COMP 421: Files & Databases

Lecture 10: It's Data Structure Week!

(Or, what Ben did at his conference)

Announcements

Project 2 has been released! Get started!

Necessary material on B+Tree latching this Wendesday in class.

Reminder: project 2 will not work unless project 1 is 100%. If you still need to fix up P1, come to office hours!

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.

$\longrightarrow$ False negatives will never occur.
$\longrightarrow$ False positives can sometimes occur.
$\longrightarrow$ See Bloom Filter Calculator.

Insert(x):

$\longrightarrow$ Use *k* hash functions to set bits in the filter to 1.

Lookup(x):

$\longrightarrow$ Check whether the bits are 1 for each hash function.

Bloom Filters

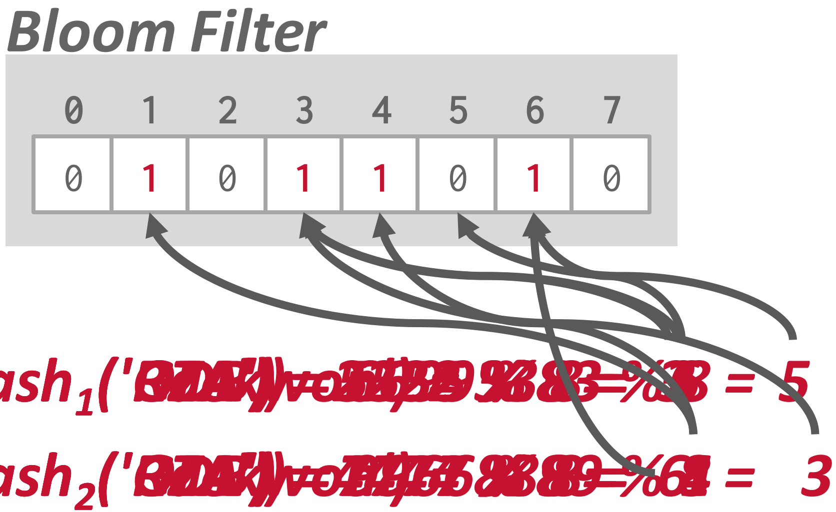
Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → ***TRUE***

Lookup 'Raekwon' → ***FALSE***

Lookup 'ODB' → ***TRUE***



Bloom Filters

False Negative Rate:

Probability that Insert(x) followed by Lookup(x) = False

After Insert(x), my bits are set forever

False Negative Rate = Zero (!)

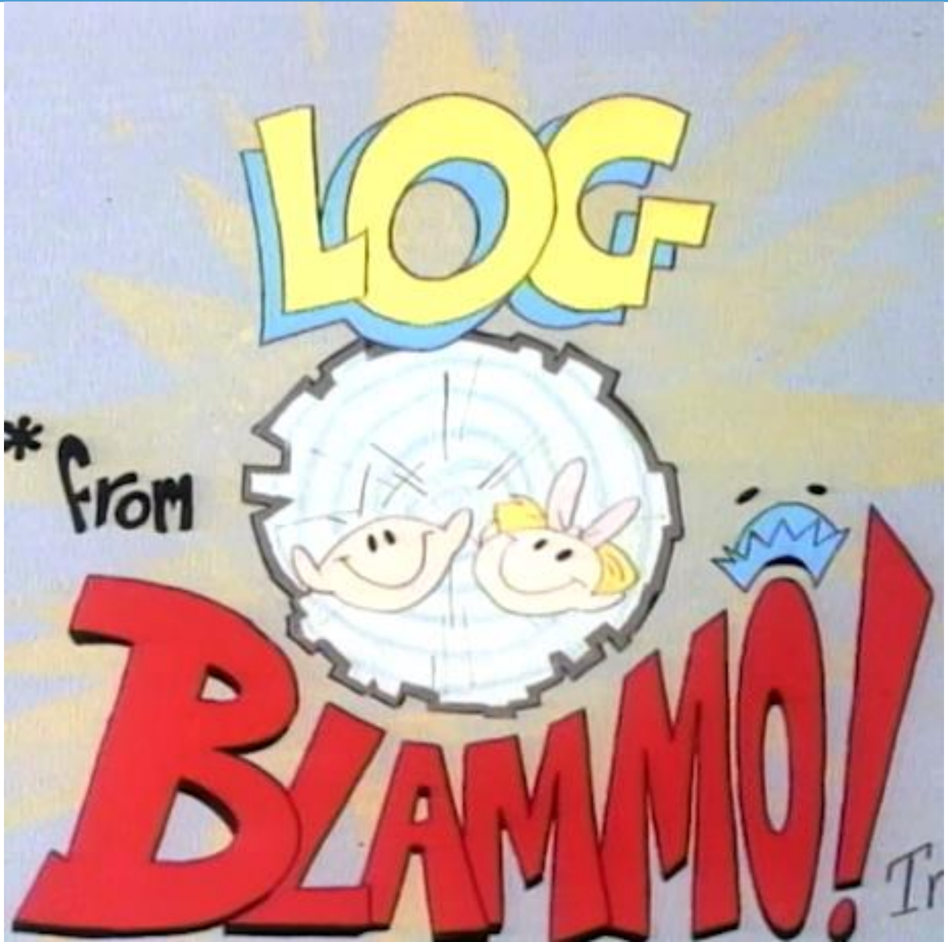
False Positive Rate:

