CS 4440 A

Emerging Database Technologies

Lecture 6
01/27/25

Announcements

Assignment 1 due tonight

Project proposal draft due next Monday (Feb 3)

- Ungraded, used for feedback
- Group size: 3~5

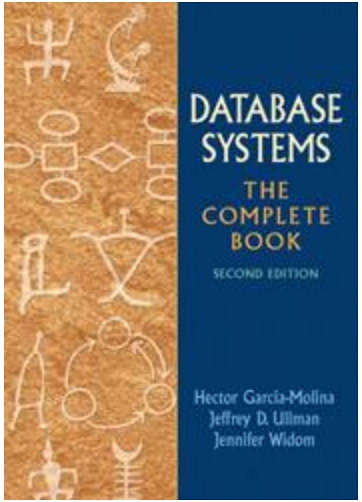
Agenda

1. Index Overview
2. Index structure basics

Reading Materials

Database Systems: The Complete Book (2nd edition)

- 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!

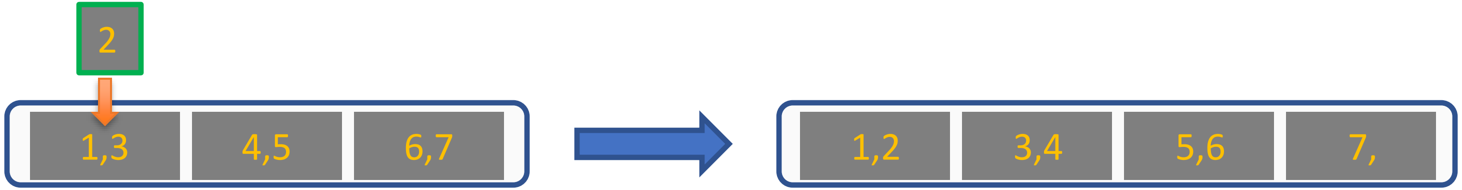
How many IO operations to search over *N sorted* records?

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

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

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 get faster insertions?

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

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

An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)

- We’ll cover both, but mainly consider secondary indexes

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

Using Indexes in SQL

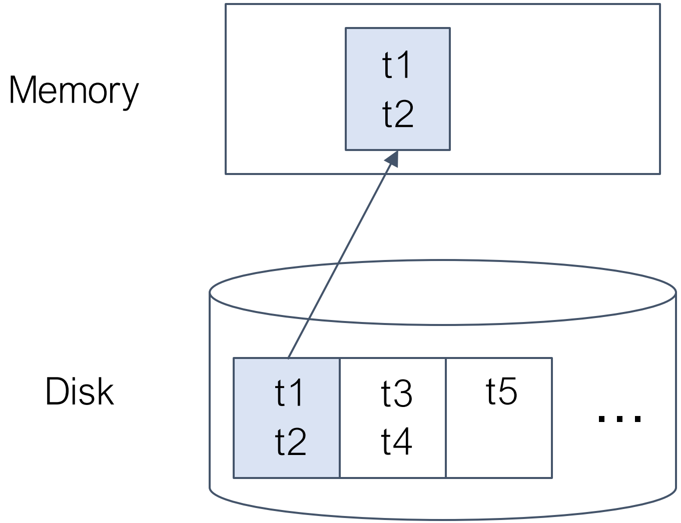
- An index is used to efficiently find tuples with certain values of attributes
- An index may speed up lookups and joins
- However, every built index makes insertions, deletions, and updates to relation more complex and time-consuming

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

```
DROP INDEX KeyIndex;
```

