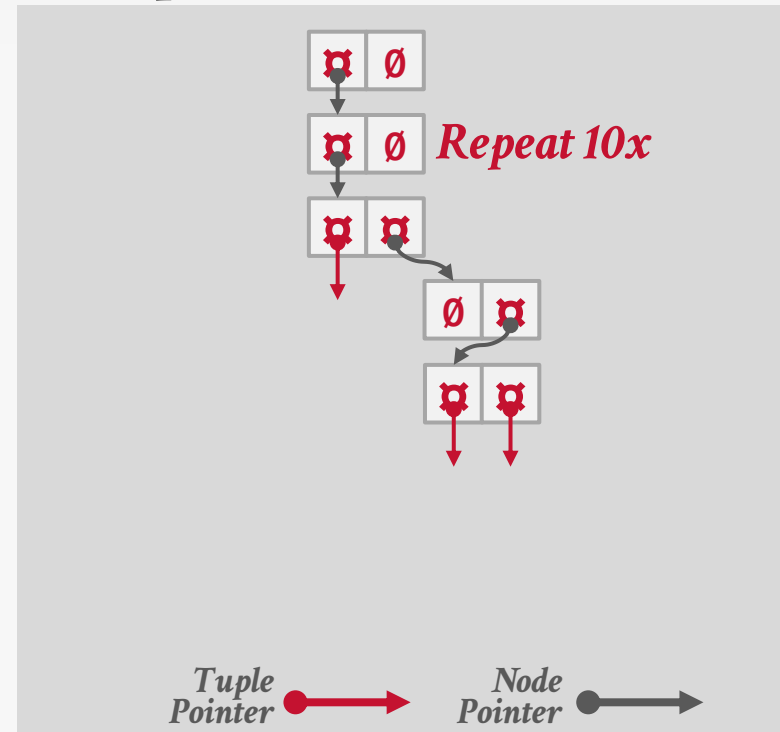
RADIX TREE

Vertically compressed trie that
 compacts nodes with a single child.
 → aka **Patricia Trees** (2-way tries)

Can produce false positives, so the
 DBMS always checks the original
 tuple to see whether a key matches.



1-bit Span Radix Tree

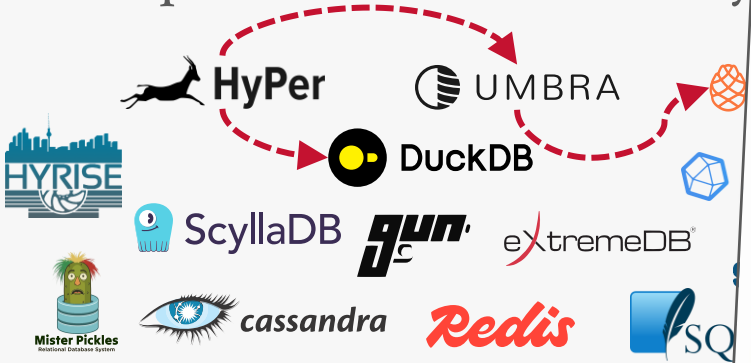


RA



Vertically compressed trie
compacts nodes with a single
→ aka Patricia Trees (2-way trie)

Can produce false positives
DBMS always checks the
tuple to see whether a key



Apache Cassandra 5.0 Features: Trie Memtables and Trie-Indexed SSTables

November 9, 2023 | Branimir Lambov

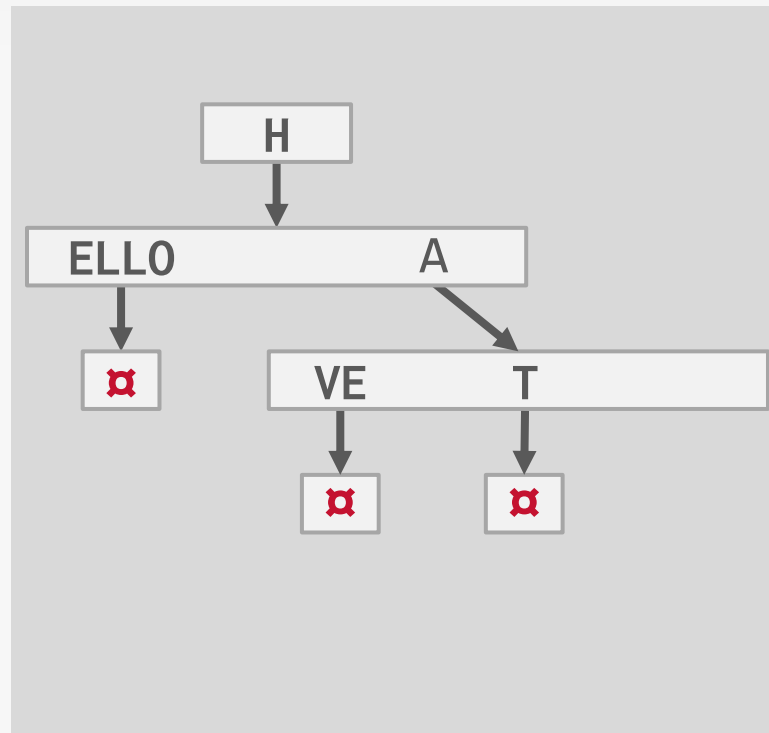
Apache Cassandra 5.0 is the project's major release for 2023, and it promises some of the biggest changes for Cassandra to-date. After more than a decade of world class engineering building Cassandra as the safest most stable distributed database, we are witness now to a new chapter of innovation introducing a host of exciting features and enhancements that empower users to take their data-driven applications to the next level - including machine learning and artificial intelligence.

This blog series aims to give a deeper dive into some of the key features of Cassandra 5.0.

Apache Cassandra is a widely used open-source NoSQL database known for its ability to handle large volumes of data across distributed clusters. But while it has gained popularity for its robust distributed architecture...