Probability Lookup(x) = True without Insert

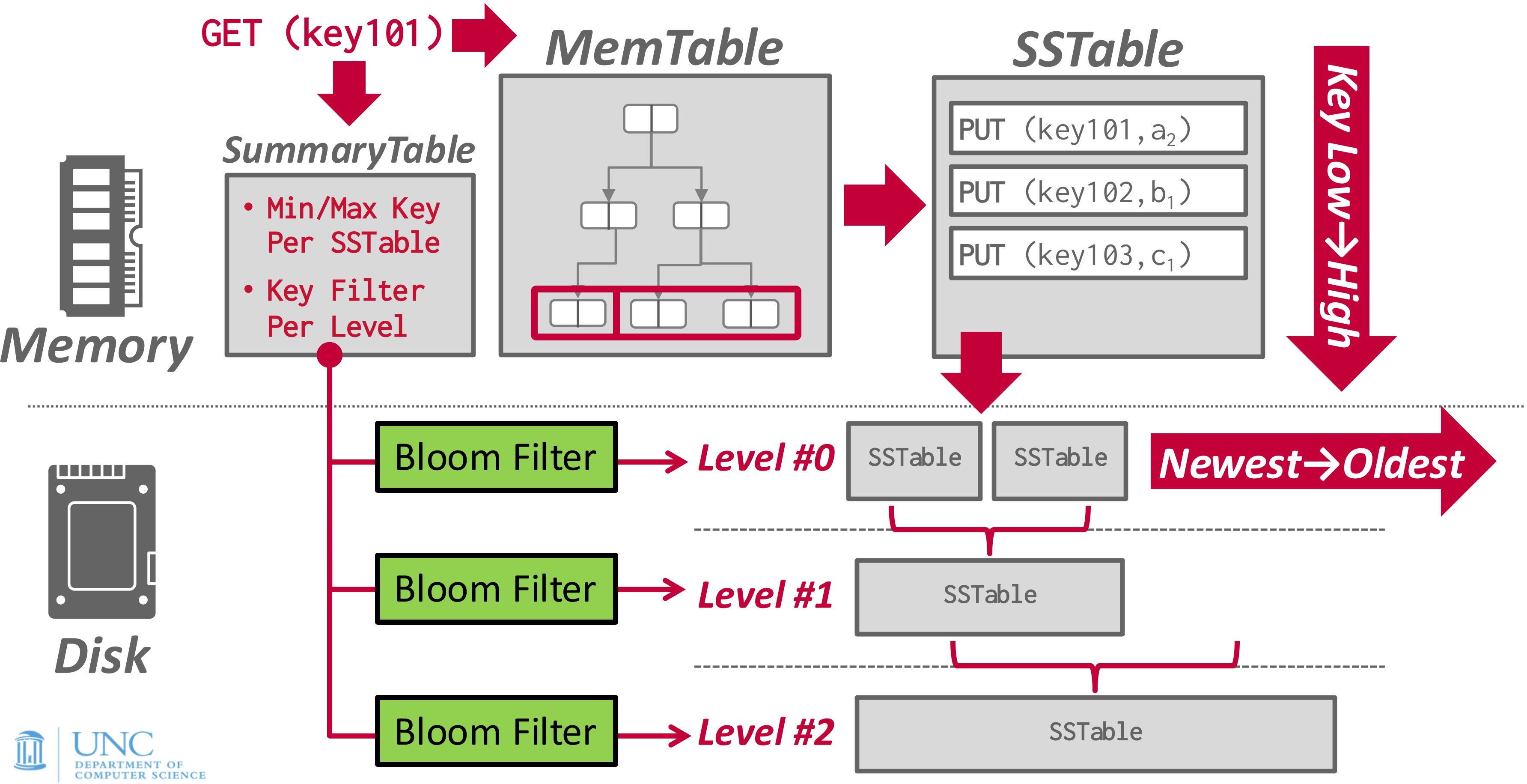
Given m bits, storing n keys, with k probes per key:

What is Prob($n + 1st$ key gives false positive)? Not 0!

It's Log!



Log-structured Storage



Bloom Filter Math

Prob(false positive) = Prob(pick k probes, all are 1)

Prob(false positive) = Prob(all k probes are 1 after n keys)

$\approx$ Prob(one probe is 1 after n keys)^k

Easier:

$$\text{Prob(probe is 0 after 1 key)} = \left(1 - \frac{1}{m}\right)^k$$

$$\text{Prob(probe is 0 after n keys)} = \left(\left(1 - \frac{1}{m}\right)^k\right)^n = \left(1 - \frac{1}{m}\right)^{kn}$$

$$= \left(\left(1 - \frac{1}{m}\right)^m\right)^{kn/m}$$

Bloom Filter Math

Prob(probe is 0 after n keys)$= \left(\left(1-\frac{1}{m}\right)^{m}\right)^{kn/m}$ (wat?)

$$\lim_{m\to\infty}\left(1-\frac{1}{m}\right)^{m} = \frac{1}{e}$$ (!)

Prob(probe is 0 after n keys) $\approx e^{-kn/m}$ for large m

Prob(Probe is 1 after n keys) $\approx \left(1-e^{-kn/m}\right)$ for large m

Prob(All k probes are 1 after n keys) $\approx \left(1-e^{-\frac{kn}{m}}\right)^{k}$ for large m

-Assumes some independence

-Not needed to get a matching result with high probability using Hoeffding Bounds

-Interested? Take my grad class!

Bloom Filters IRL

Given n, for any $0 < \epsilon < 1$, find smallest m such that:

ϵ = Prob(All k probes are 1 after n keys) $\approx \left(1 - e^{-kn/m}\right)^k$

First, fix n, m and minimize w.r.t k, $k^* = \frac{m}{n} \ln 2$

Next, for given n, using k^* probes, solve for m^*

Optimal # bits per item to achive false positive rate ϵ:

$$\frac{m^*}{n} \approx -2.08 \ln \epsilon$$

Bloom Filters IRL

Optimal # bits per item to achive false positive rate ϵ:

$$\frac{m^*}{n} = -\frac{\ln \epsilon}{(\ln 2)^2} \approx -2.08 \ln \epsilon \qquad k^* = \frac{m}{n} \ln 2 = -\frac{\ln \epsilon}{\ln 2}$$

Why do this?

Time(insert) or Time(lookup) = $O(k^*) = O\left(\log \frac{1}{\epsilon}\right)$

Say I fix false positive rate of 1%:

$\approx$ 9.6 bits / item

$\approx$ 7 probes

For any number of keys!

