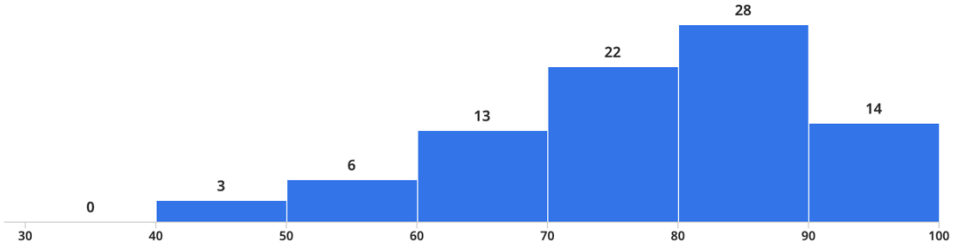
CS 6400 A

Database Systems Concepts and Design

Lecture 11
10/07/26

Announcements



- Midterm grades released
  - Max: 98, Median: 79.5, Mean: 77.3, Standard deviation: 12.5
  - Answer key on Canvas
  - Regrade request open on Gradescope until next Wednesday (Oct 14)
- Assignment 2 released
  - Due Oct 28. **Start early!**
  - Extra credit (10%) awarded for top 3 entries

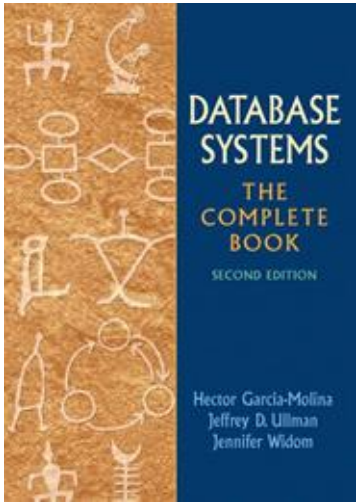
Agenda

1. Index overview
2. Using Index in SQL
3. Index structure basics

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 8.3: Indexes in SQL
- Chapter 14.1: Index-Structure Basics



Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang and CS145 (Intro to Big Data Systems) taught by Peter Bailis.

1. Index Overview

Index Motivation

Person(name, age)

Suppose we want to search for people of a specific age

First idea: Sort the records by age… we know how to do this fast!

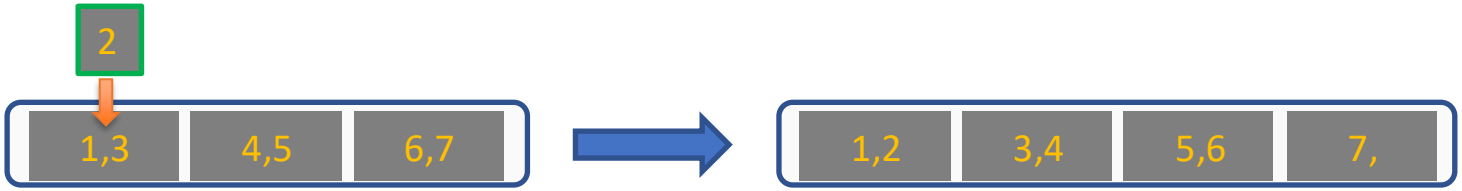
How many IO operations to search over *N sorted* records?

- Simple scan: *O(N)*
- Binary search: *O*($\log_2 N$)

Could we get even cheaper search? E.g. go from $\log_2 N \rightarrow \log_{200} N$?

Index Motivation

What about if we want to **insert** a new person, but keep the list sorted?



We would have to potentially shift ***N*** records, requiring up to ~ **2*N/P** IO operations (where P = # of records per page)!

- We could leave some “slack” in the pages…

Could we maintain sorted data without shifting all subsequent records?

Index Motivation

What about if we want to be able to search quickly along multiple attributes (e.g. not just age)?

- We could keep multiple copies of the records, each sorted by one attribute set… this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

We'll create separate data structures called indexes to address all these points

Further Motivation for Indexes: NoSQL!

NoSQL engines are (basically) ***just indexes!***

- A lot more is left to the user in NoSQL… one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!
- Example indexes: B+ Trees, hash indexes, LSM Tree

Indexes are critical across all DBMS types

Indexes: High-level

An *index* on a file speeds up selections on the *search key fields* for the index.

- Search key properties
  - Any subset of fields
  - is **not** the same as *key of a relation*

Example:

Product(name, maker, price)

On which attributes would you build indexes?

More precisely

An *index* is a **data structure** mapping search keys to sets of rows in a database table

- Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table

Indexes can be clustered or unclustered:

- Clustered index: Data records are physically organized in the same order as the index keys.
- Unclustered index: Data records are not necessarily stored in index-key order; index entries contain pointers to the records.

Operations on an Index

Search: Quickly find all records which meet some *condition on the search key attributes*

- Point queries, range queries, …

Insert / Remove entries

- Bulk Load / Delete.

Indexing is one the most important features provided by a database for performance

Overview of Common Index Types

B+ Tree and its variants *(next Monday)*

- The default index in most RDBMS
- Good for point and range queries; maintains sorted order

Hash Tables *(next Wednesday)*

- These are **persistent hash tables** designed specifically for disk storage
- Good for point queries

Other specialized indexes:

- Bitmap indexes: for low-cardinality data
- Multi-dimensional indexes: KD-tree, R-tree for GIS data
- LSM-tree: optimized for high write/insert throughput; batches writes and periodically merges sorted data
- Vector indexes: HNSW, IVF-PQ for high-dimensional vector data

