ClickHouse

Inside ClickHouse full-text search: fast, native and columnar

Elmi Ahmadov, Software Engineer @ ClickHouse

11.11.2025

What is ClickHouse?

Your (soon-to-be) favorite database!

Open source	Column-oriented	Distributed	OLAP database
• Development started 2009	• Best for aggregations	• Replication	• Analytics use cases
• Production 2012	• Files per column	• Sharding	• Aggregations
• OSS 2016	• Sorting and indexing	• Multi-master	• Visualizations
	• Background merges	• Cross-region	• Mostly immutable data

Agenda

01 **Use case**
What is the problem?

02 **Full-text search**
Building blocks

03 **Demo**

04 **Search**
Direct read optimization

ClickHouse

01 Use case

What is the problem?

Example use case: Observability

```
CREATE TABLE logs (
      `id` UInt64,
      `timestamp` DateTime64,
      `message` String
)
ENGINE = MergeTree
ORDER BY `timestamp`;

SELECT count() FROM logs WHERE message LIKE '%9157e9d3-ea7d-45e6-9ea6-2197781204d4%';

SELECT count() FROM logs WHERE hasAllTokens(message, '9157e9d3-ea7d-45e6-9ea6-2197781204d4');
```

Existing bloom filter based skip index

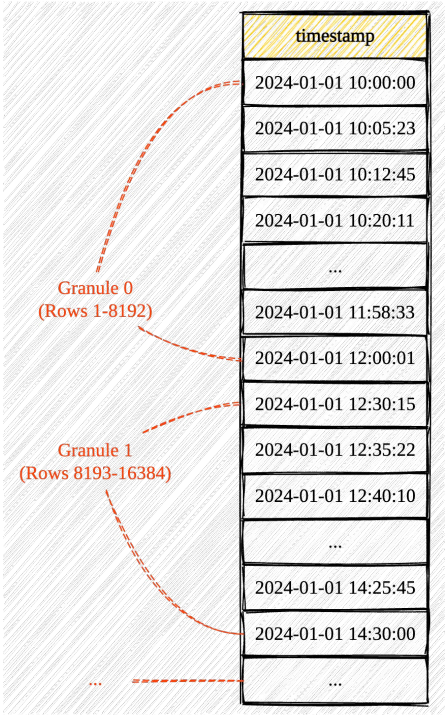
```
CREATE TABLE logs (
      `id` UInt64,
      `timestamp` DateTime64,
      `message` String,
      INDEX idx_message(message) TYPE tokenbf_v1(32768, 8, 128)
)
ENGINE = MergeTree
ORDER BY `timestamp`;
```

What is a bloom filter?

- A Bloom filter is a space-efficient, probabilistic data structure used to test whether an element is a member of a set of size N.
- It can determine with certainty that an element is not in the set (no false negatives), but can only indicate that an element is possibly present - allowing for false positives.

What is a granule?

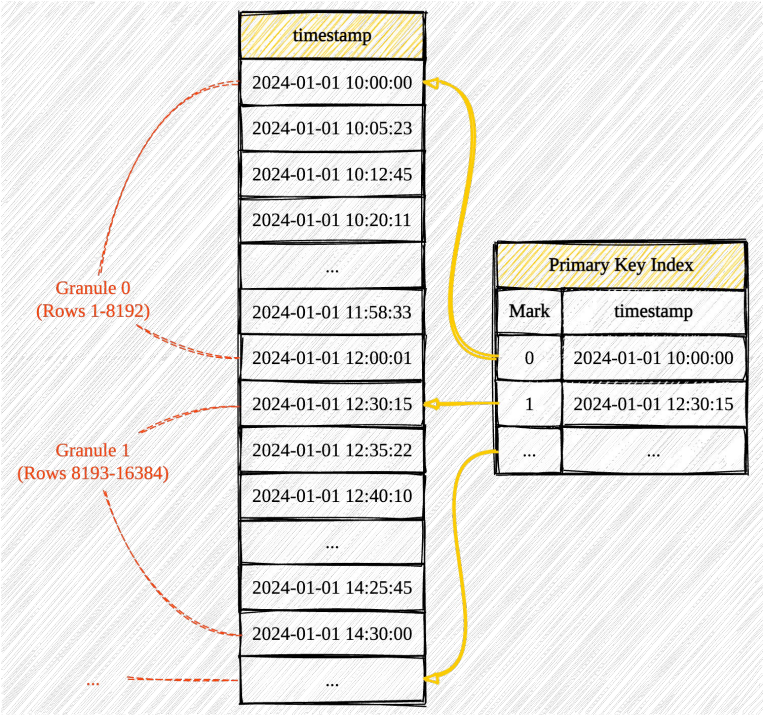
- A granule represents the smallest indivisible data unit processed by the scan and index lookup operators in ClickHouse.
- The rows of a part are further logically divided into groups of 8192 records, called granules.