Recall: Simple cost model

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

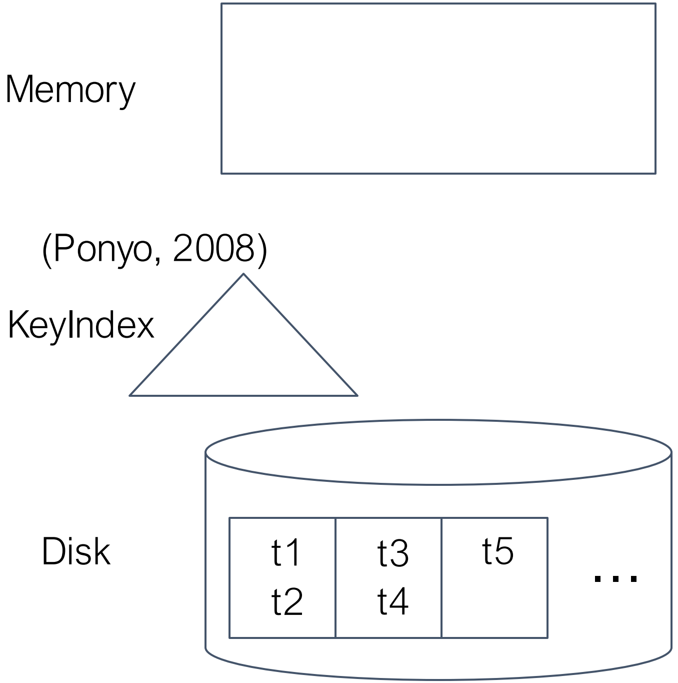


Index on a key

- 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;
```

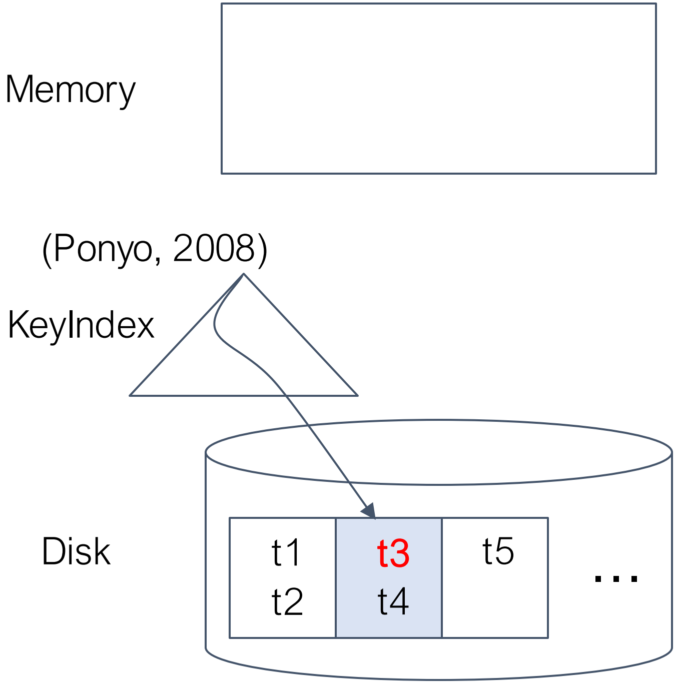


Index on a key

- 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;
```