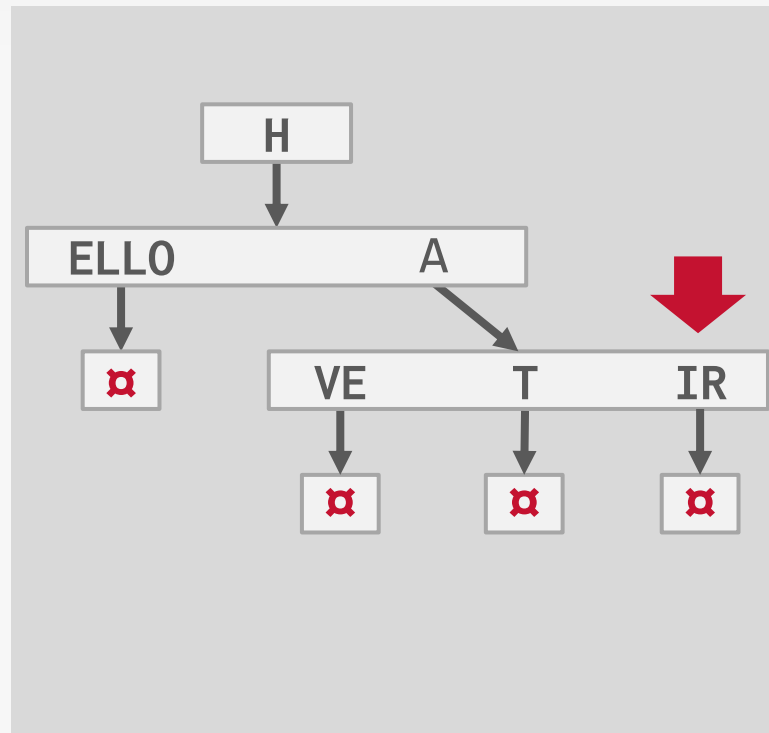
RADIX TREE: MODIFICATIONS

Insert HAIR



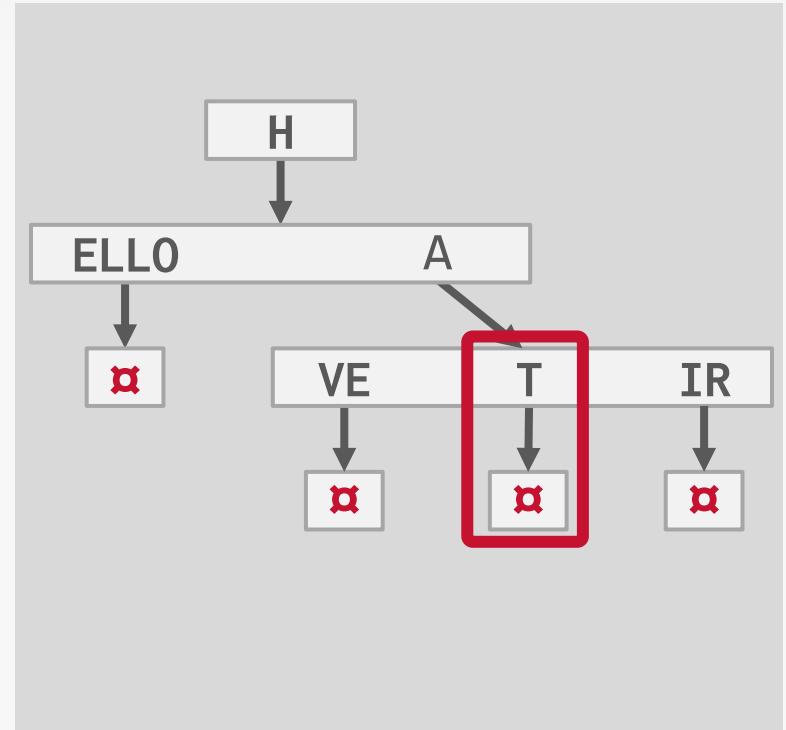
RADIX TREE: MODIFICATIONS

Insert HAIR



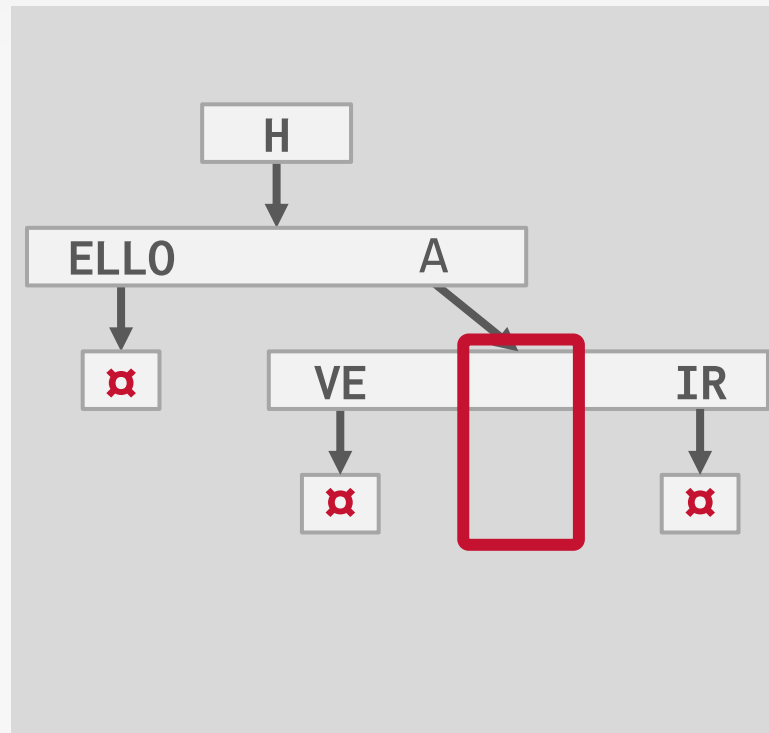
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT



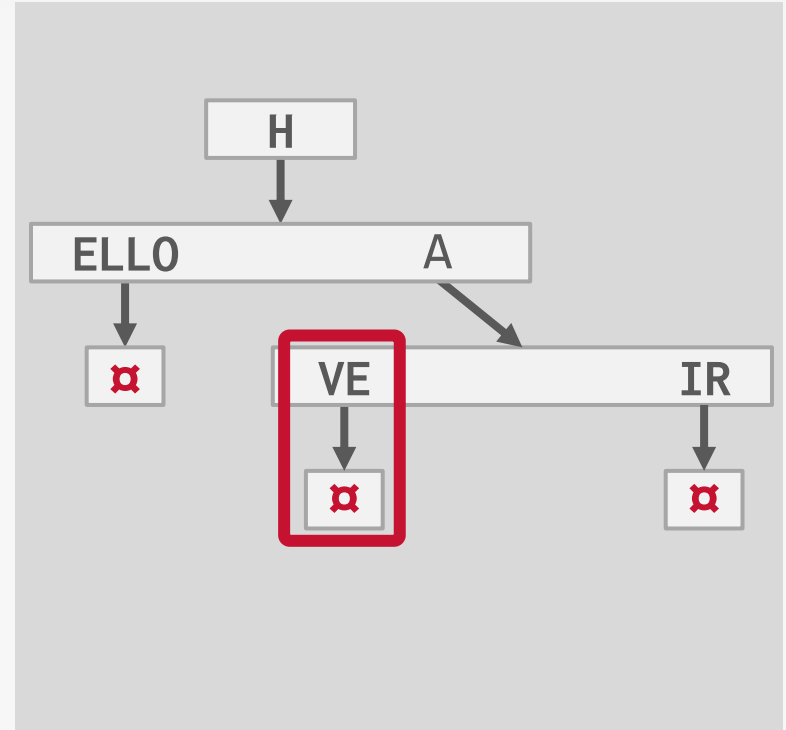
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT



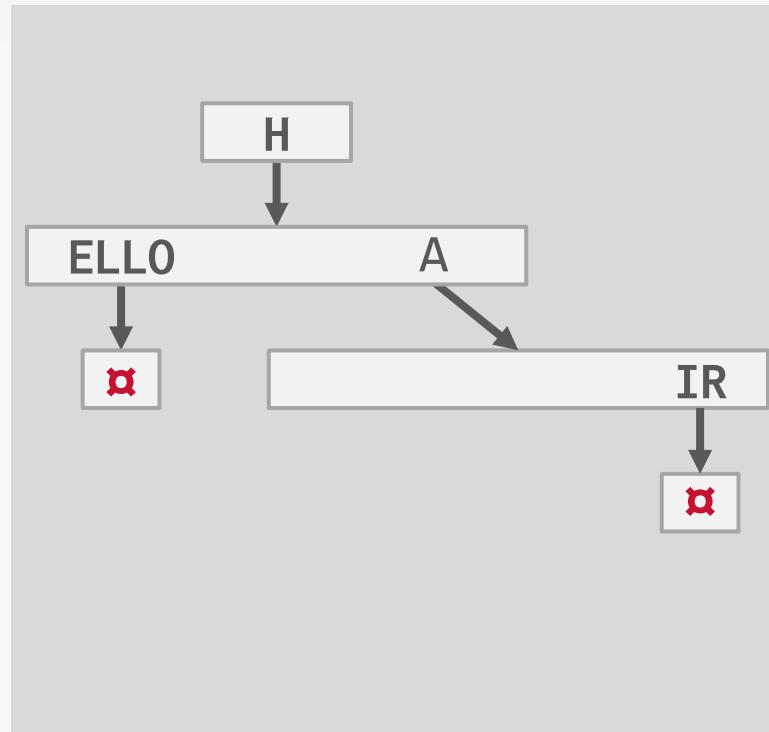
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT
Delete HAVE



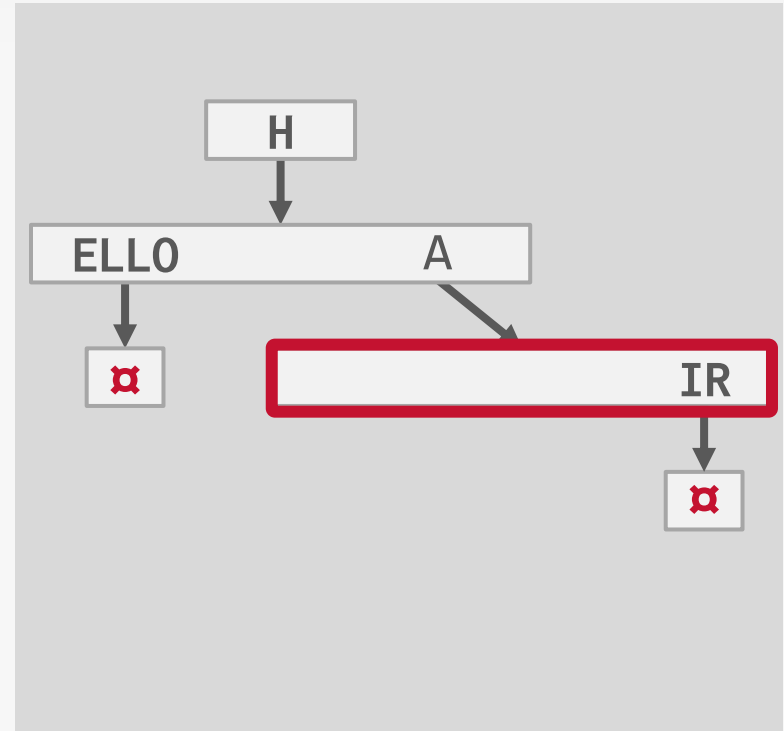
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT
Delete HAVE



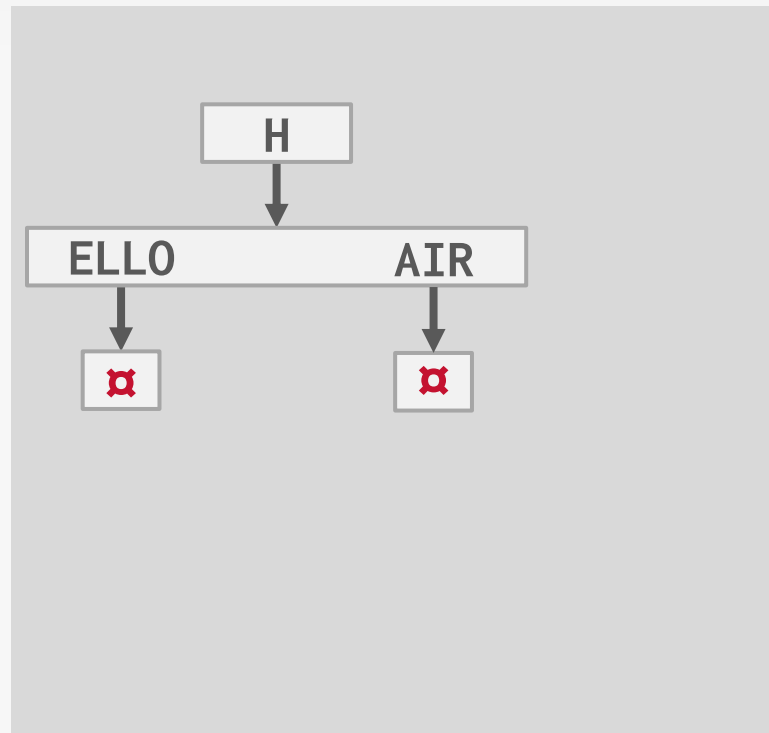
RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT
Delete HAVE



RADIX TREE: MODIFICATIONS

Insert HAIR
Delete HAT
Delete HAVE



TRIE VARIANTS

Judy Arrays (HP)

→ 256-way radix tree. First known radix tree that supports adaptive node representation.

ART Index (HyPer)

→ 256-way radix tree that also adapts node types based on its population.