ClickHouse

What is a skip index?

- Skip index allows to skip reading granules.
- Avoid reading significant chunks of data that are guaranteed to have no matching values.



ClickHouse

Limitations

- **Hard to tune:** Using a Bloom filter requires manual tuning of the byte size, the number of hash functions, and an understanding how they affect the false positive rate. This often requires deep expertise

- **False positives:** Bloom filters may answer that a value might exist even when it doesn’t. This means that additional rows have to be scanned, reducing efficiency of the search.

- **Data read:** Bloom filters just skips the granules, afterwards the granule data needs to be read for matching.

ClickHouse

02

Solution: Full-text search

Building blocks

The new text index

```
CREATE TABLE hackernews (
  ...
  INDEX idx_text(text) TYPE text(
       tokenizer = splitByNonAlpha | splitByString(['::', ' ']) | ngrams(N) | sparseGrams(min, max) | array
  )
)
ENGINE = MergeTree
ORDER BY `time`;

SELECT count() FROM hackernews WHERE hasAnyTokens(text, ['clickhouse', 'amsterdam']);

SELECT count() FROM hackernews WHERE hasAllTokens(text, ['clickhouse', 'amsterdam']);
```

New functions

- Two new functions have been introduced
  - **hasAnyTokens**, finds columns containing any search terms

```
SELECT count() FROM hackernews WHERE hasToken(text, 'clickhouse') OR hasToken(text, 'munich');

SELECT count() FROM hackernews WHERE hasAnyTokens(text, ['clickhouse', 'munich']);
```

  - **hasAllTokens**, finds columns containing all search terms

```
SELECT count() FROM hackernews WHERE hasToken(text, 'clickhouse') AND hasToken(text, 'munich');

SELECT count() FROM hackernews WHERE hasAllTokens(text, ['clickhouse', 'munich']);
```

Full-text search

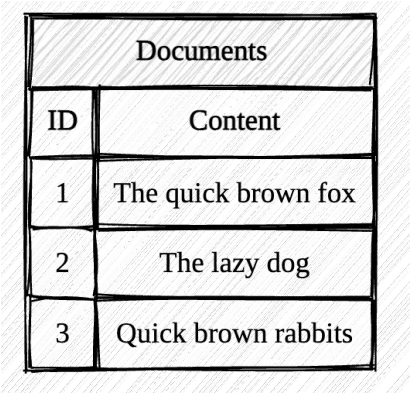
- ClickHouse's full-text search is powered by a **native inverted index**, also known as a *text index*.
- Conceptually, this index consists of two key components:
  - A term dictionary, storing all unique terms across all documents.
  - Posting lists, which record the row numbers of documents that contain each term.

Inverted index

- When you read multiple documents from beginning to end, you can remember for each document the (unique) terms it contains.
  This **document → terms** mapping is sometimes called **forward index**.

- An **inverted index** search flips that around: It stores a **term → documents** mapping. Given a term, it allows you to find all documents that contain it.