2. Using Index in SQL

Using Indexes in SQL

An index is used to efficiently find tuples with certain values of attributes

Selection of indexes

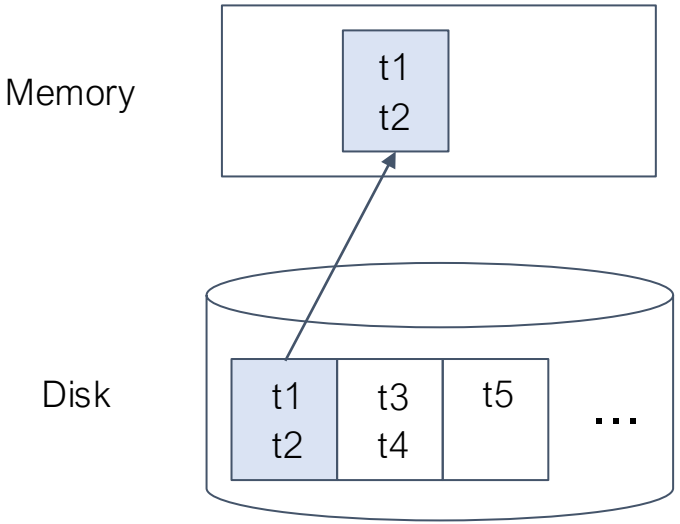
- An index may speed up lookups and joins
- However, every built index makes insertions, deletions, and updates to relation more complex and time-consuming

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

```
DROP INDEX KeyIndex;
```

Recall: Simple I/O based cost model

- Multiple tuples are stored in blocks on disk
- Every block needed is always retrieved from disk
- Disk I/Os are expensive

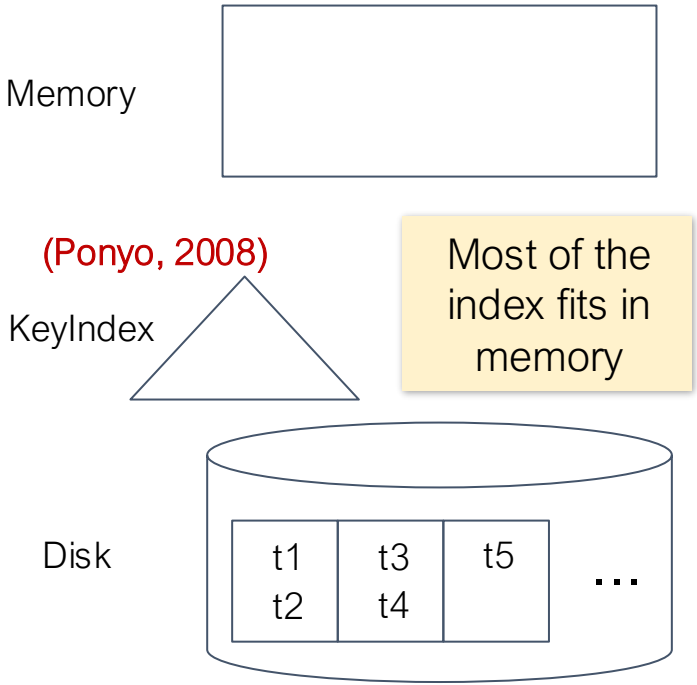


Index on a key

- An index on a key is often useful
- Retrieve at most one block to memory for tuple
  - Possibly other blocks for the index itself

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

```
SELECT *
FROM Movies
WHERE title = 'Ponyo' AND year = 2008;
```

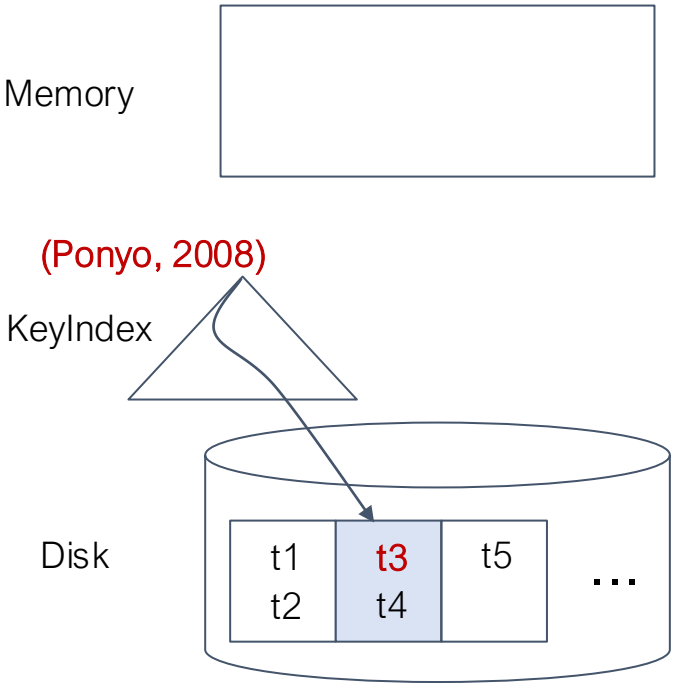


Index on a key

- An index on a key is often useful
- Retrieve at most one block to memory for tuple
  - Possibly other blocks for the index itself

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

```
SELECT *
FROM Movies
WHERE title = 'Ponyo' AND year = 2008;
```

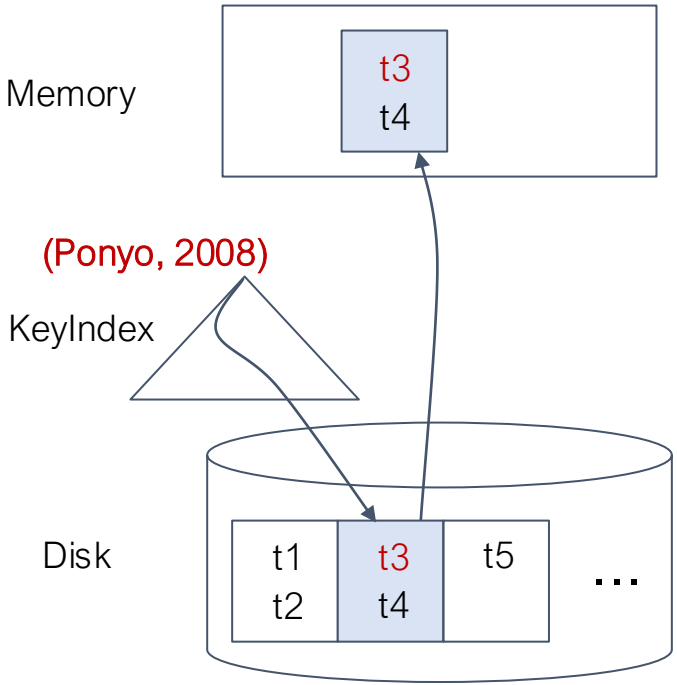


Index on a key

- An index on a key is often useful
- Retrieve at most one block to memory for tuple
  - Possibly other blocks for the index itself

```
CREATE INDEX KeyIndex ON Movies(title, year);
```

```
SELECT *
FROM Movies
WHERE title = 'Ponyo' AND year = 2008;
```