Masstree (Silo)

→ B+Tree that uses tries inside each node for its key/value mapping.

SuRF (CMU)

→ Compressed static trie that supports point and range queries.

OBSERVATION

The indexes that we've discussed are useful for "point" and "range" queries:

- Find all customers in the 15217 zipcode.
- Find all orders between June 2024 and September 2024.

They are **not** good at keyword searches:

- Example: Find all Wikipedia articles that contain the word "Pavlo"

OBSERVATION

The indexes that we've discussed are useful for "point" and "range" queries:

- Find all customers in the 15217 zipcode.
- Find all orders between June 2024 and September 2024.

They are **not** good at keyword searches:

- Example: Find all Wikipedia articles that contain the word "Pavlo"

revisions(id, content, ...)

id	content
11	Wu-Tang Clan is an American hip hop musical collective formed in Staten Island, New York City, in 1992...
22	Carnegie Mellon University (CMU) is a private research university in Pittsburgh, Pennsylvania. The institution was established in 1900 by Andrew Carnegie...
33	In computing, a database is an organized collection of data or a type of data store based on the use of a database management system (DBMS), the software...
44	Andrew Pavlo, best known as Andy Pavlo, is an associate professor of Computer Science at Carnegie Mellon University. He conducts research on database...

```
CREATE INDEX idx_rev_cntnt
ON revisions (content);
```

```
SELECT pageID FROM revisions
WHERE content LIKE '%Pavlo%';
```

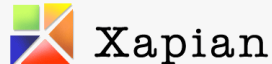
INVERTED INDEX

An **inverted index** stores a mapping of terms to records that contain those terms in the target attribute.

→ Sometimes called a *full-text search index*.

→ Originally called a **concordance** (1200s).

Many major DBMSs support these natively. But there are also specialized DBMSs and libraries.

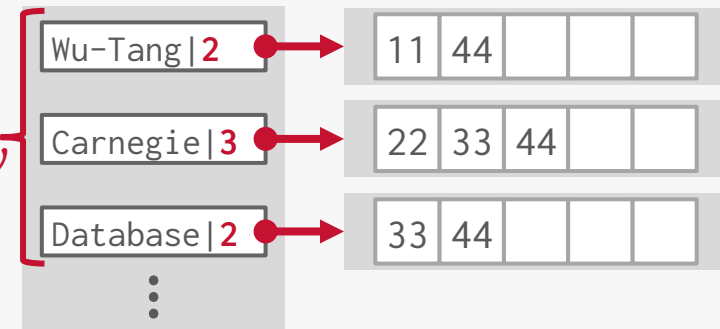


revisions(id, content, ...)

id	content
11	Wu-Tang Clan is an American hip hop musical collective formed in Staten Island, New York City, in 1992...
22	Carnegie Mellon University (CMU) is a private research university in Pittsburgh, Pennsylvania. The institution was established in 1900 by Andrew Carnegie...
33	In computing, a database is an organized collection of data or a type of data store based on the use of a database management system (DBMS), the software...
44	Andrew Pavlo, best known as Andy Pavlo, is an associate professor of Computer Science at Carnegie Mellon University. He conducts research on database...

Dictionary

Posting Lists



INVERTED INDEX: LUCENE

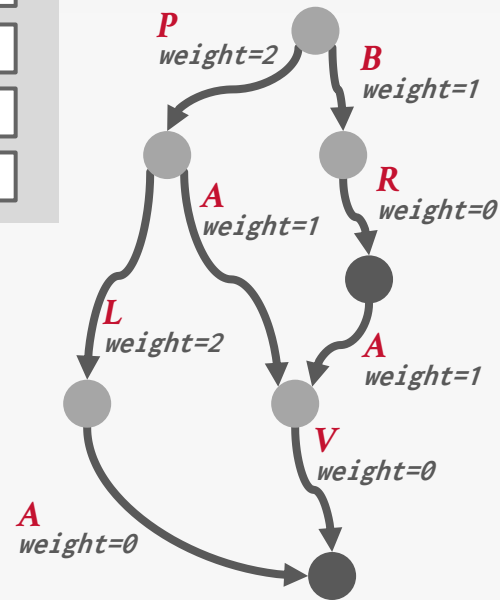
Uses a **Finite State Transducer** for determining offset of terms in dictionary.

Incrementally create dictionary segments and then merge them in the background.

- Uses **compression methods** we previously discussed (e.g., delta, bit packing).
- Also supports precomputed aggregations for terms and occurrences.

Dictionary

1	BR
2	BRAV
3	PAV
4	PLA



INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

Incrementally create dictionary segments and then merge them in the background.

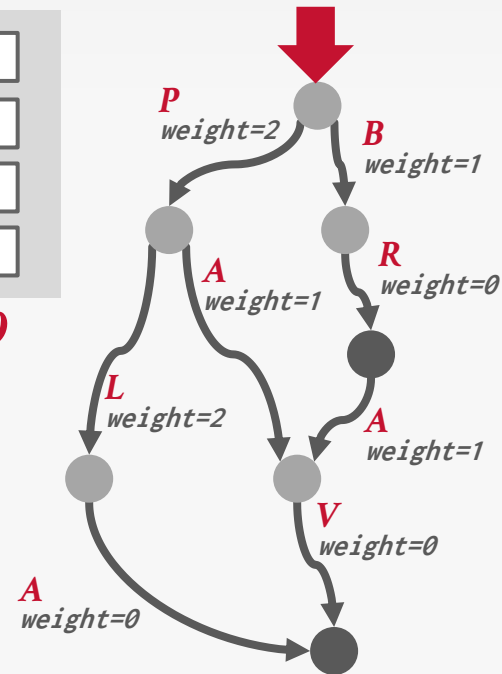
- Uses **compression methods** we previously discussed (e.g., delta, bit packing).
- Also supports precomputed aggregations for terms and occurrences.

Dictionary

1	BR
2	BRAV
3	PAV
4	PLA

Offset= 0

Find PAV



INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

Incrementally create dictionary segments and then merge them in the background.

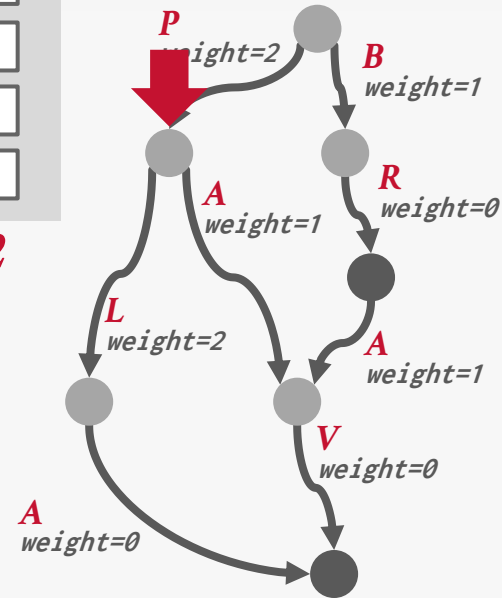
- Uses **compression methods** we previously discussed (e.g., delta, bit packing).
- Also supports precomputed aggregations for terms and occurrences.

Dictionary

1	BR
2	BRAV
3	PAV
4	PLA

Offset= 2

Find PAV



INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

Incrementally create dictionary segments and then merge them in the background.

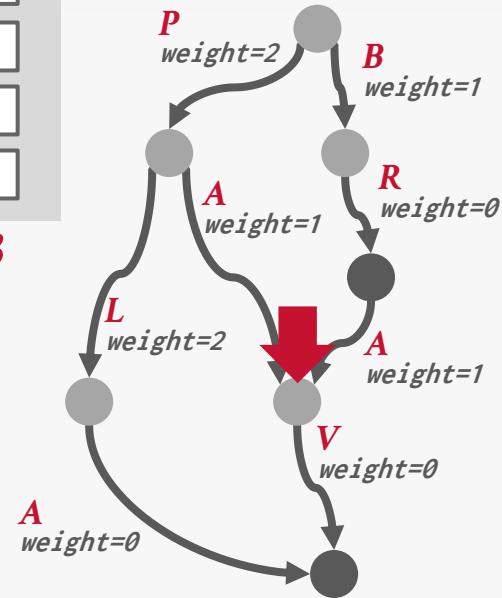
- Uses **compression methods** we previously discussed (e.g., delta, bit packing).
- Also supports precomputed aggregations for terms and occurrences.

Dictionary

1	BR
2	BRAV
3	PAV
4	PLA

Offset= 3

Find PAV

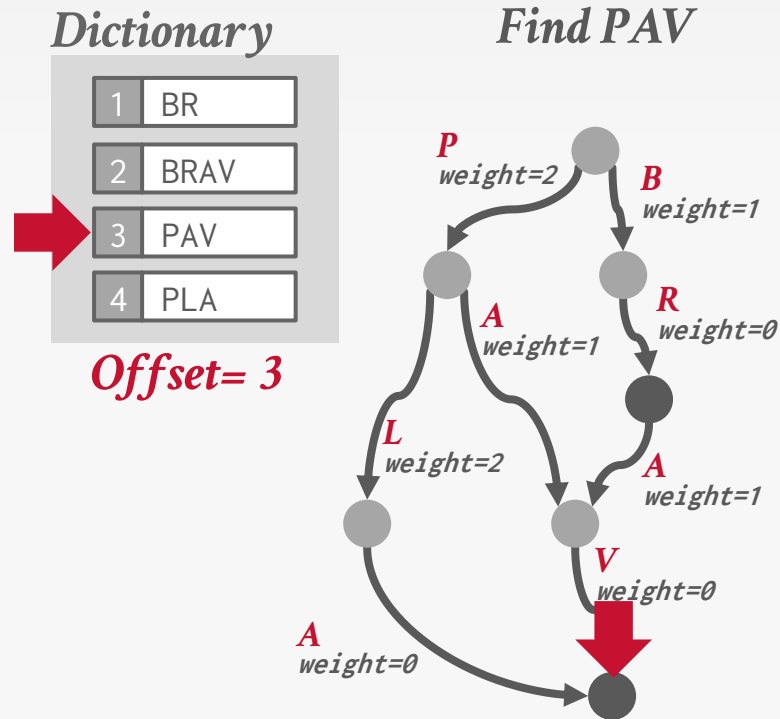


INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

Incrementally create dictionary segments and then merge them in the background.