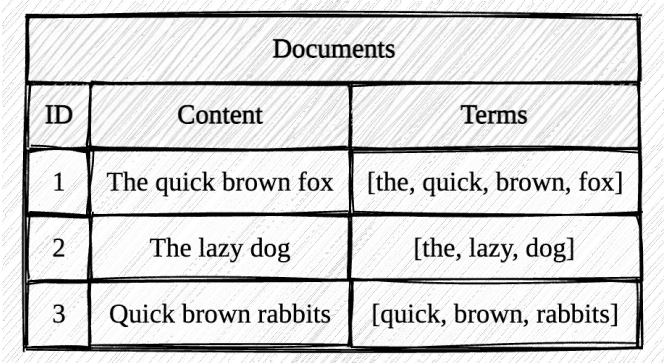
- It's like the back of a book: instead of reading pages to find a word, you look up the word to see which pages it's on. That reverse lookup is what makes search engines fast, and it's built into ClickHouse.



Documents	
ID	Content
1	The quick brown fox
2	The lazy dog
3	Quick brown rabbits

Inverted index

- Forward Index (**document → terms**)



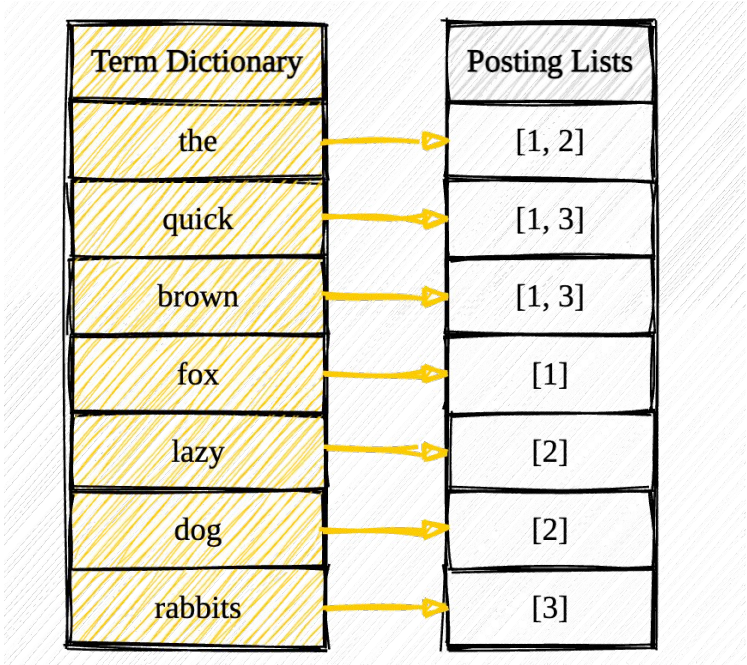
Documents		
ID	Content	Terms
1	The quick brown fox	[the, quick, brown, fox]
2	The lazy dog	[the, lazy, dog]
3	Quick brown rabbits	[quick, brown, rabbits]

Inverted index

- Forward Index (**document → terms**)

Documents		
ID	Content	Terms
1	The quick brown fox	[the, quick, brown, fox]
2	The lazy dog	[the, lazy, dog]
3	Quick brown rabbits	[quick, brown, rabbits]

- Inverted index (**term → documents**)



ClickHouse

How the data is organized

- While building an index, we maintain a hash table to store a **term** → **posting list** mapping.
- When flushed to disk, an inverted index consists of three files on disk:
  - A file with posting lists
  - A file with dictionary blocks
  - A file with the sparse index