`sizeof(ptr)`=32 bits

`sizeof(char)`=8 bits

Other Fi

Counting Bloom Filter

→ Supports dynamically addin

→ Uses integers instead of bits

occurrences of a key in a se

Cuckoo Filter

→ Also supports dynamically a

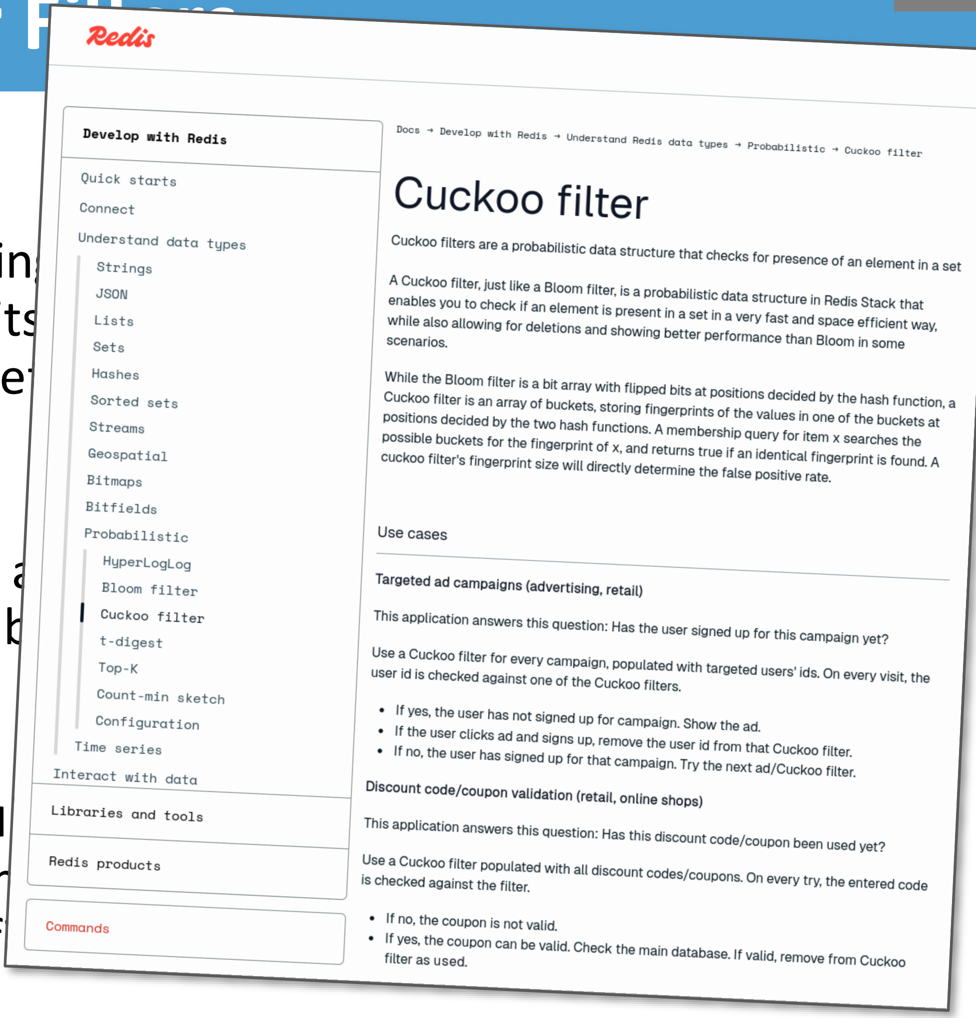
→ Uses a Cuckoo Hash Table b

of full keys.

Succinct Range Filter (Su

→ Immutable compact trie th

exact matches and range f



Redis

Develop with Redis

Quick starts
Connect
Understand data types
Strings
JSON
Lists
Sets
Hashes
Sorted sets
Streams
Geospatial
Bitmaps
Bitfields
Probabilistic
HyperLogLog
Bloom filter
Cuckoo filter
t-digest
Top-K
Count-min sketch
Configuration
Time series
Interact with data
Libraries and tools
Redis products
Commands

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.

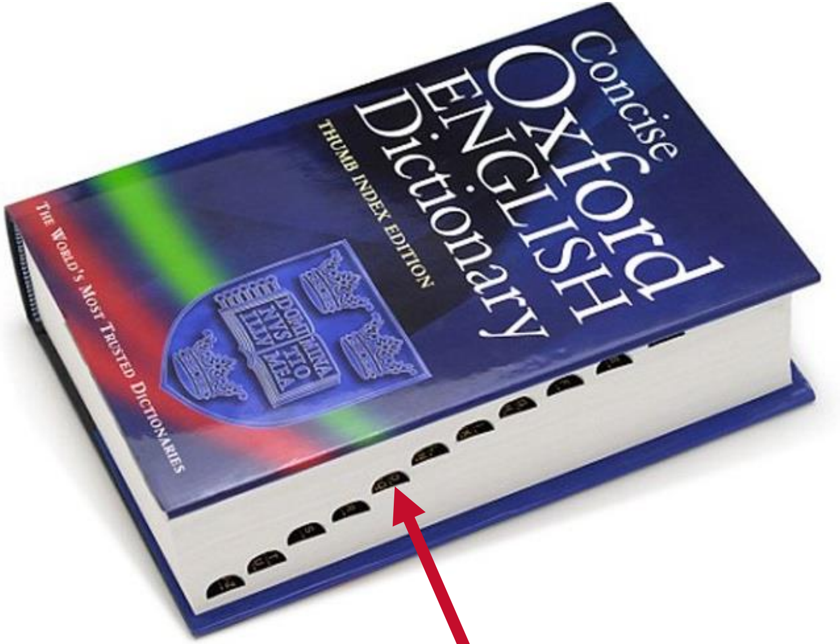
- If no, the coupon is not valid.
- If yes, the coupon can be valid. Check the main database. If valid, remove from Cuckoo filter as used.

Indexes vs. Filters

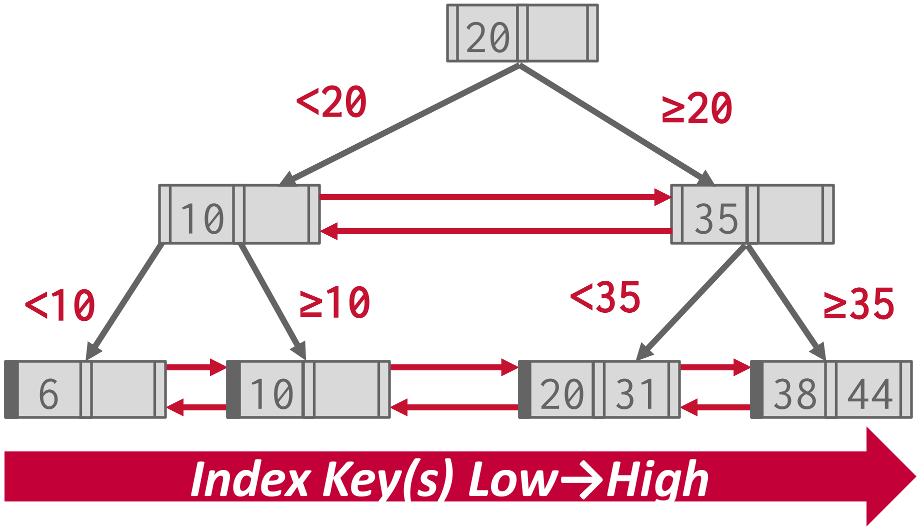
An **index** data structure of a subset of a table's attributes that are organized and/or sorted to the location of specific tuples using those attributes.
→ Example: B+Tree

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

B+Trees as Fancy Linked Lists



"M words begin here"



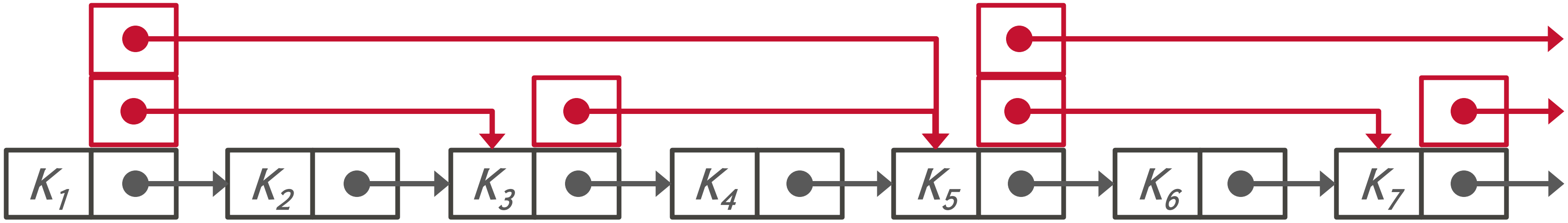
Observation

Linked lists are "simplest" index, but...

All operations have to linear search.

→ Average Cost: **O(n)**

→ More than one way to index a linked list...



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 p fraction of the keys of one below it

Maintains keys in sorted order without requiring global rebalancing.

→ **Want: 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 consulting a random number generator. Although skip lists have bad worst-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 $1\,^1/_3$ 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* 1*a*). If the list is stored in sorted order and every other node of the list also has a pointer to the node two ahead it in the list (*Figure* 1*b*), 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* 1*c*) requires that no more than $\lceil n/4 \rceil + 2$ nodes be examined. If every $(2^i)^{th}$ node has a pointer 2^i nodes ahead (*Figure* 1*d*), the number of nodes that must be examined can be reduced to $\lceil \log_2 n \rceil$ 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 $(2^i)^{th}$ node has a pointer 2^i 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 1*e*)? A node's i^{th} forward pointer, instead of pointing 2^{i-1} nodes ahead, points to the next node of level i 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 i* node has *i forward* pointers, indexed 1 through *i*. 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.