- Uses **compression methods** we previously discussed (e.g., delta, bit packing).
- Also supports precomputed aggregations for terms and occurrences.



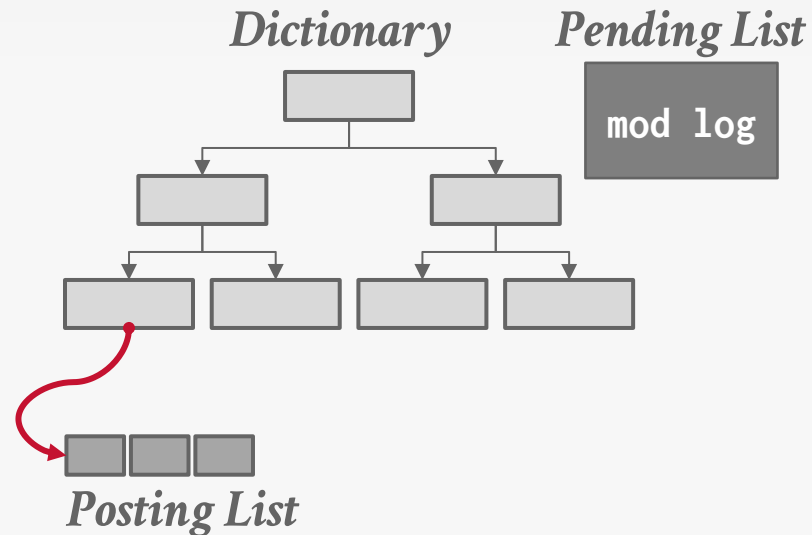
INVERTED INDEX: POSTGRESQL

PostgreSQL's **Generalized Inverted Index** (GIN) uses a B+Tree for the term dictionary that map to a posting list data structure.

Posting list contents varies depending on number of records per term:

- **Few**: Sorted list of record ids.
- **Many**: Another B+Tree of record ids.

Uses a separate "pending list" log to avoid incremental updates.



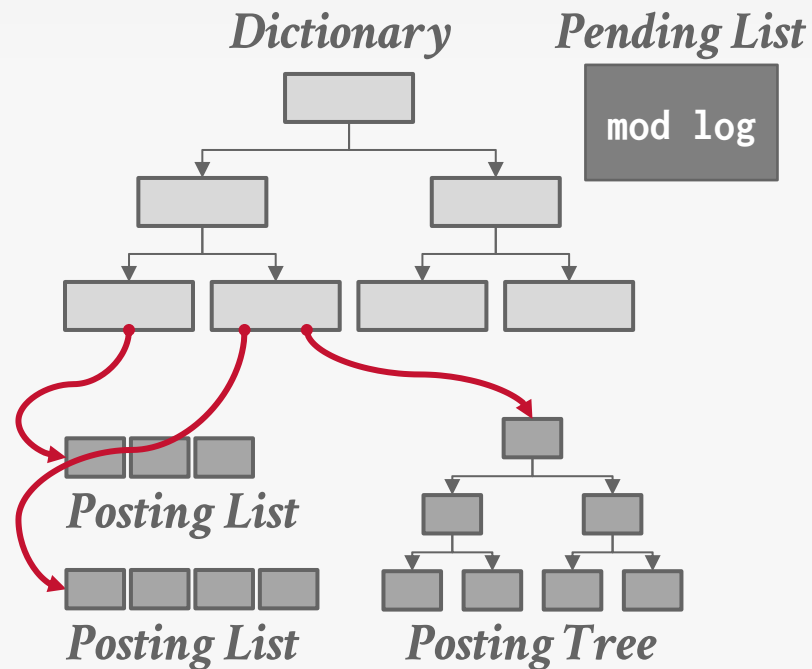
INVERTED INDEX: POSTGRESQL

PostgreSQL's **Generalized Inverted Index** (GIN) uses a B+Tree for the term dictionary that map to a posting list data structure.

Posting list contents varies depending on number of records per term:

- **Few**: Sorted list of record ids.
- **Many**: Another B+Tree of record ids.

Uses a separate "pending list" log to avoid incremental updates.



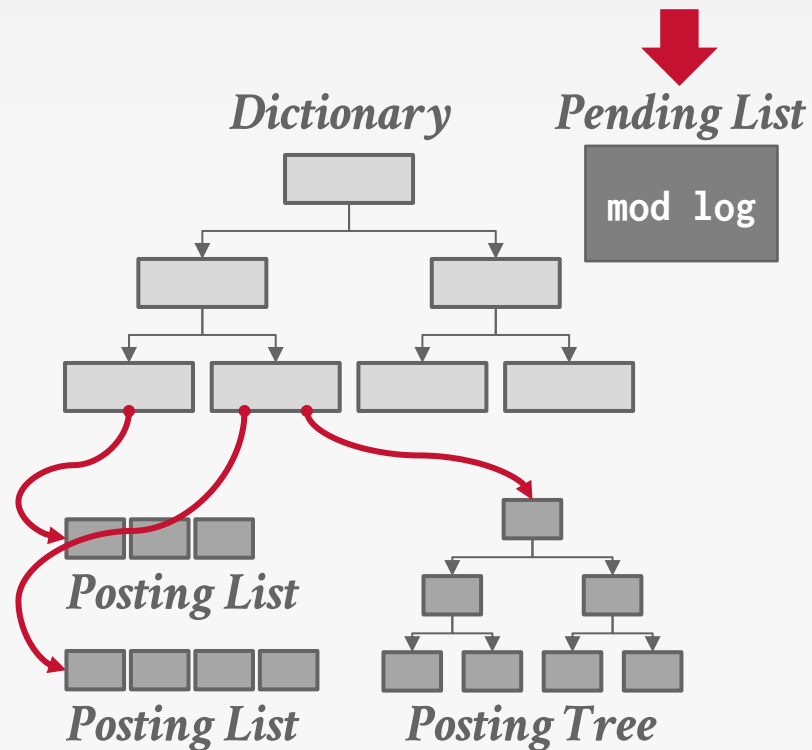
INVERTED INDEX: POSTGRESQL

PostgreSQL's **Generalized Inverted Index** (GIN) uses a B+Tree for the term dictionary that map to a posting list data structure.

Posting list contents varies depending on number of records per term:

- **Few**: Sorted list of record ids.
- **Many**: Another B+Tree of record ids.

Uses a separate "pending list" log to avoid incremental updates.



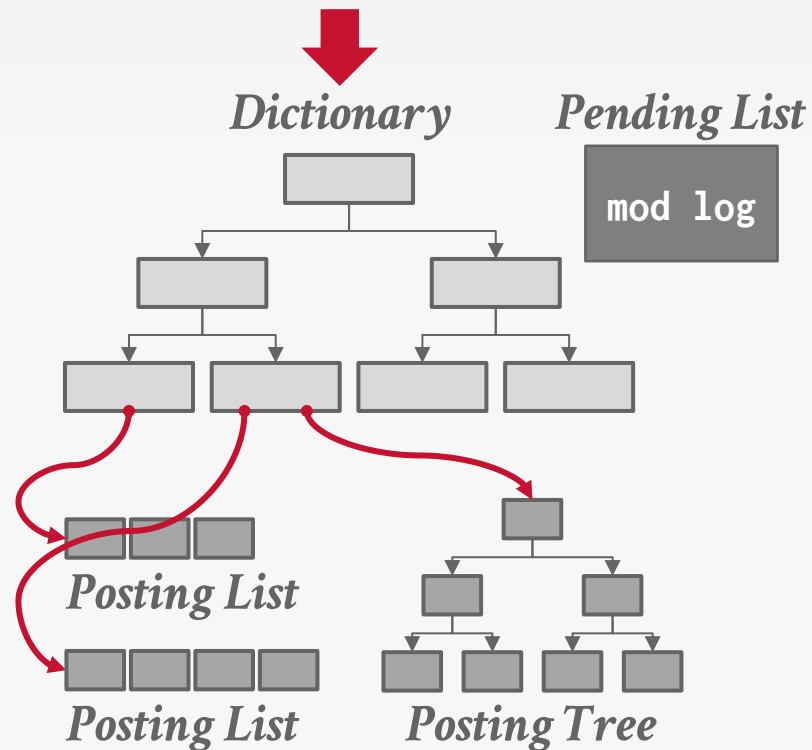
INVERTED INDEX: POSTGRESQL

PostgreSQL's **Generalized Inverted Index** (GIN) uses a B+Tree for the term dictionary that map to a posting list data structure.

Posting list contents varies depending on number of records per term:

- **Few**: Sorted list of record ids.
- **Many**: Another B+Tree of record ids.

Uses a separate "pending list" log to avoid incremental updates.



INVERTED INDEX: ENHANCEMENTS

Rankings: The DBMS can rank search results based on the frequency of terms in each record relative to other records.

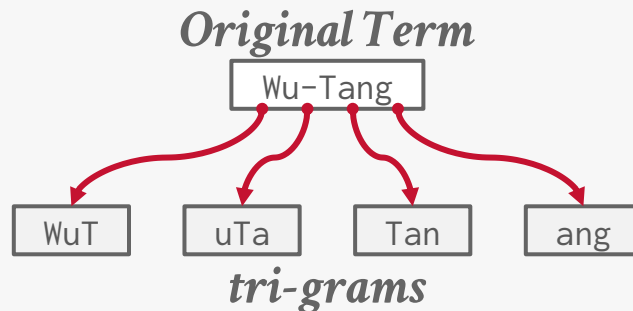
→ Examples: [TF-IDF](#), [BM25](#)

Tokenizers: Split terms into *n*-grams to support fuzzy text searches and autocomplete ("did you mean").