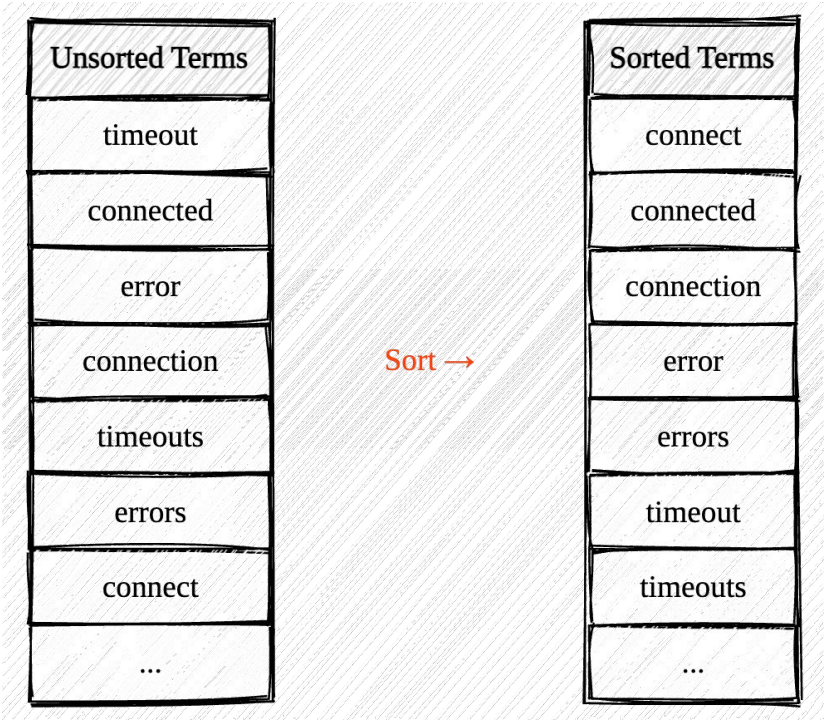
Posting list

- To keep things fast, ClickHouse uses **Roaring bitmaps**, a modern, high-performance format for storing large sets of integers.
- The idea is simple:
  - Think of a posting list as a bitmap, e.g., [3, 5, 8] → 000**1**0**1**00**1**
  - Store each chunk in a specialized container, based on its content.
    - Array – for sparse data (few values)
    - Bitmap – for dense data
    - Run-length encoding – for long consecutive sequences

- This design keeps storage compact and enables extremely fast set operations — including all combinations of AND, OR, and NOT between posting lists. Roaring bitmaps also use specialized SIMD-accelerated code paths for maximum speed.

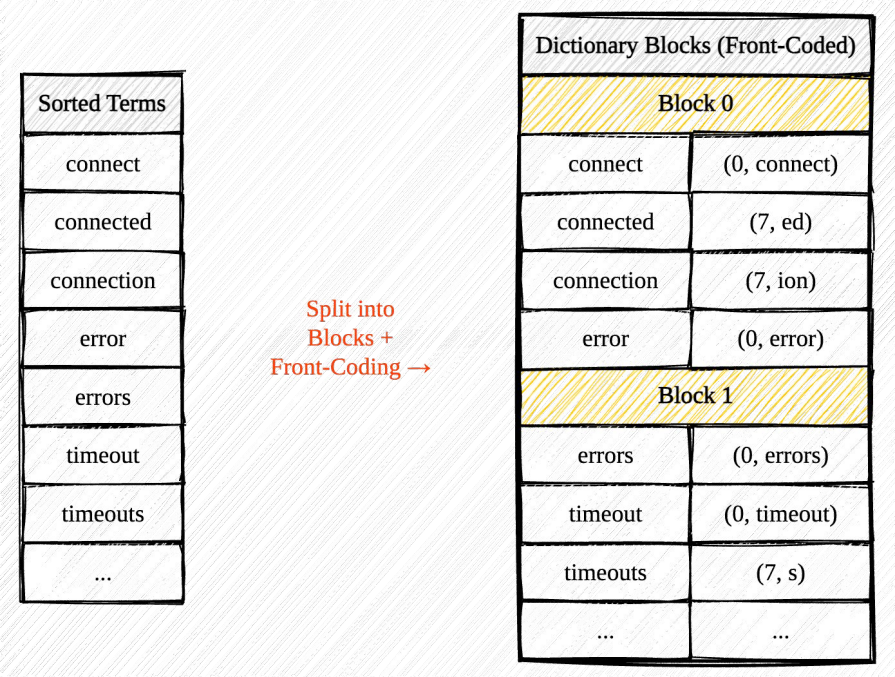
Dictionary blocks & sparse index

- First, all terms are sorted alphabetically.