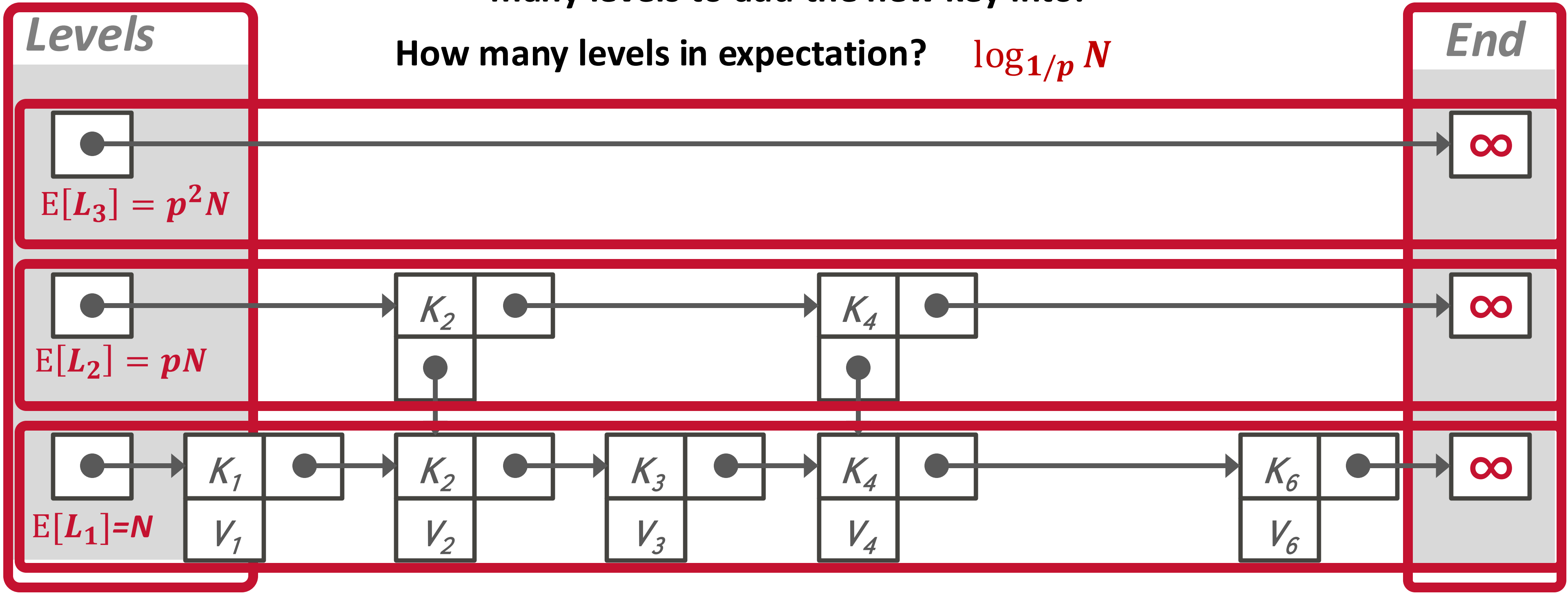
SingleStore WIREDTIGER

RocksDB Couchbase

Skip Lists Basics

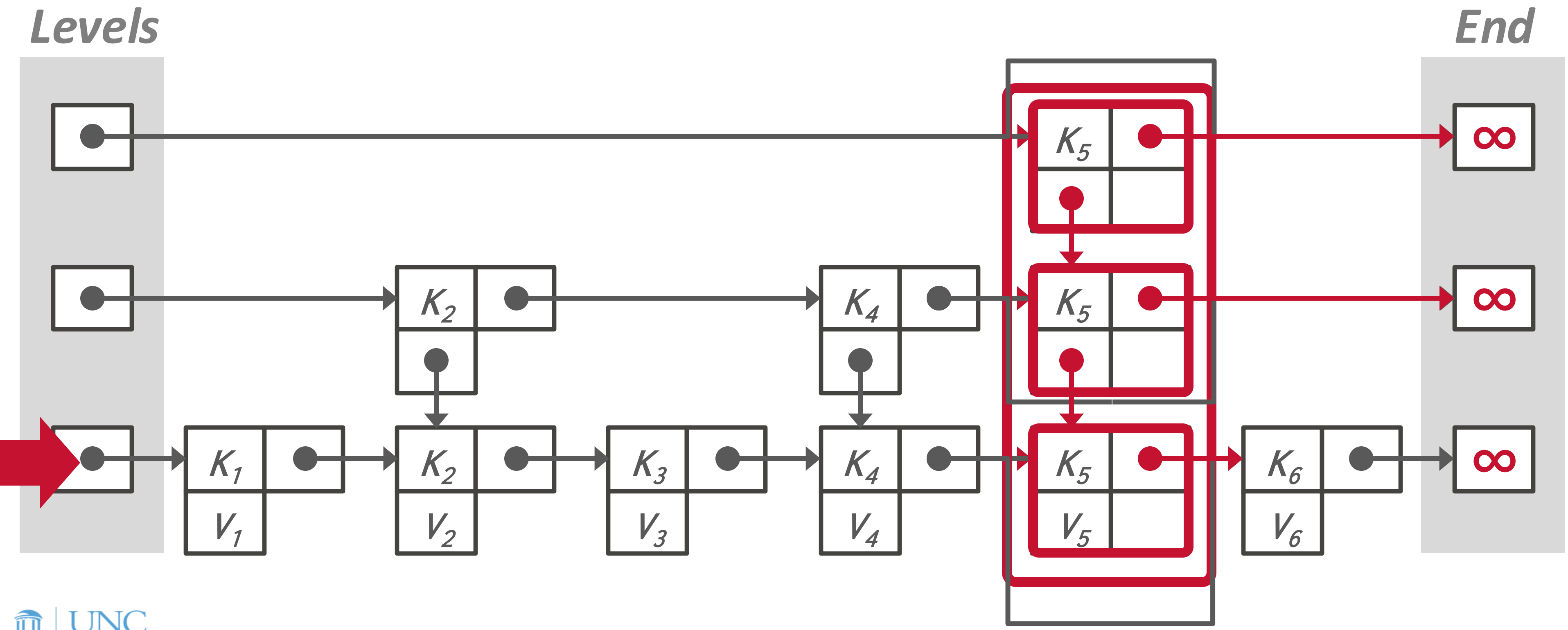
Flip a coin w/ prob p to decide how many levels to add the new key into.