→ Examples: [Elastic N-gram](#), [pg_trgm](#)

```
SELECT pageID FROM revisions
WHERE content LIKE 'the';
```

```
SELECT pageID FROM revisions
WHERE content LIKE 'Wu-Tang';
```



OBSERVATION

Inverted indexes search data based on its contents with some term tweaking based on linguistic models.

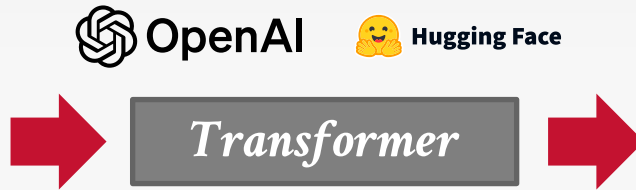
→ Example: Normalization ("Wu-Tang" matches "Wu Tang").

Instead of searching for records containing keywords (e.g., "Wu-Tang"), an application may want search for records that are semantically similar to topics (e.g., "hip-hop groups with songs about slinging").

SEMANTIC SIMILARITY SEARCH

Album(id, name, year, lyrics)

id	name	year	lyrics
Id1	Enter the Wu-Tang	1993	<text>
Id2	Run the Jewels 2	2015	<text>
Id3	Liquid Swords	1995	<text>
Id4	We Got It from Here	2016	<text>



Embeddings

Id1 → [0.32, 0.78, 0.30, ...]

Id2 → [0.99, 0.19, 0.81, ...]

Id3 → [0.01, 0.18, 0.85, ...]

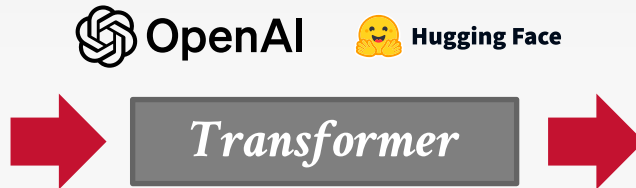
Id4 → [0.19, 0.82, 0.24, ...]

⋮

SEMANTIC SIMILARITY SEARCH

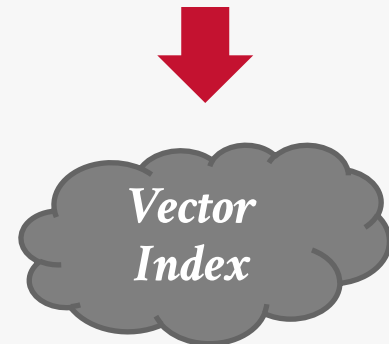
Album(id, name, year, lyrics)

id	name	year	lyrics
Id1	Enter the Wu-Tang	1993	<text>
Id2	Run the Jewels 2	2015	<text>
Id3	Liquid Swords	1995	<text>
Id4	We Got It from Here	2016	<text>



Embeddings

```
Id1 → [0.32, 0.78, 0.30, ...]
Id2 → [0.99, 0.19, 0.81, ...]
Id3 → [0.01, 0.18, 0.85, ...]
Id4 → [0.19, 0.82, 0.24, ...]
⋮
```



SEMANTIC SIMILARITY SEARCH

Album(id, name, year, lyrics)

id	name	year	lyrics
Id1	Enter the Wu-Tang	1993	<text>
Id2	Run the Jewels 2	2015	<text>
Id3	Liquid Swords	1995	<text>
Id4	We Got It from Here	2016	<text>



OpenAI



Hugging Face

Transformer

Embeddings

Id1 → [0.32, 0.78, 0.30, ...]

Id2 → [0.99, 0.19, 0.81, ...]

Id3 → [0.01, 0.18, 0.85, ...]

Id4 → [0.19, 0.82, 0.24, ...]

⋮

Query

Find albums with lyrics about
running from the police

*Vector
Index*

SEMANTIC SIMILARITY SEARCH

Album(id, name, year, lyrics)

id	name	year	lyrics
Id1	Enter the Wu-Tang	1993	<text>
Id2	Run the Jewels 2	2015	<text>
Id3	Liquid Swords	1995	<text>
Id4	We Got It from Here	2016	<text>

 OpenAI  Hugging Face

Transformer

Embeddings

Id1 → [0.32, 0.78, 0.30, ...]
Id2 → [0.99, 0.19, 0.81, ...]
Id3 → [0.01, 0.18, 0.85, ...]
Id4 → [0.19, 0.82, 0.24, ...]
⋮

Query

Find albums with lyrics about
running from the police

[0.02, 0.10, 0.24, ...]

Ranked List of Ids

*Vector
Index*

SEMANTIC SIMILARITY SEARCH

Album(id, name, year, lyrics)

id	name	year	lyrics
Id1	<u>Enter the Wu-Tang</u>	1993	<text>
Id2	<u>Run the Jewels 2</u>	2015	<text>
Id3	<u>Liquid Swords</u>	1995	<text>
Id4	<u>We Got It from Here</u>	2016	<text>



OpenAI



Hugging Face

Transformer

Embeddings

Id1 → [0.32, 0.78, 0.30, ...]

Id2 → [0.99, 0.19, 0.81, ...]

Id3 → [0.01, 0.18, 0.85, ...]

Id4 → [0.19, 0.82, 0.24, ...]

⋮

*Vector
Index*

SEMANTIC SIMILARITY SEARCH

Album(id, name, year, lyrics)

id	name	year	lyrics
Id1	Enter the Wu-Tang	1993	<text>
Id2	Run the Jewels 2	2015	<text>
Id3	Liquid Swords	1995	<text>
Id4	We Got It from Here	2016	<text>



OpenAI



Hugging Face

Transformer

Embeddings

Id1 → [0.32, 0.78, 0.30, ...]

Id2 → [0.99, 0.19, 0.81, ...]

Id3 → [0.01, 0.18, 0.85, ...]

Id4 → [0.19, 0.82, 0.24, ...]

⋮

Query

Find albums with lyrics about
running from the police
and released after **2010**

[0.02, 0.10, 0.24, ...]

year > 2010

*Vector
Index*

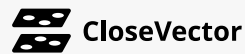
VECTOR INDEXES

Specialized data structures to perform nearest-neighbor searches on embeddings (i.e., one-dimensional arrays of floating-point numbers).

- May also need to filter data before / after vector searches.
- Result correctness depends on whether the result "feels right".

Two approaches:

- Inverted Indexes
- Graphs



VECTOR INDEXES



→ Graphs



VECTOR SEARCH: INVERTED INDEXES

Partition vectors into smaller groups using a clustering algorithm and then build an inverted index that maps cluster centroids to records.

- Preprocess / quantize vectors to reduce dimensionality.
- Some implementations support incremental updates.
- Examples: [IVFFlat](#), [pgVector](#)

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.

- Also check nearby groups to improve accuracy.

VECTOR SEARCH: INVERTED INDEXES

Compute k-means clustering on embeddings to find centroids.

Assign each embedding to the cluster with its closest centroid.

Embeddings



Id1 → [0.32, 0.78, 0.30, ...]

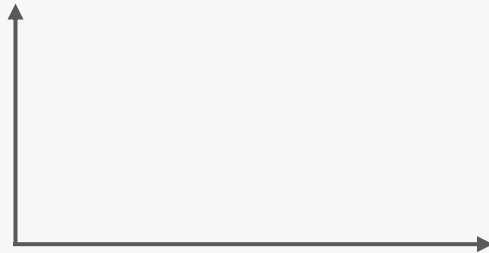
Id2 → [0.99, 0.19, 0.81, ...]

Id3 → [0.01, 0.18, 0.85, ...]

Id4 → [0.19, 0.82, 0.24, ...]

⋮

Clusters



VECTOR SEARCH: INVERTED INDEXES

Compute k-means clustering on embeddings to find centroids.

Assign each embedding to the cluster with its closest centroid.

Embeddings

Id1 → [0.32, 0.78, 0.30, ...]

Id2 → [0.99, 0.19, 0.81, ...]

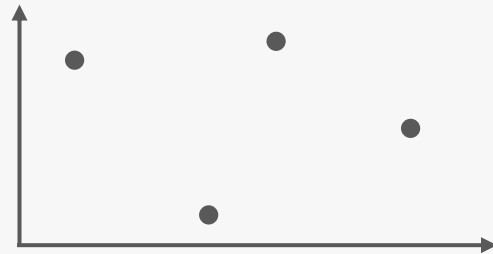
Id3 → [0.01, 0.18, 0.85, ...]

Id4 → [0.19, 0.82, 0.24, ...]

⋮



Clusters



VECTOR SEARCH: INVERTED INDEXES

Compute k-means clustering on embeddings to find centroids.

Assign each embedding to the cluster with its closest centroid.

Embeddings

Id1 → [0.32, 0.78, 0.30, ...]

Id2 → [0.99, 0.19, 0.81, ...]

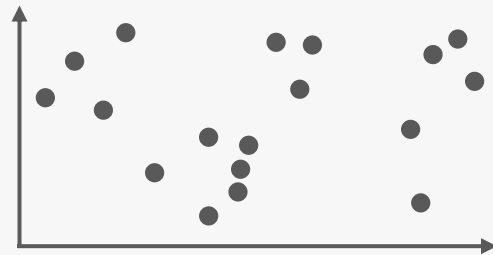
Id3 → [0.01, 0.18, 0.85, ...]

Id4 → [0.19, 0.82, 0.24, ...]