Indexes can be used in joins

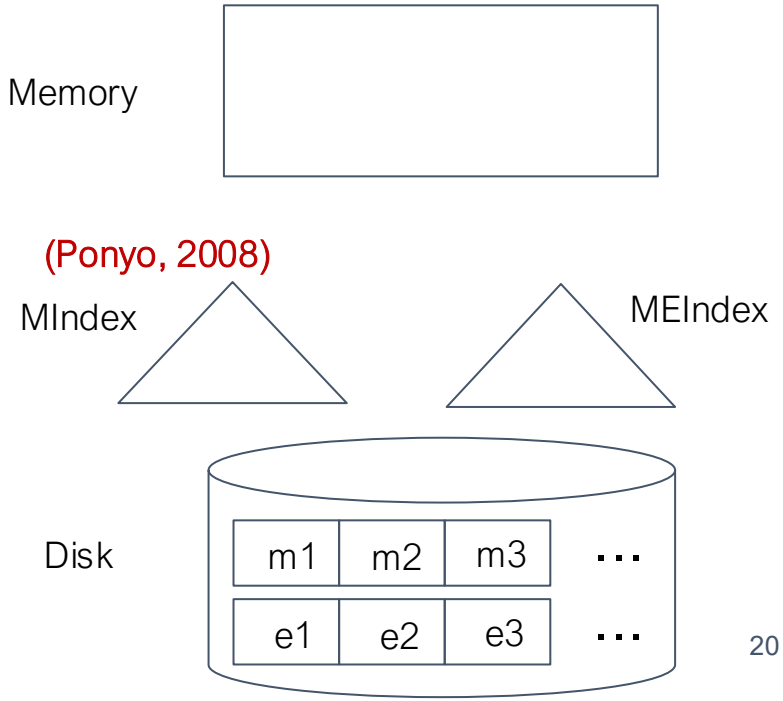
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Ponyo' AND year = 2008
  AND producerC# = cert#;
```



Indexes can be used in joins

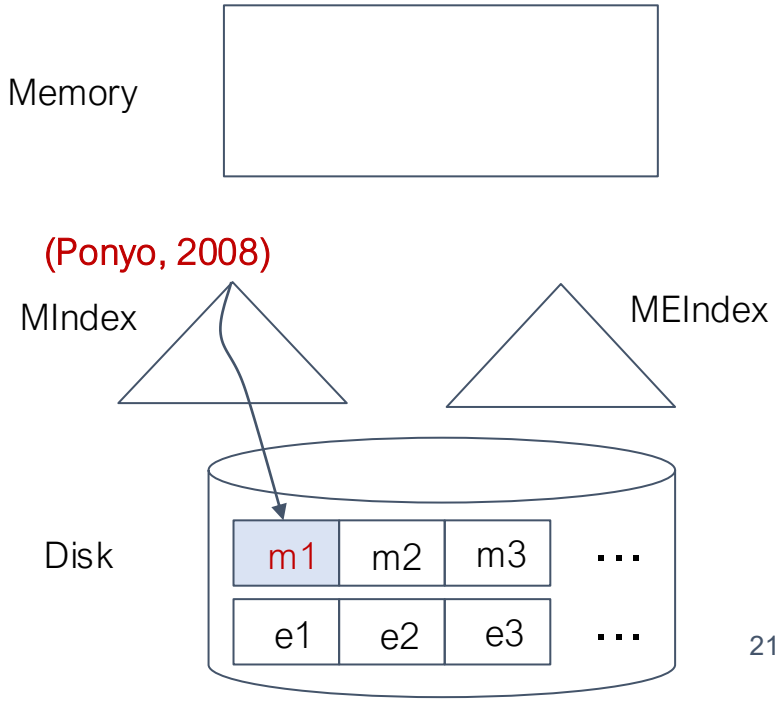
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Ponyo' AND year = 2008
  AND producerC# = cert#;
```