Dictionary blocks & sparse index

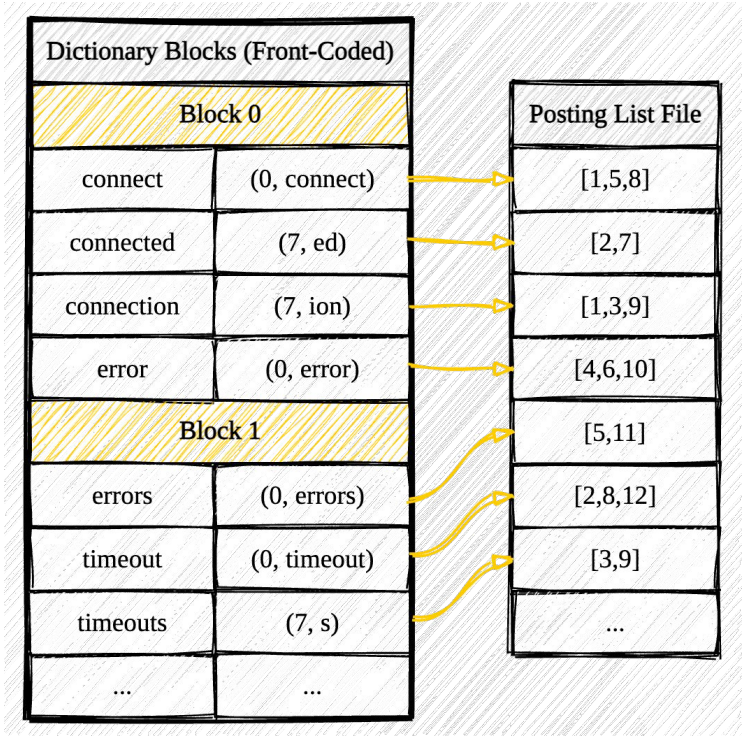
- First, all terms are sorted alphabetically.
- Sorted terms are splitted into blocks:
  - Dictionary blocks are compressed by the front-coding compression.



ClickHouse

Dictionary blocks & sparse index

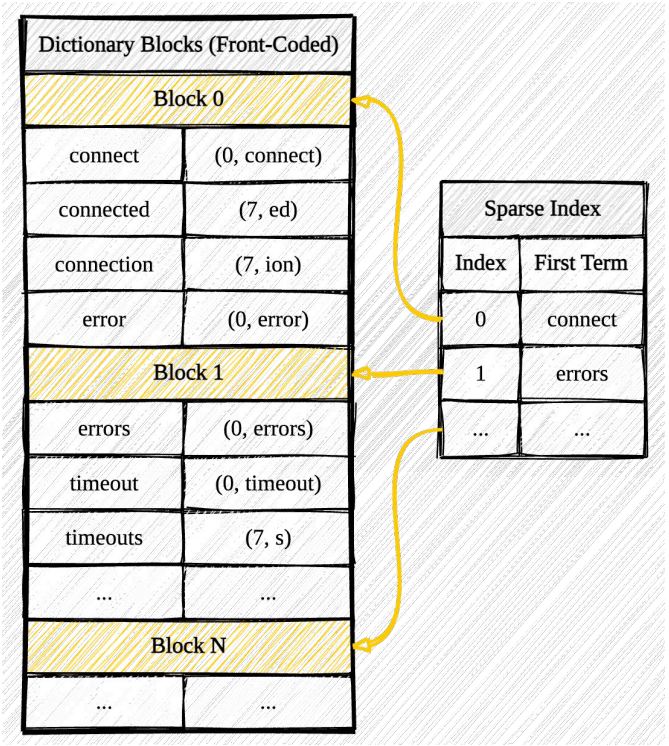
- First, all terms are sorted alphabetically.
- Sorted terms are splitted into blocks:
  - Dictionary blocks are compressed by the front-coding compression.
  - Each term stores to an offset of its postings in the posting list file.