⋮



Clusters



VECTOR SEARCH: INVERTED INDEXES

Compute k-means clustering on embeddings to find centroids.

Assign each embedding to the cluster with its closest centroid.

Embeddings

Id1 → [0.32, 0.78, 0.30, ...]

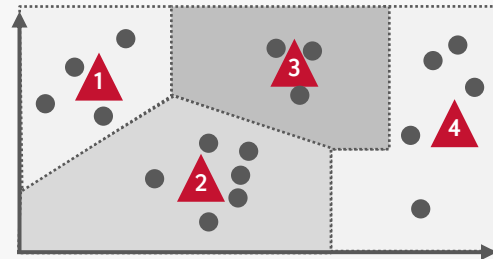
Id2 → [0.99, 0.19, 0.81, ...]

Id3 → [0.01, 0.18, 0.85, ...]

Id4 → [0.19, 0.82, 0.24, ...]

⋮

Clusters

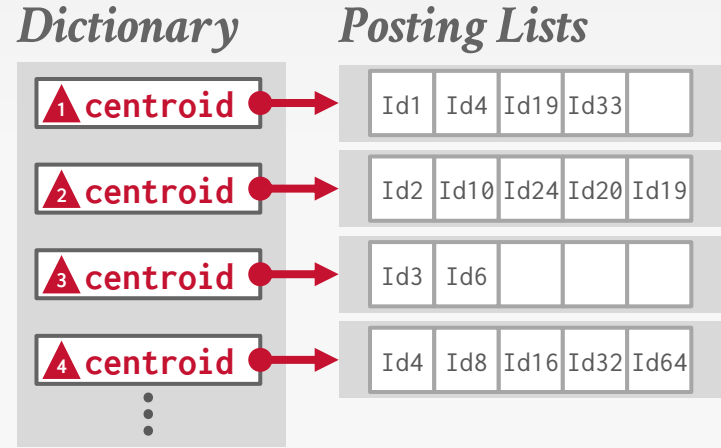


VECTOR SEARCH: INVERTED INDEXES

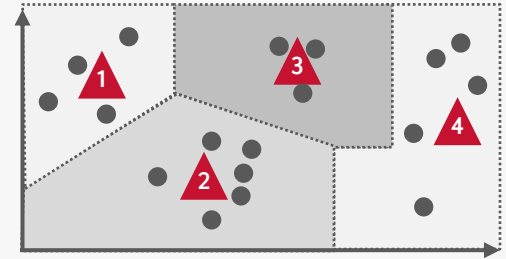
Compute k-means clustering on embeddings to find centroids.

Assign each embedding to the cluster with its closest centroid.

Build an inverted index mapping centroids to a posting list of the records in each cluster.



Clusters



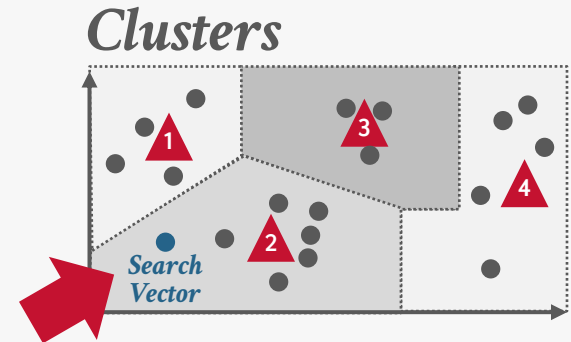
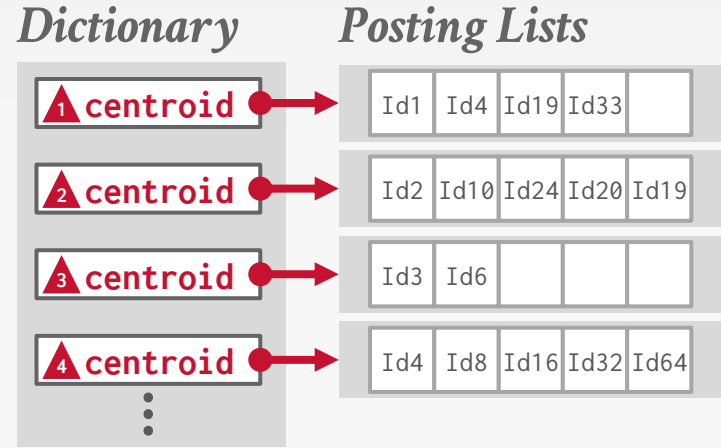
VECTOR SEARCH: INVERTED INDEXES

Compute k-means clustering on embeddings to find centroids.

Assign each embedding to the cluster with its closest centroid.

Build an inverted index mapping centroids to a posting list of the records in each cluster.

To find matches, compute centroid of closest cluster on search embedding and then evaluate posting list



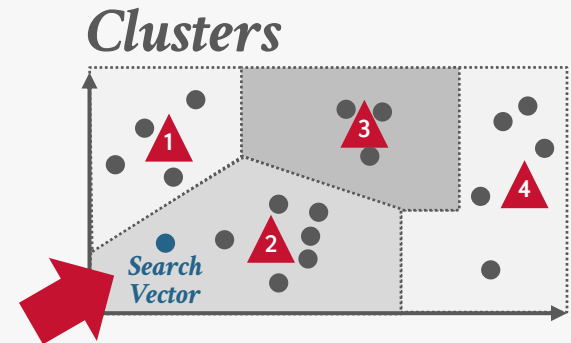
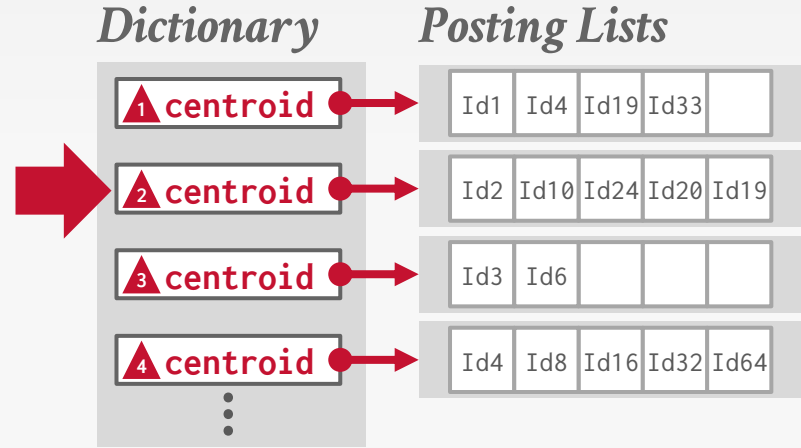
VECTOR SEARCH: INVERTED INDEXES

Compute k-means clustering on embeddings to find centroids.

Assign each embedding to the cluster with its closest centroid.

Build an inverted index mapping centroids to a posting list of the records in each cluster.

To find matches, compute centroid of closest cluster on search embedding and then evaluate posting list



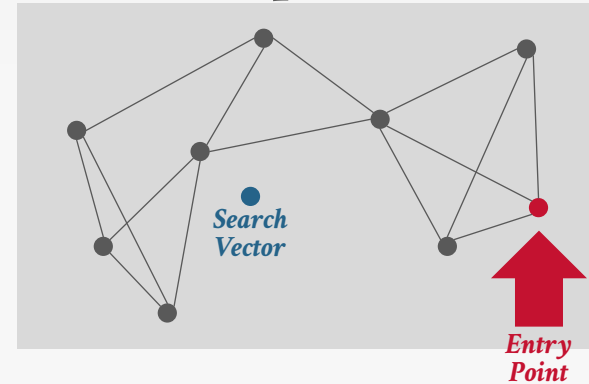
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



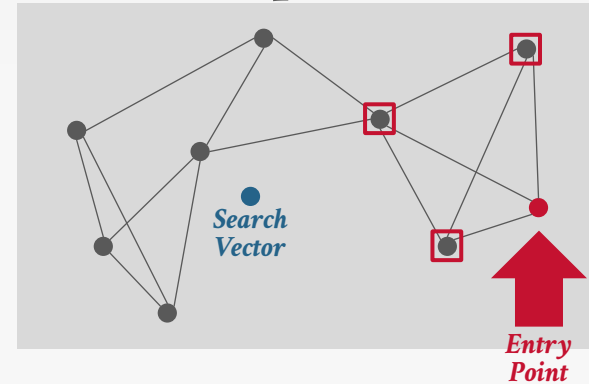
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



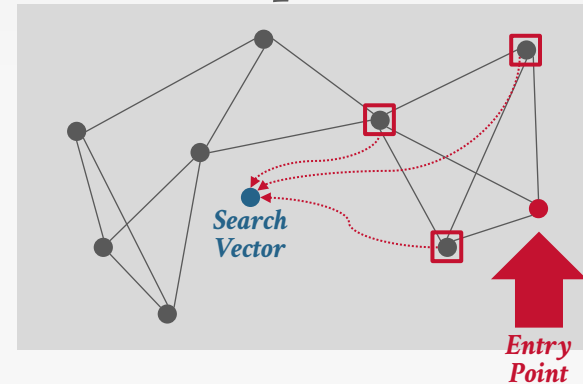
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



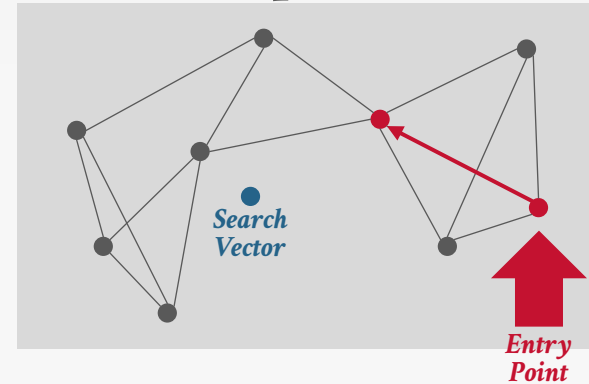
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