Indexes can be used in joins

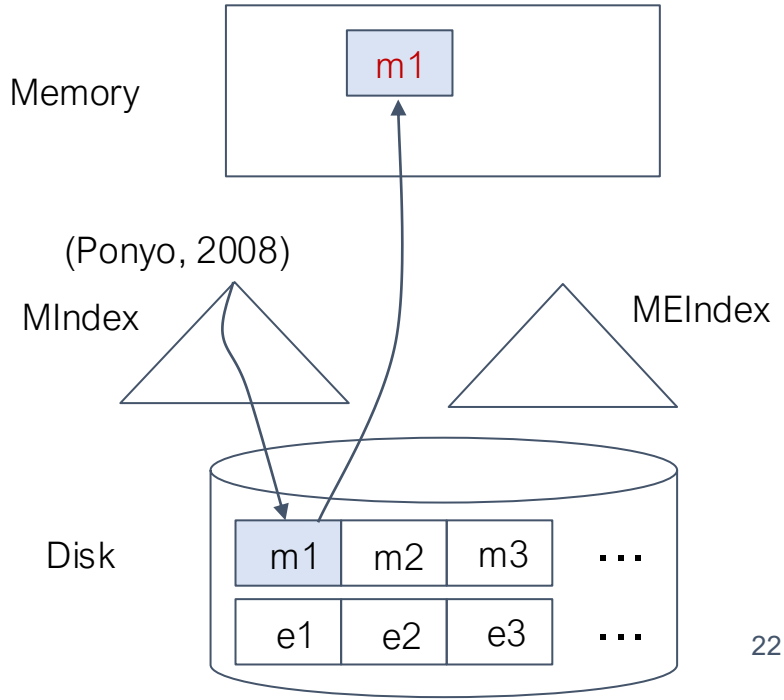
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Ponyo' AND year = 2008
  AND producerC# = cert#;
```



Indexes can be used in joins

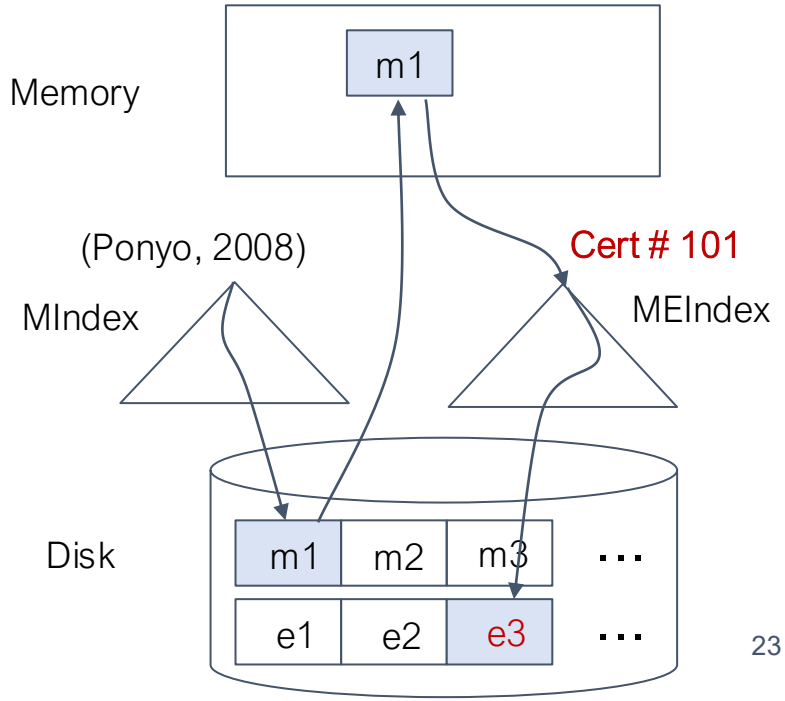
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Ponyo' AND year = 2008
  AND producerC# = cert#;
```



Indexes can be used in joins

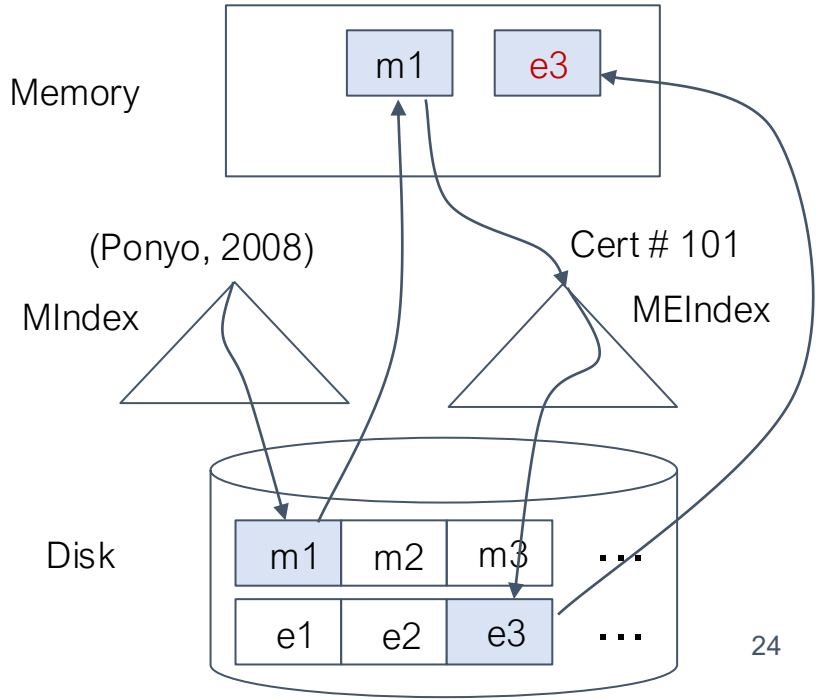
With the right indexes, the join below only requires 2 page reads for the tuples

- And possibly a small number of other pages for accessing the indexes

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MEIndex ON MovieExec(cert#);
```

```
SELECT name
FROM Movies, MovieExec
WHERE title = 'Ponyo' AND year = 2008
  AND producerC# = cert#;
```



Covering Index

We say that an index is *covering* if all the needed columns are in the index.

Needed columns is the union of those in WHERE *and* SELECT.

- Means that the query can be answered using the index--without reading the table!