ClickHouse

Dictionary blocks & sparse index

- First, all terms are sorted alphabetically.
- Sorted terms are splitted into blocks:
  - Dictionary blocks are compressed by the front-coding compression.
  - Each term stores to an offset of its postings in the posting list file.
- Sparse index is similar to ClickHouse primary key index:
  - Each sparse index entry points to the beginning of each block by an offset.
  - Additionally, there is a bloom filter on top of the sparse index



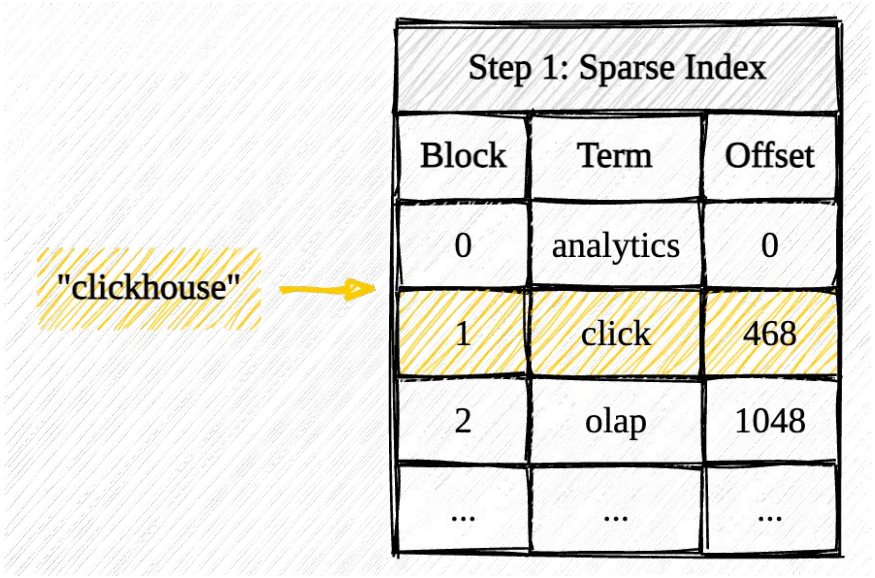
ClickHouse

ClickHouse

03 Demo

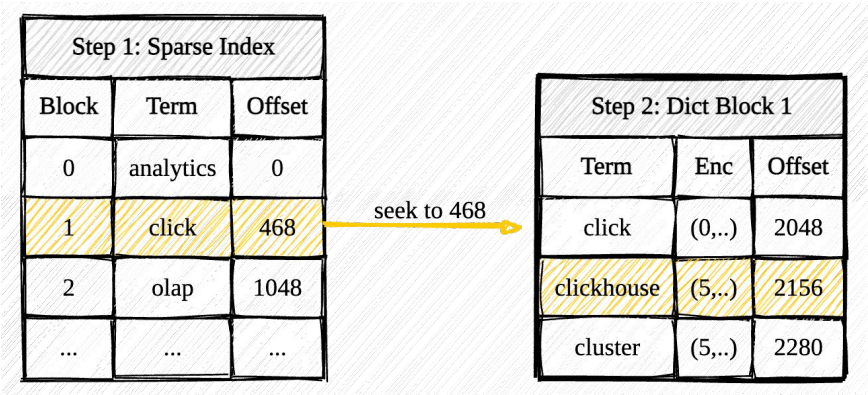
What happened!?

- First we use a binary search on the sparse index to find the upper-bound of the search term.
  - This gives us the dictionary block ID + 1.
  - Get the offset of the dictionary block.



What happened!?

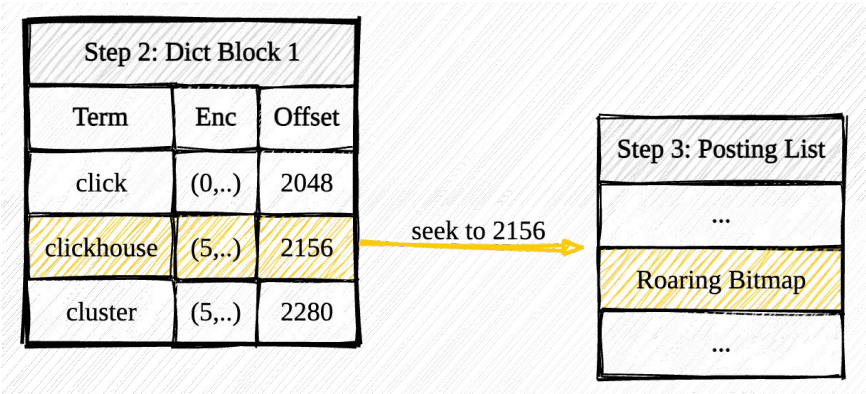
- First we use a binary search on the sparse index to find the upper-bound of the search term.
  - This gives us the dictionary block ID + 1.
  - Get the offset of the dictionary block.
- Search the term in the dictionary block.
  - If present, return the offset.