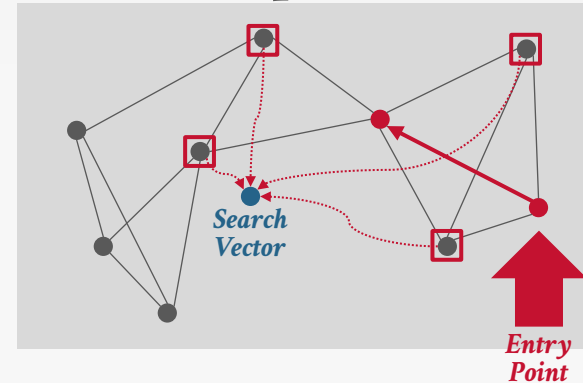
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



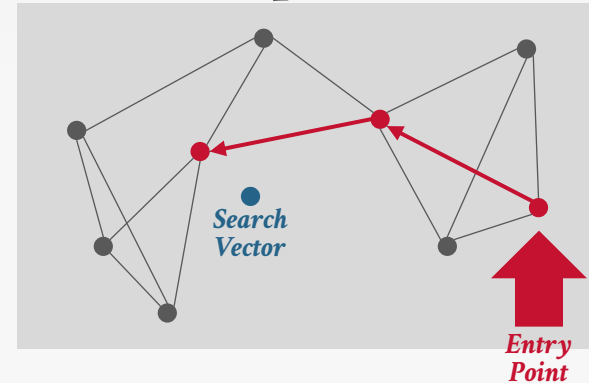
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



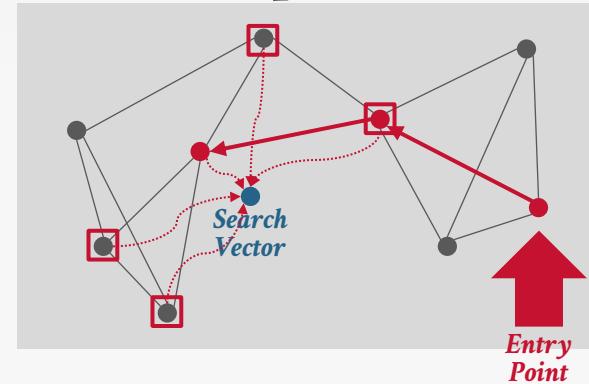
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



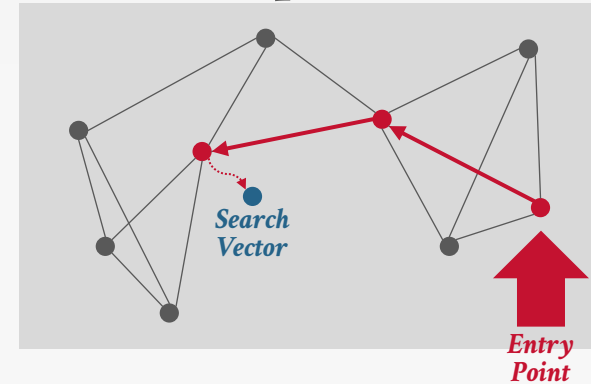
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Vector Graph



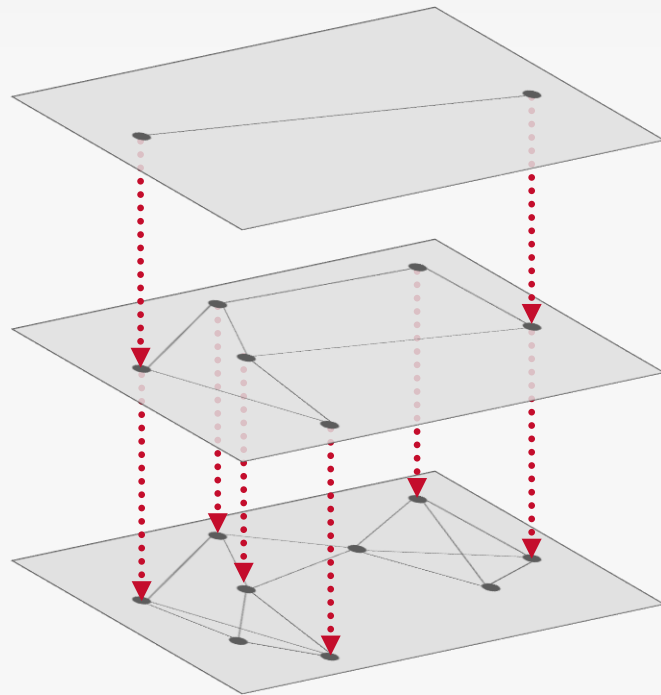
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Hierarchical Graph

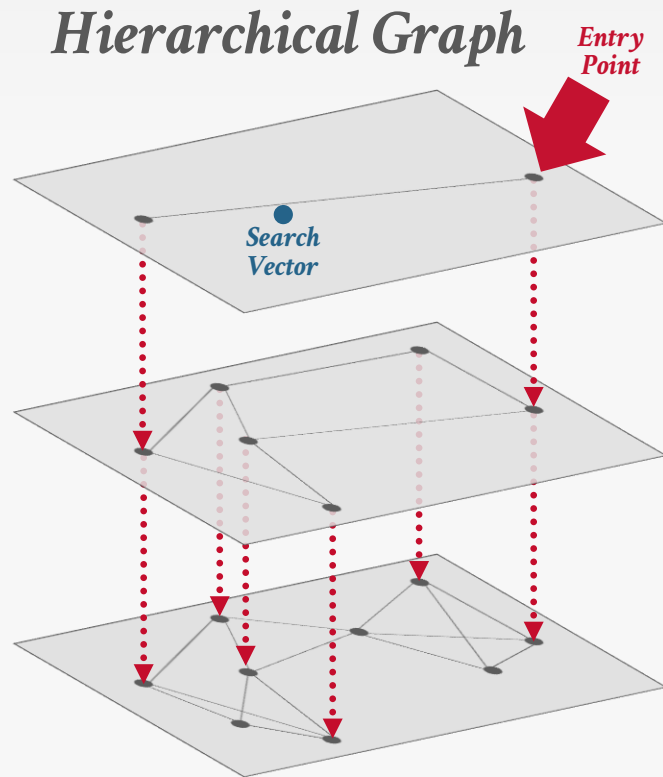


VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

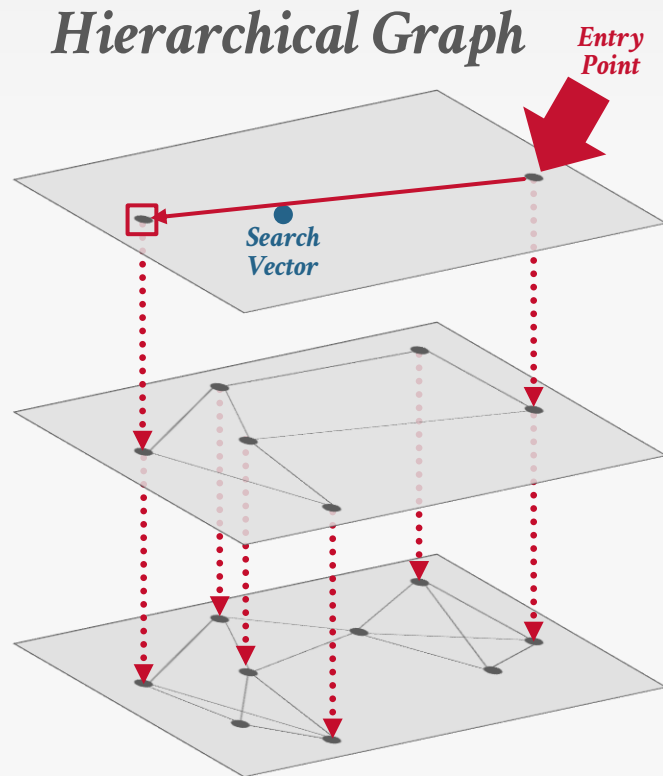


VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.



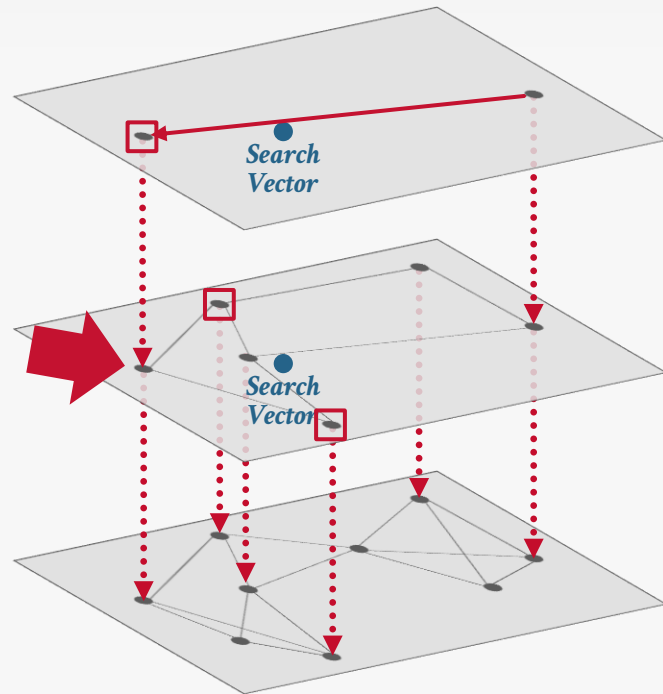
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Hierarchical Graph