```
CREATE INDEX MIndex ON Movies(title, year);
```

```
CREATE INDEX MIndex ON
Movies(name, title, year);
```

```
SELECT name
FROM Movies
WHERE title = ‘Ponyo’ AND
year = 2008;
```

Index Activity

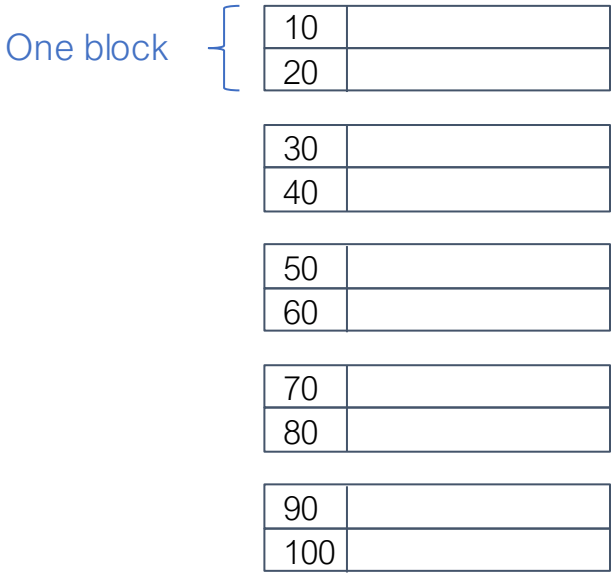
- Collab notebook link: https://tinyurl.com/56m5uuhw
- To try it yourself, make a COPY of the notebook
- Follow the instructions to download the database file and load it in the notebook



3. Index Structure Basics

Sequential file

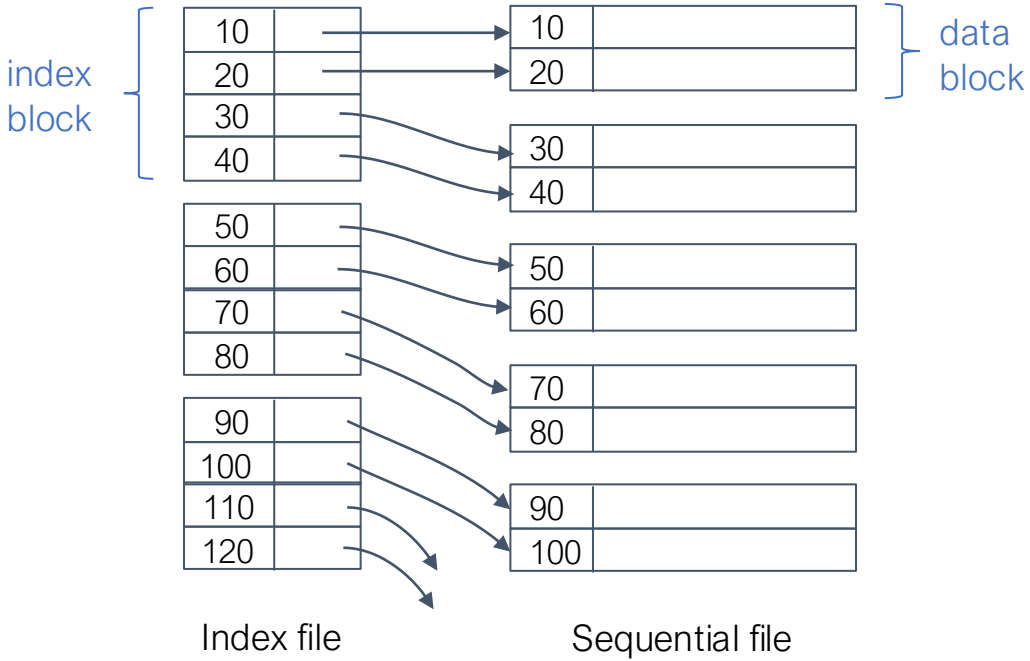
- A file containing tuples of a relation sorted by their primary key



Sequential file

Dense index

- A sequence of blocks holding keys of records and pointers to the records



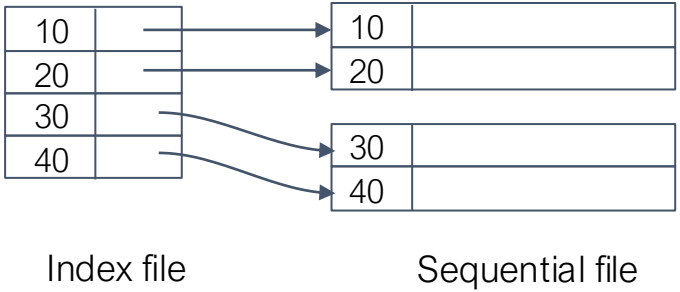
One pointer per record

Dense index

Given key K, search index blocks for K, then follow associated pointer

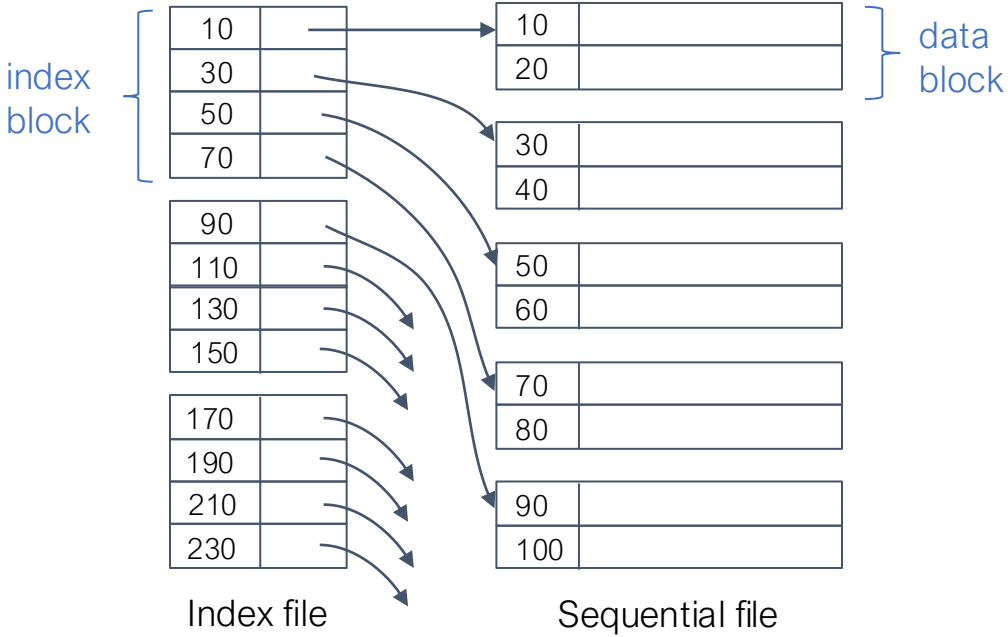
Why is this efficient?

- Number of index blocks usually smaller than number of data blocks