How many levels in expectation? $\log_{1/p} N$



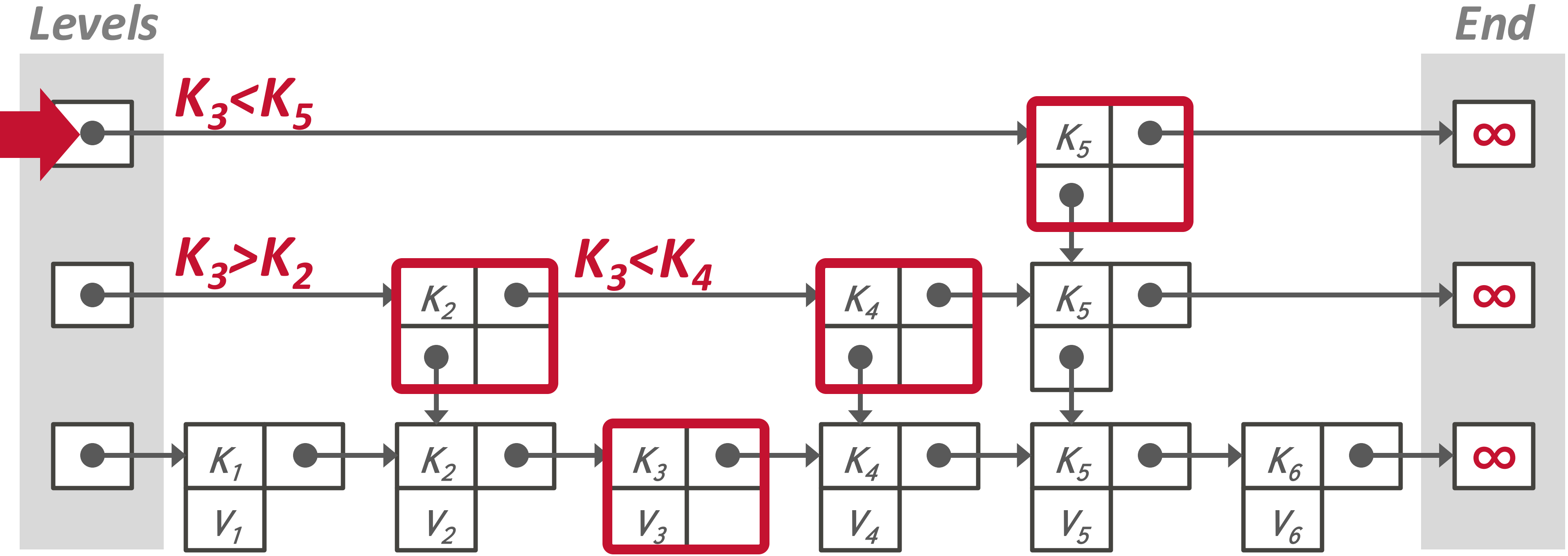
Skip Lists: INSERT

Insert K_5



Skip Lists: SEARCH

Find K_3



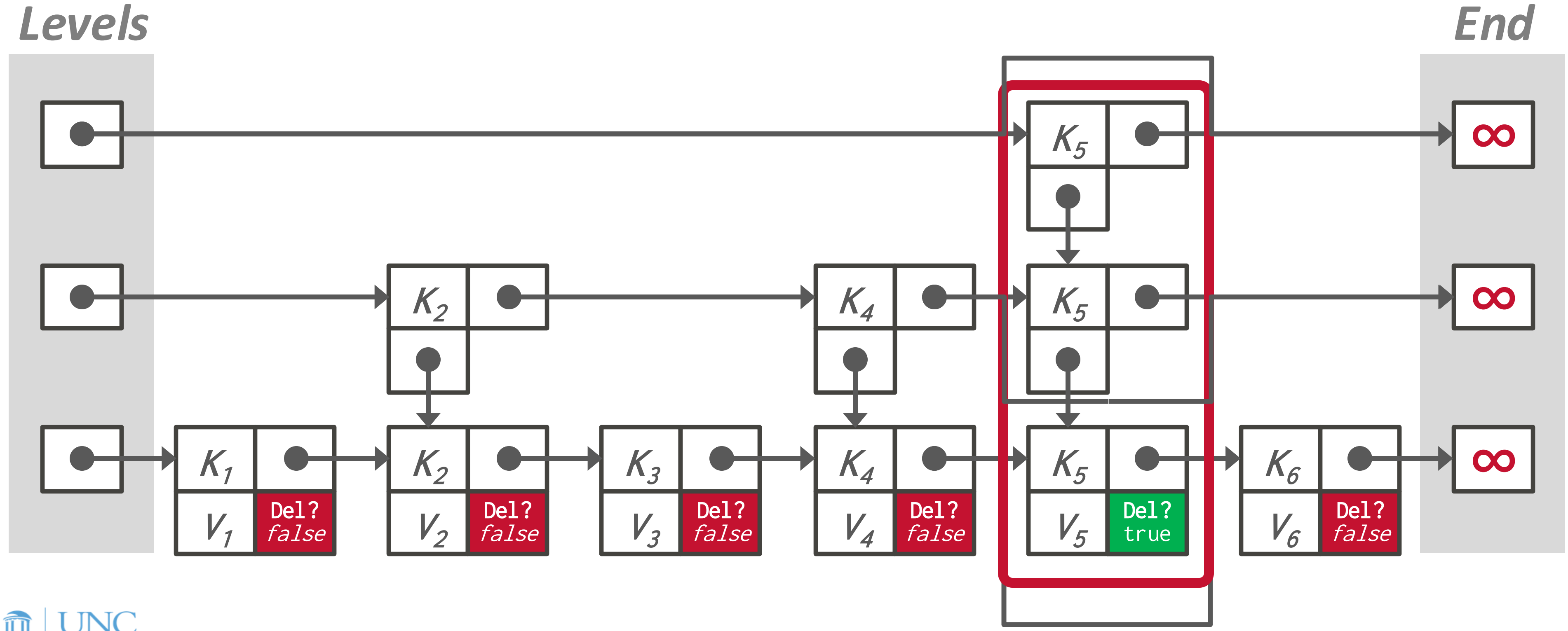
Skip Lists: DELETE

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

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

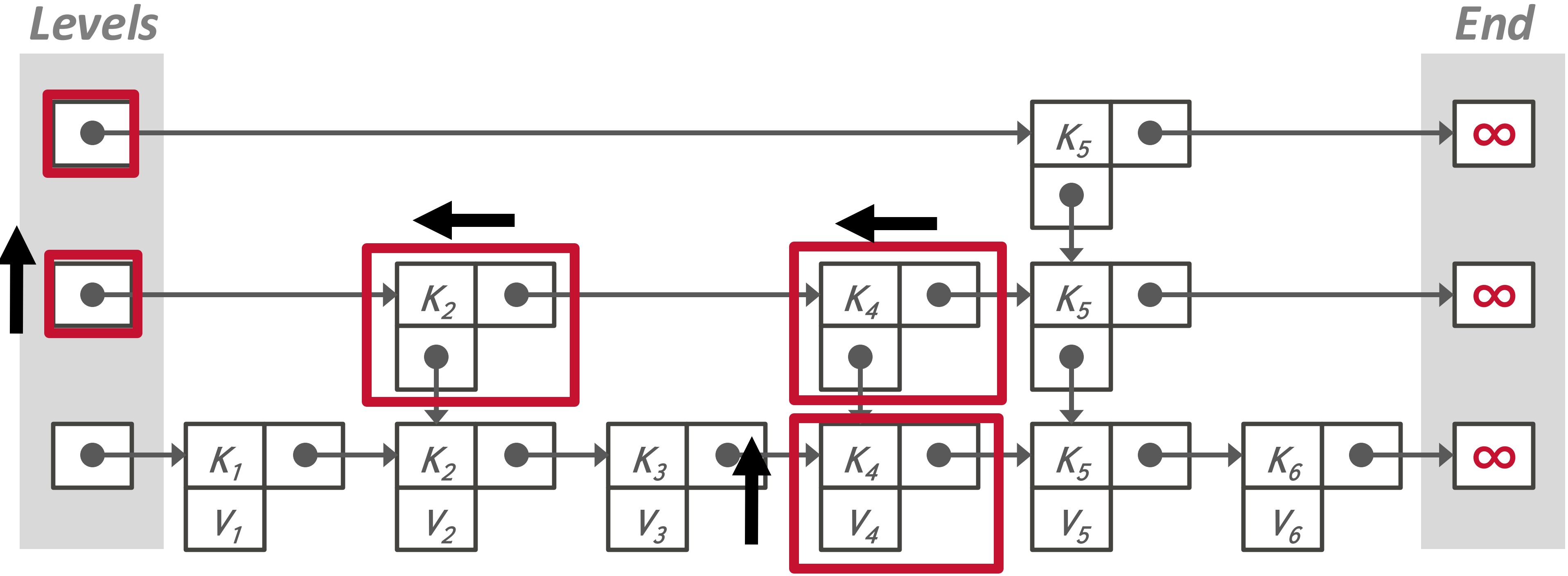
Skip Lists: DELETE

Delete K_5



Skip List Math

Find K_4



Skip List Math

Question: Expected length to traverse a skiplist with N keys and link probability p?

- Let M be random variable, path length

Break up the traversal path (moving backwards):

- Move left until up-link
- Move up one level, repeat
- Let M_i be # left steps before finding an up-link