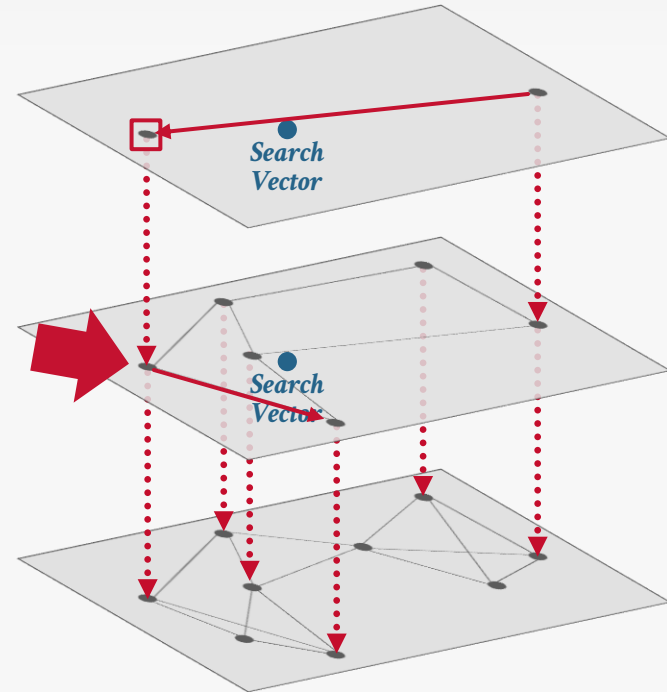
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Hierarchical Graph



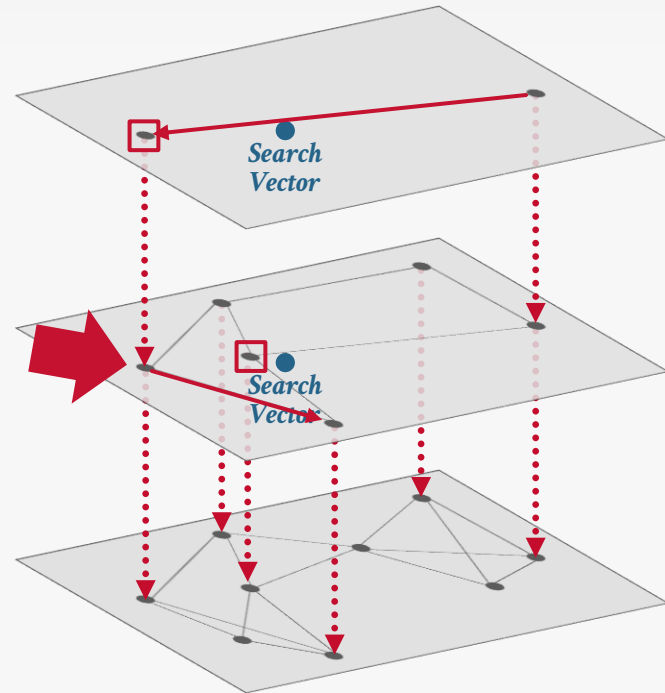
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Hierarchical Graph



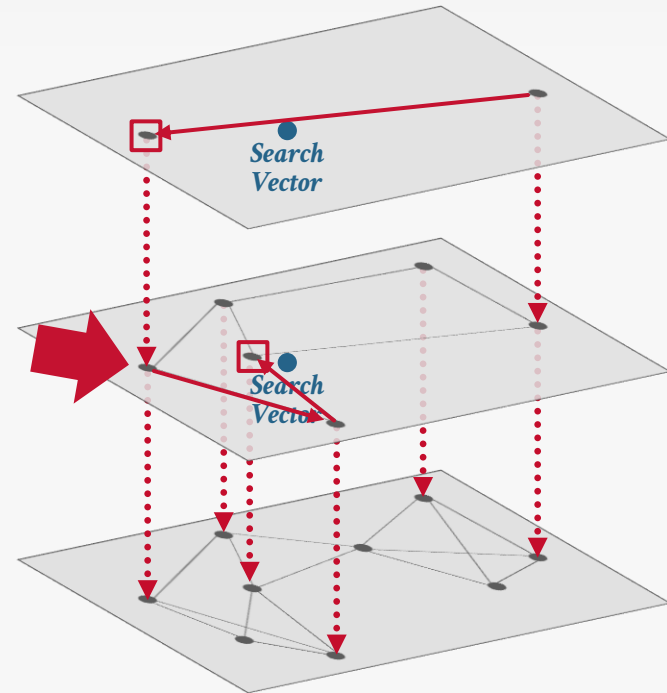
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Hierarchical Graph



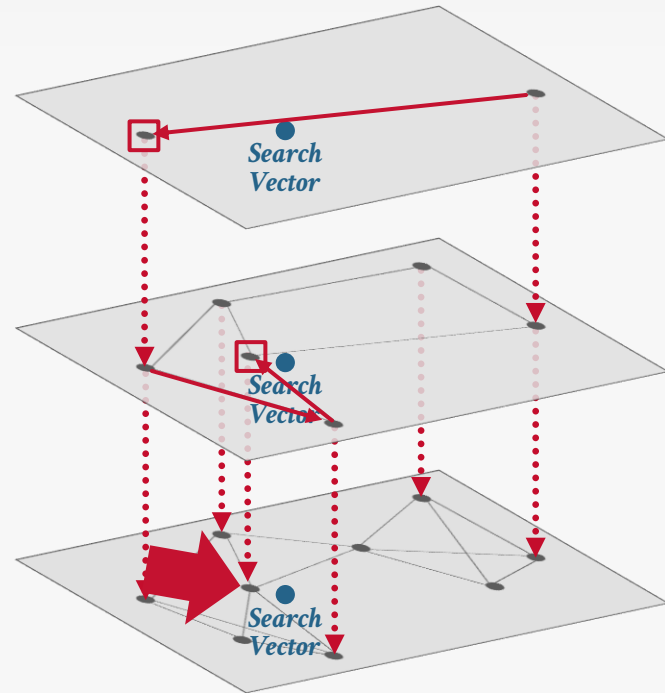
VECTOR SEARCH: GRAPH

Build a graph where each node represents a vector and it has edges to (at most) its n nearest neighbors.

- Can use multiple levels of graphs ([HNSW](#))
- Examples: [Faiss](#), [hnswlib](#), [DiskANN](#)

To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Hierarchical Graph



PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      WHERE c = 'WuTang';
```

PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      WHERE c = 'WuTang';
```

```
SELECT b FROM foo
      WHERE a = 123
      AND c = 'WuTang';
```

PARTIAL INDEXES

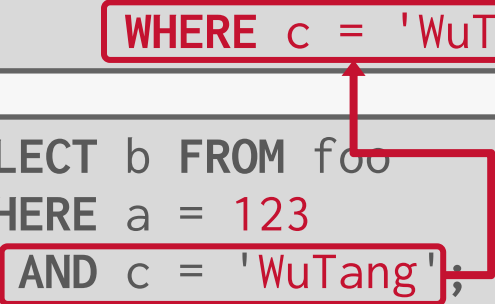
Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      WHERE c = 'WuTang';
```

```
SELECT b FROM foo
      WHERE a = 123
      AND c = 'WuTang';
```




PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.


One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      WHERE c = 'WuTang';
```



```
SELECT b FROM foo
      WHERE a = 123
```



INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      INCLUDE (c);
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      INCLUDE (c);
```

```
SELECT b FROM foo
WHERE a = 123
      AND c = 'WuTang';
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo
      ON foo (a, b)
      INCLUDE (c);
```

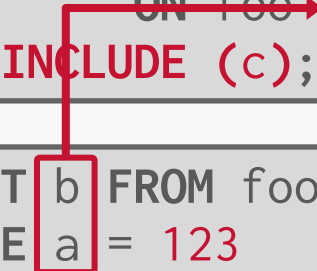
```
SELECT b FROM foo
WHERE a = 123
      AND c = 'WuTang';
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
INCLUDE (c);
```



```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```

INDEX INCLUDE COLUMNS

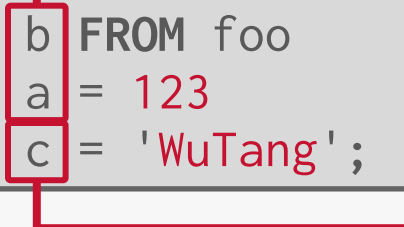
Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
INCLUDE (c);
```



```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```

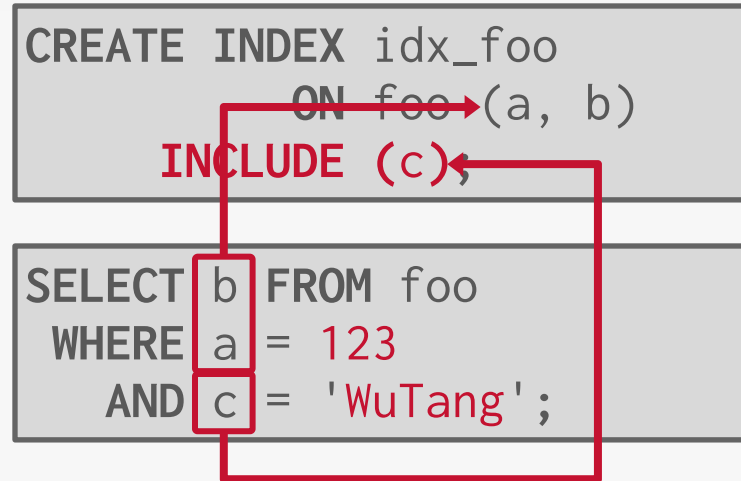


INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

If all attributes a query needs are available in an index, then the DBMS does not need to retrieve the tuple.
→ AKA covering index or index-only scans.



CONCLUSION

We will see filters again this semester.

B+Trees are still the way to go for tree indexes.

Inverted indexes are covered in [CMU 11-442](#).

We did not discuss geo-spatial tree indexes:

→ Examples: R-Tree, Quad-Tree, KD-Tree

→ This is covered in [CMU 15-826](#).

NEXT CLASS

How to make indexes thread-safe!