- Keys are sorted, so we can use binary search
- The index may be small enough to fit in memory



Sparse index

- Has one key-pointer pair per block of the data file
- Uses less space than dense index, but needs more time to find a record



One pointer per block

In-class Exercise

Suppose a block holds 3 records or 10 key-pointer pairs

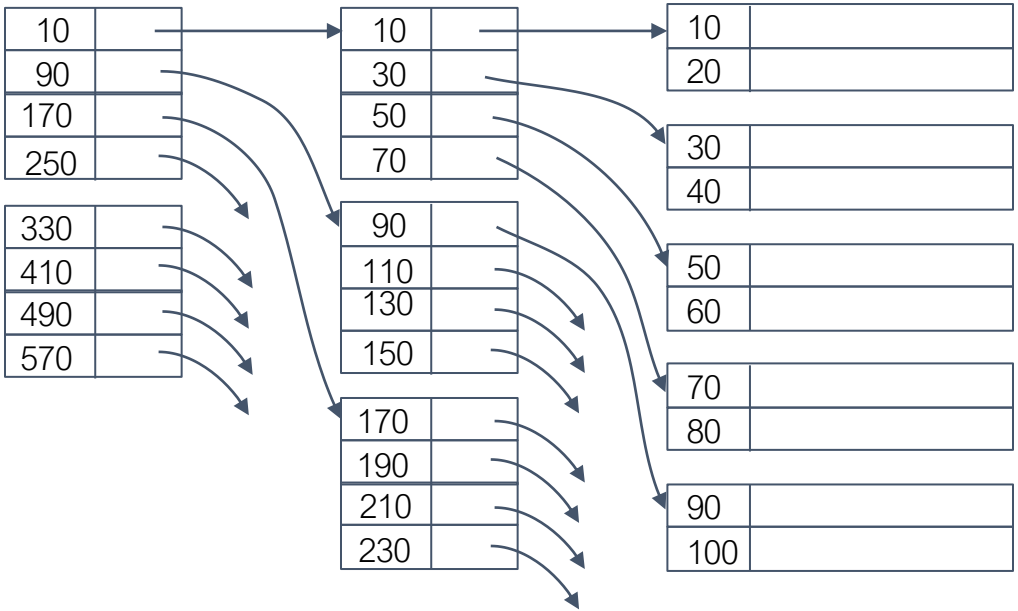
If there are n records in a data file, how many blocks are needed to hold

- The data file and a dense index
- The data file and a sparse index

Multiple levels of index

If the index file is still large, add another level of indexing

- Basic idea of the B+-tree index (next lecture)

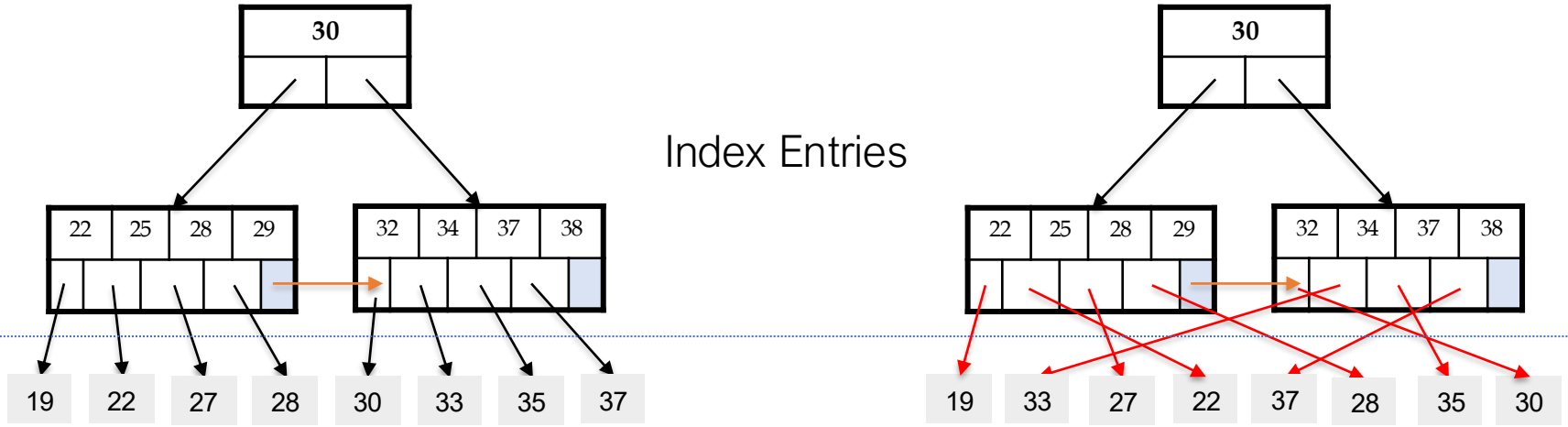


Q: Should the blocks of additional levels be dense or sparse?

Clustered Indexes

An index is clustered if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Clustered: often on primary key

Unclustered

Q: How many clustered/unclustered indexes can a table have?