Main idea: each link is independent, identically distributed w/ prob p

Skip List Math

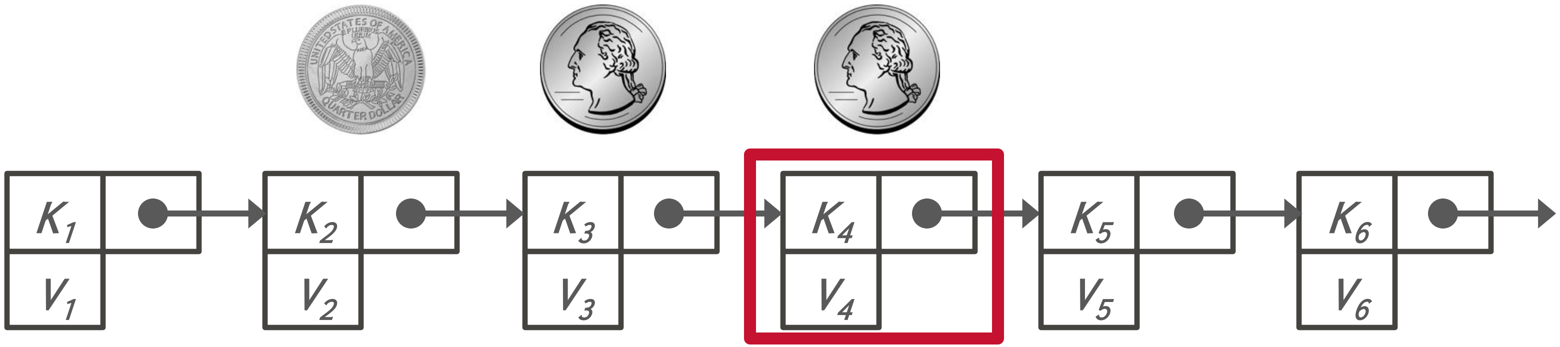
Question: Starting at any node, how many steps left until up-link?

Question: How many p-coin flips to get tails?

$$M_i \sim Geometric(p);\; Pr(M_i = x) = (1-p)^{x-1}\,p$$

$$\mathrm{E}[M_i] = \sum_{x=1}^{\infty} x \cdot (1-p)^{x-1}\,p = \frac{1}{p}$$

Left steps per up-link (expected)



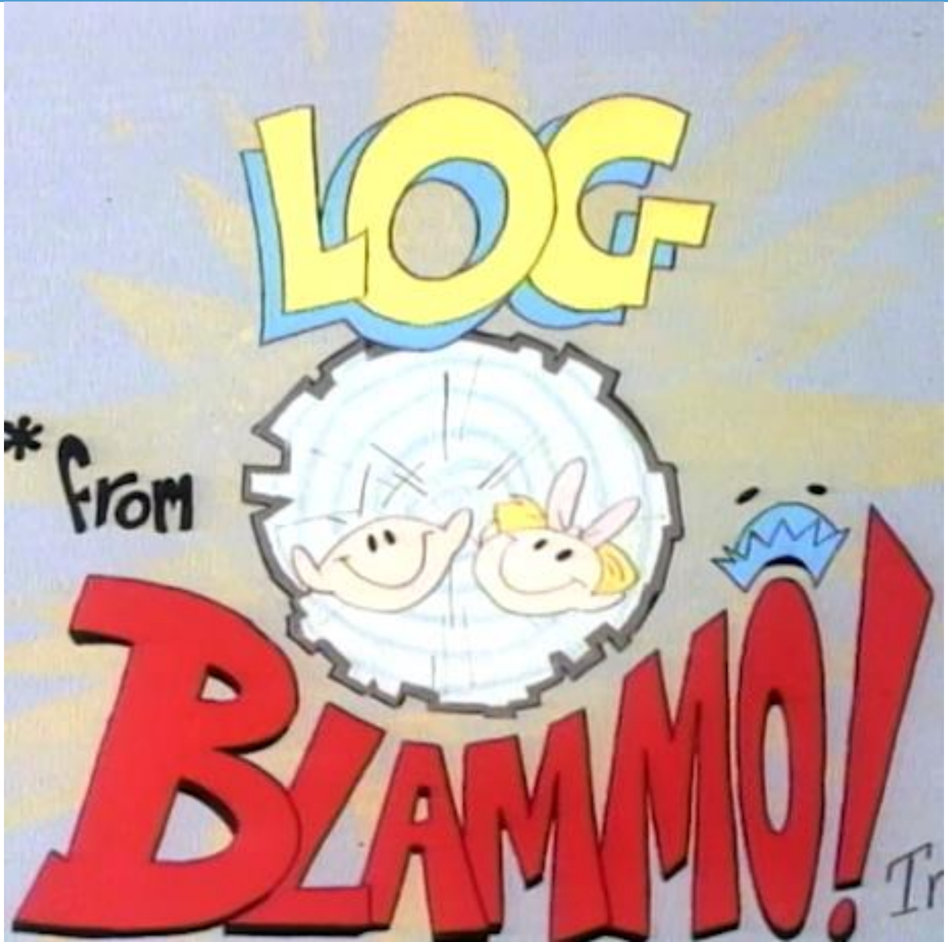
Skip List Math

Question: Expected length to traverse a skiplist with N keys and link probability p?

$$
\begin{aligned}
E[M] &= E[M_1 + M_2 + M_3 + \cdots + M_{L-1}] \\
&\approx E[M_i] \cdot E[\#\ Levels] \\
&= \frac{1}{p} \cdot \log_{\frac{1}{p}} N = O(\log N)
\end{aligned}
$$

In other words...