ClickHouse

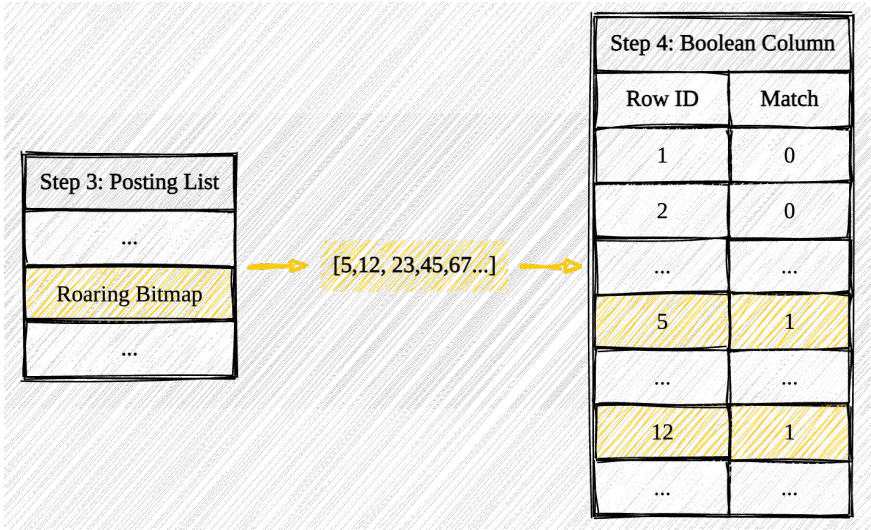
What happened!?

- First we use a binary search on the sparse index to find the upper-bound of the search term.
  - This gives us the dictionary block ID + 1.
  - Get the offset of the dictionary block.
- Search the term in the dictionary block.
  - If present, return the offset.
- Read posting list from the given offset.



What happened!?

- First we use a binary search on the sparse index to find the upper-bound of the search term:
  - This gives us the dictionary block ID + 1.
  - Get the offset of the dictionary block.
- Search the term in the dictionary block:
  - If present, return the offset.
- Read posting list from the given offset.
- Create a virtual boolean column:
  - Doc IDs from posting list get 1 (true)
  - others get 0 (false)



ClickHouse

PREWHERE optimization - direct read

- Apply the special **PREWHERE** optimization and let ClickHouse does its magic

```
SELECT count() FROM hackernews WHERE hasAnyTokens(text, ['clickhouse']);
```



```
-- hasAnyTokens_text_virt_column is filled during the optimization process.
SELECT count() FROM hackernews PREWHERE hasAnyTokens_text_virt_column = 1;
```

New functions & Posting lists

- Using bitmap as a posting list has multiple benefits:
  - **hasAnyTokens**

```
SELECT count() FROM hackernews WHERE hasAnyTokens(text, ['clickhouse', 'munich']);

VirtualColumn = PostingList(clickhouse) OR PostingList(munich)
```

  - **hasAllTokens**

```
SELECT count() FROM hackernews WHERE hasAllTokens(text, ['clickhouse', 'munich']);

VirtualColumn = PostingList(clickhouse) AND PostingList(munich)
```

I loved it! Can I try it myself?

- Full-text search is available on ClickHouse OSS since v25.10.
- We offer a private preview for our ClickHouse Cloud customers.

Your search ends here

Thanks! Questions?

ClickHouse