Clustered vs. Unclustered Index

Recall that for a disk with block access, **sequential IO is much faster than random IO**

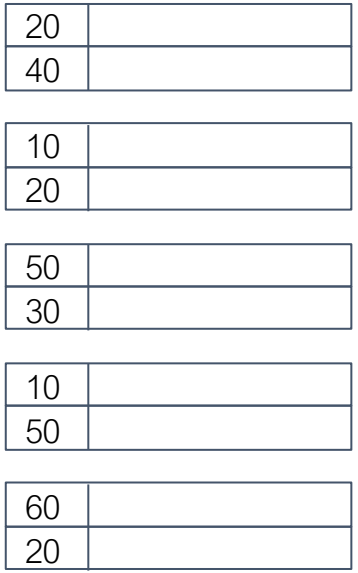
For point lookup, no difference between clustered / unclustered

For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:

- A random IO costs ~ 10ms (sequential much much faster)
- For R = 100,000 records- **difference between ~10ms and ~17min!**

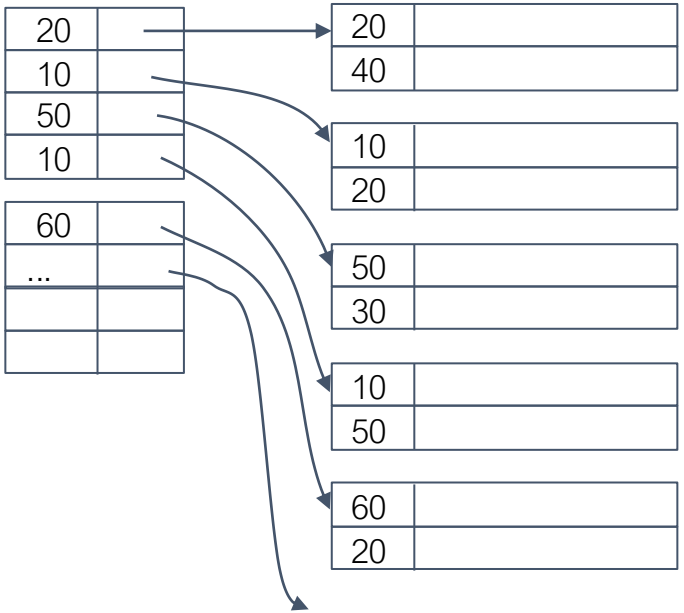
Non-clustered Index

Unlike a clustered index, does not determine the placement of records



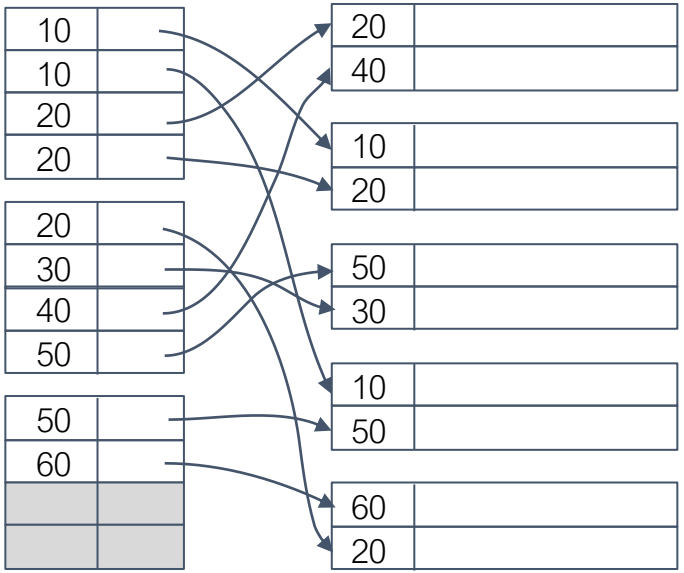
Non-clustered Index

Using a sparse index doesn't make sense



Non-clustered index

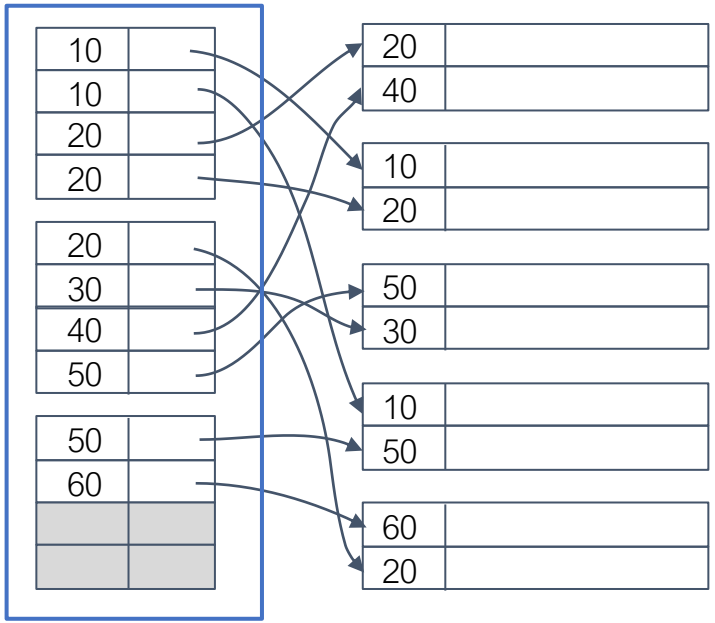
A non-clustered index *must be dense* at the data-entry level.



Non-clustered index

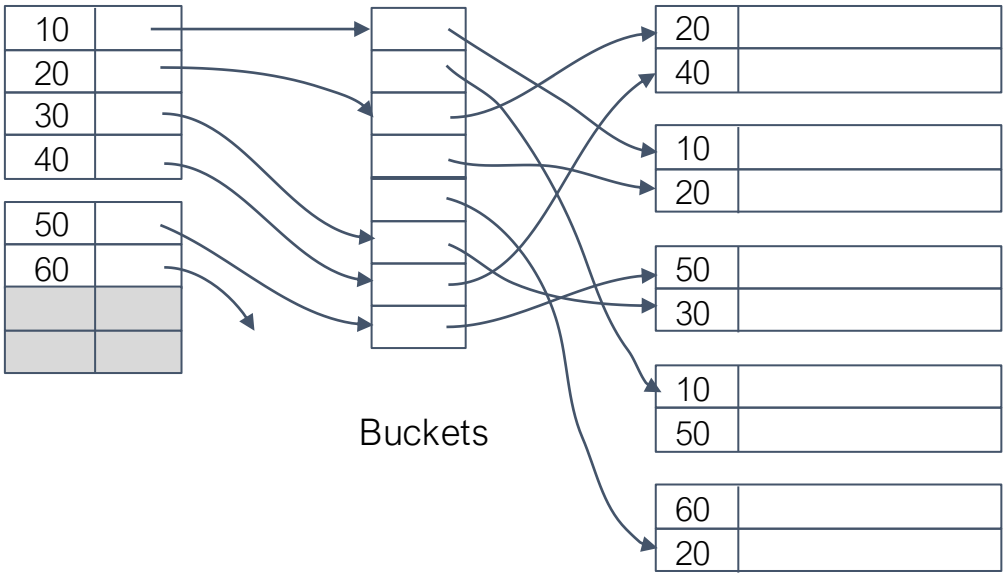
A non-clustered index must be dense at the data-entry level.

But we can build sparse index levels over the sorted index itself.



Non-clustered index