It's Log!



Skip Lists

Advantages:

→ Uses less memory than a typical B+Tree if you do <u>not</u> 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

Both B+Trees and Skip Lists have the same weakness: Lookup(x) == full traversal

Lookup for keys that don't exist are slow

- "Negative caching", insert tombstone for objects that don't exist

Want: Best of both worlds. An index data structure with filter-like properties

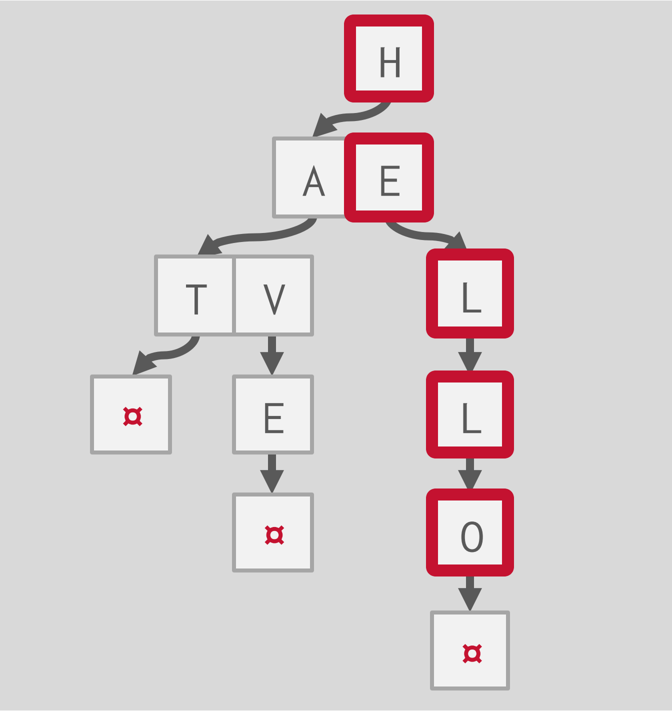
Trie Index

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

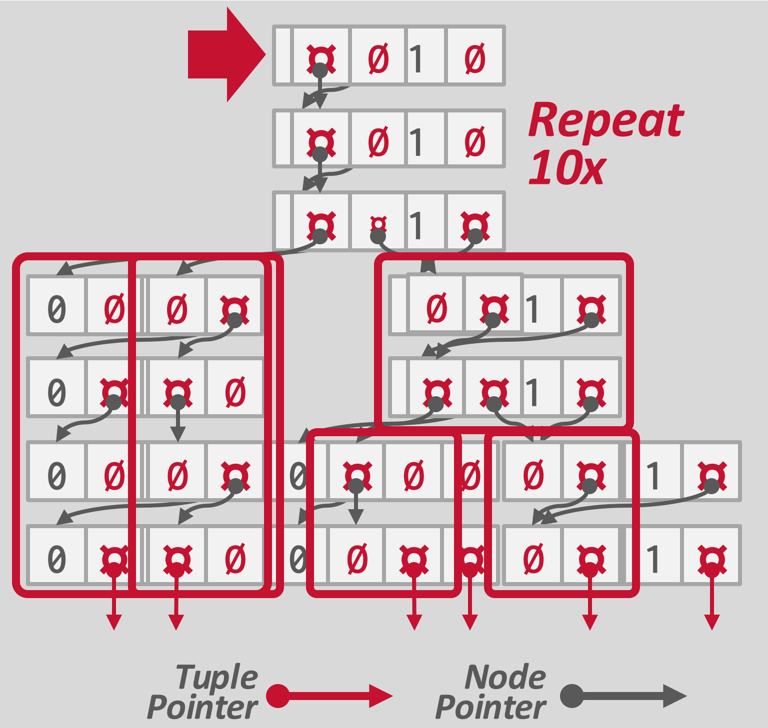
→ If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

This determines the **<u>fan-out</u>** of each node and the physical **<u>height</u>** of the tree.

→ ***n***-way Trie = Fan-Out of ***n***

Trie Key Span

1-bit Span Trie

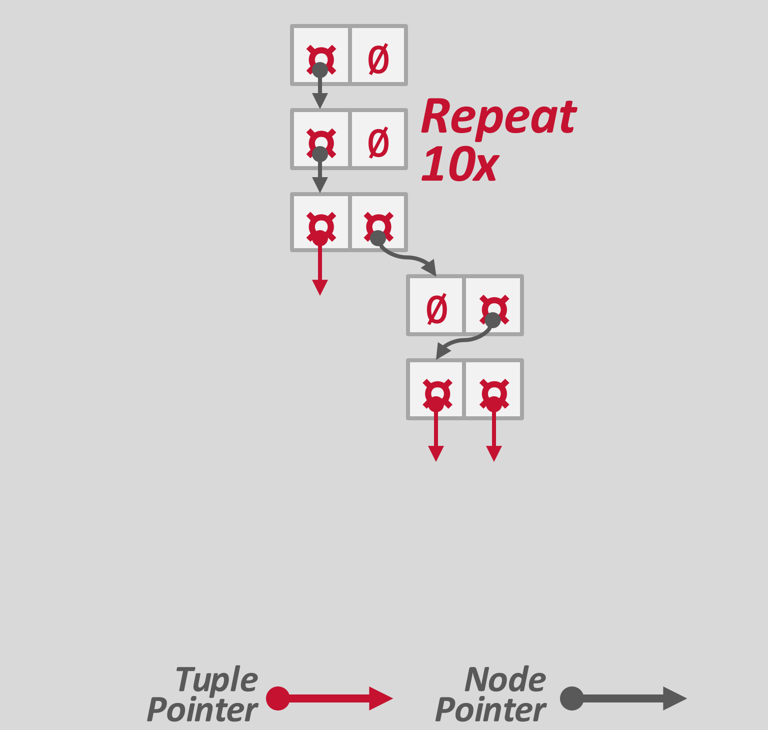


Keys: `K10,K25,K31`

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

Radix Tree

1-bit Span Radix Tree



Vertically compressed trie that compacts nodes with a single child.

→ Also known as ***Patricia Tree***.

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

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.

Lucene, elasticsearch, Solr, Xapian, OpenSearch, Sphinx, UNC Department of Computer Science, tantivy, vespa, splunk>

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...