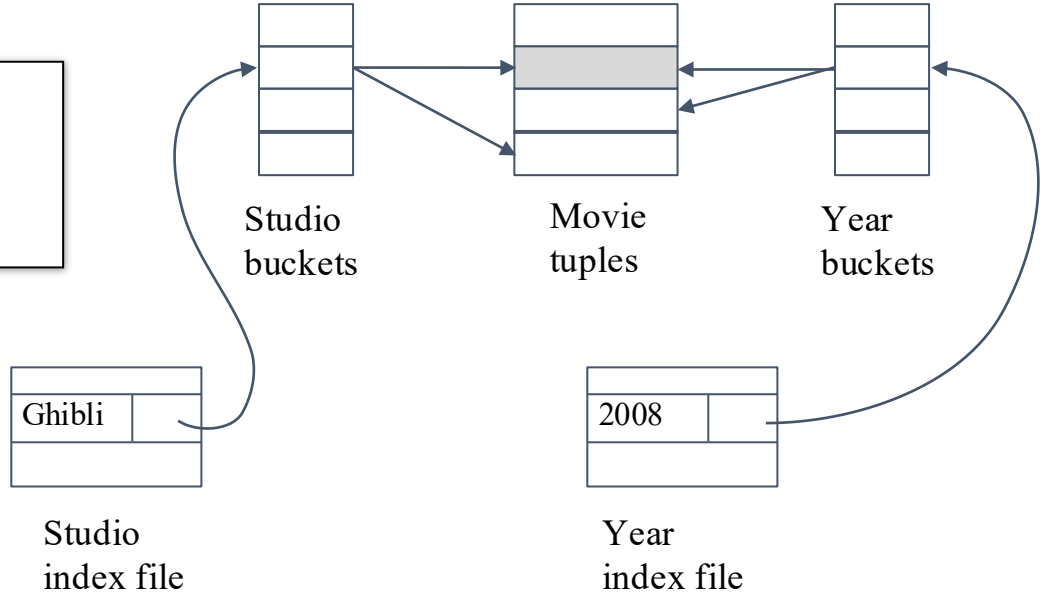
To remove redundant keys in index file, use level of indirection



When is indirection useful?

- When a key is larger than a pointer and each key appears twice on average
- Another advantage: use bucket pointers without looking at most of the records

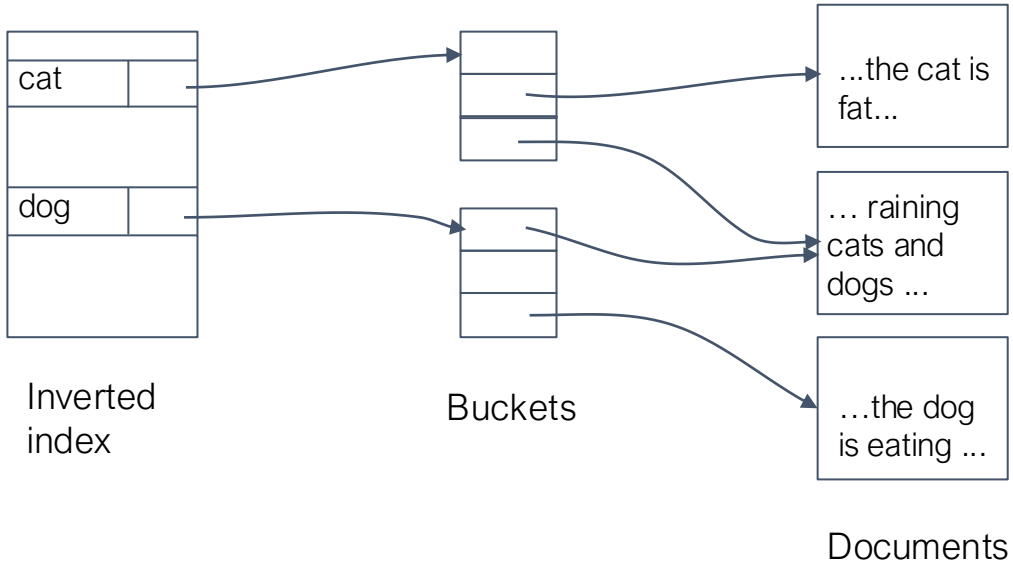
```
SELECT title
FROM Movies
WHERE studioName = 'Ghibli'
   AND year = 2008;
```



Inverted index

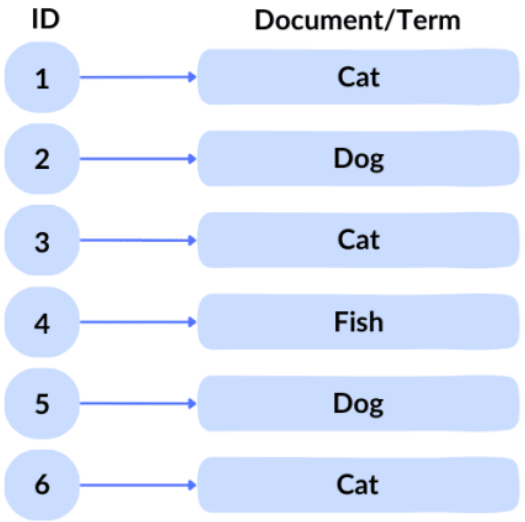
Basically the same structure as the non-clustered index

- e.g., Search for documents containing “cat” or “dog” (or both)



Inverted Index

Where the name came from



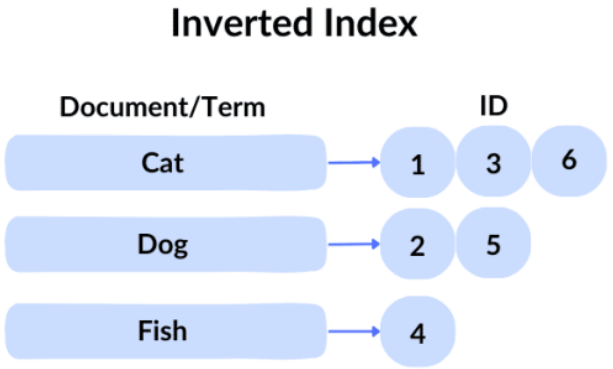


Image source: https://spotintelligence.com/2023/10/30/inverted-indexing/

Store more information in inverted index

Can answer more complex queries like:

- Find documents where “dog” and “cat” are within 10 words
- Find documents about dogs that refer to other documents about cats