Term / Frequency

Dictionary	*Posting Lists*
Wu-Tang | 2	11, 44
Carnegie | 3	22, 33, 44
Database | 2	33, 44
⋮	

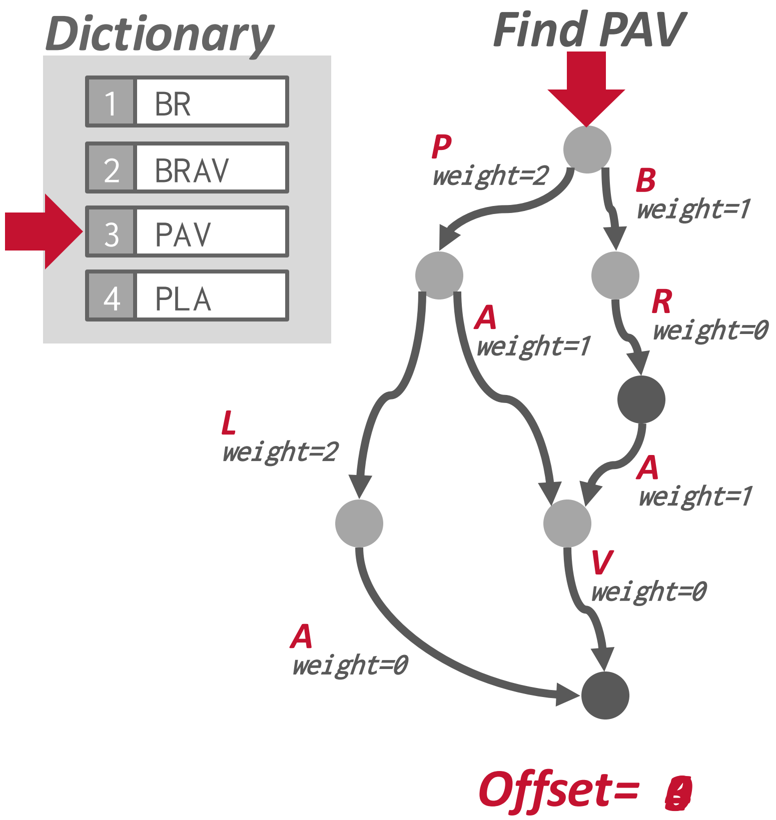
Inverted Index: Lucene

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.



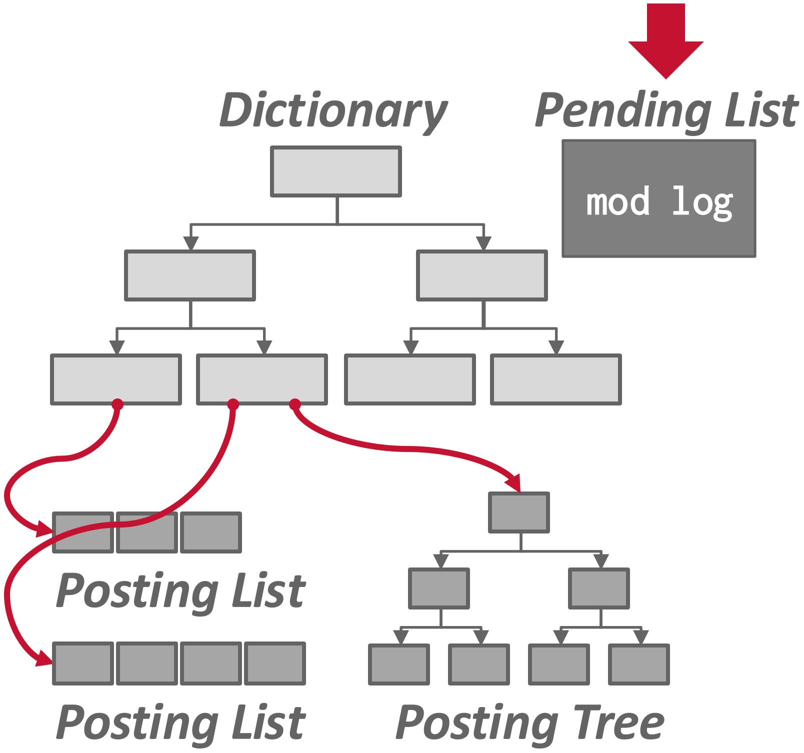
Inverted Index: PostgreSQL

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.



OBSERVATION

Inverted indexes search data based on its contents.

→ There is a little magic to tweak terms based on linguistic models.

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

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

Vector Indexes

Specialized data structures to perform nearest-neighbor searches on **<u>embeddings</u>**.

→ An embedding is an array of floating point numbers.

→ May also need to filter data before / after vector searches.

The correctness of a query depends on whether the result "feels right".

Pinecone Weaviate milvus qdrant

marqo CloseVector VectorDB turbopuffer <(°O°)>

Vector Indexes: Inverted File

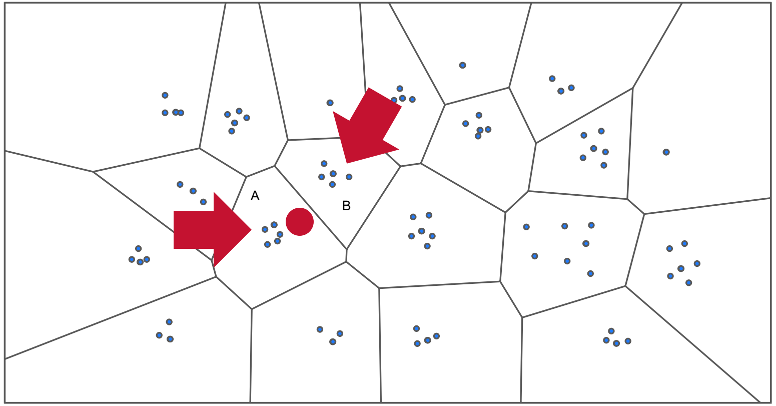
Partition vectors into smaller groups using a clustering algorithm.

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.

Preprocess / quantize vectors to reduce dimensionality.

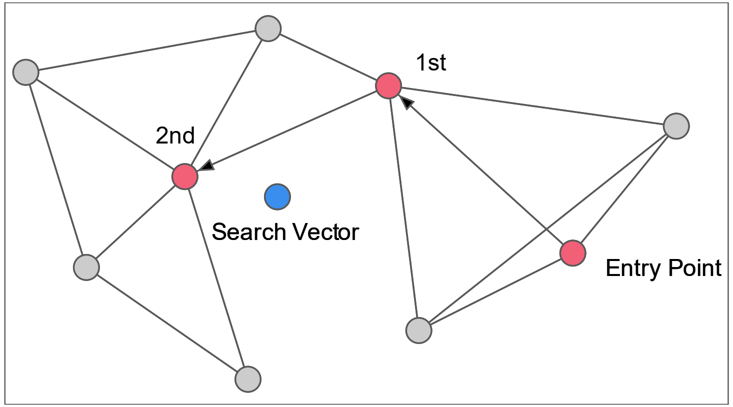
Example: IVFFlat



Source: Chi Zhang

Vector Indexes: Navigable Small Worlds

Build a graph where each node represents a vector and it has edges to its ***n*** nearest neighbors.
→ Can use multiple levels of graphs (HNSW)



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

Example: Faiss, hnswlib

Source: Chi Zhang

Conclusion

We will see filters again this semester.

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

We did not discuss geo-spatial tree indexes:
→ Examples: R-Tree, Quad-Tree, KD-Tree

Next Class

How to make indexes thread-safe!