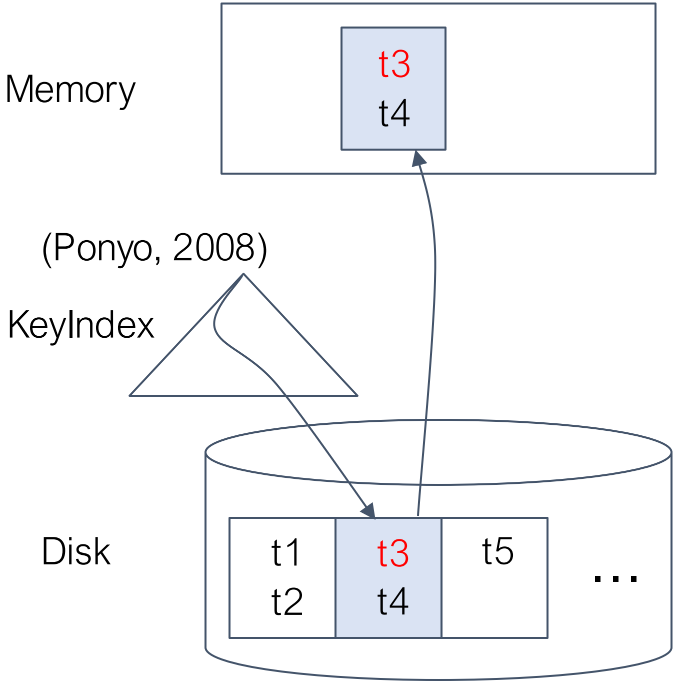


Index on a key

- 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

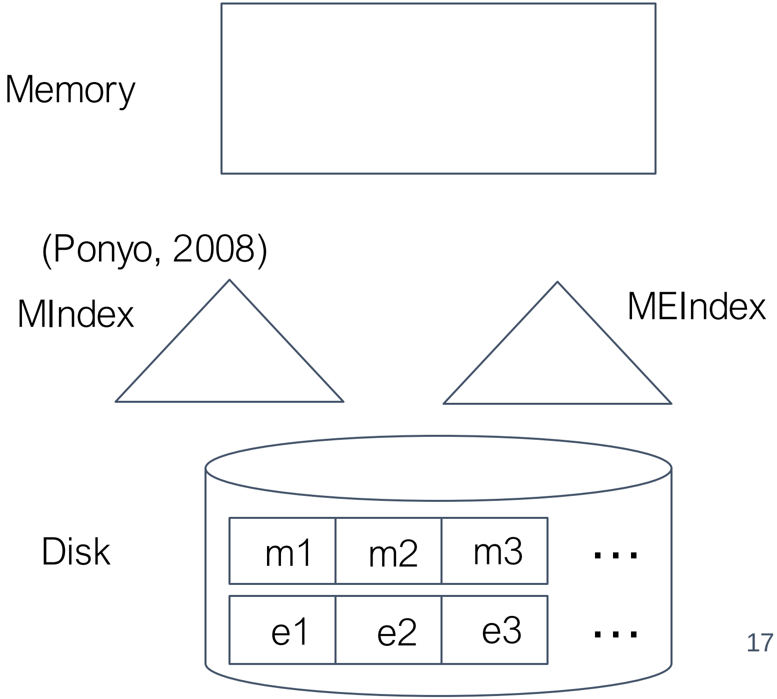
With the right indexes, the join below only requires 2 page reads for the tuples

- 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

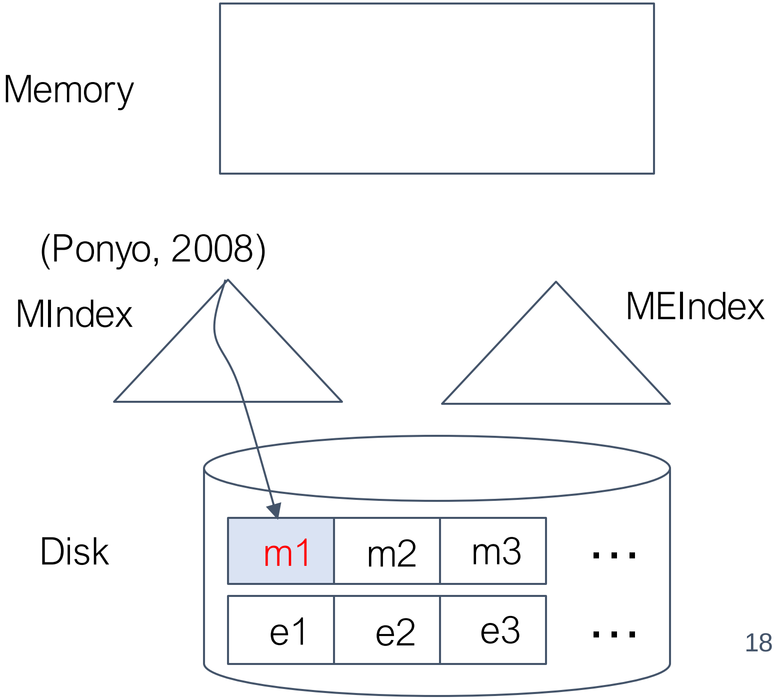
With the right indexes, the join below only requires 2 page reads for the tuples

- 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

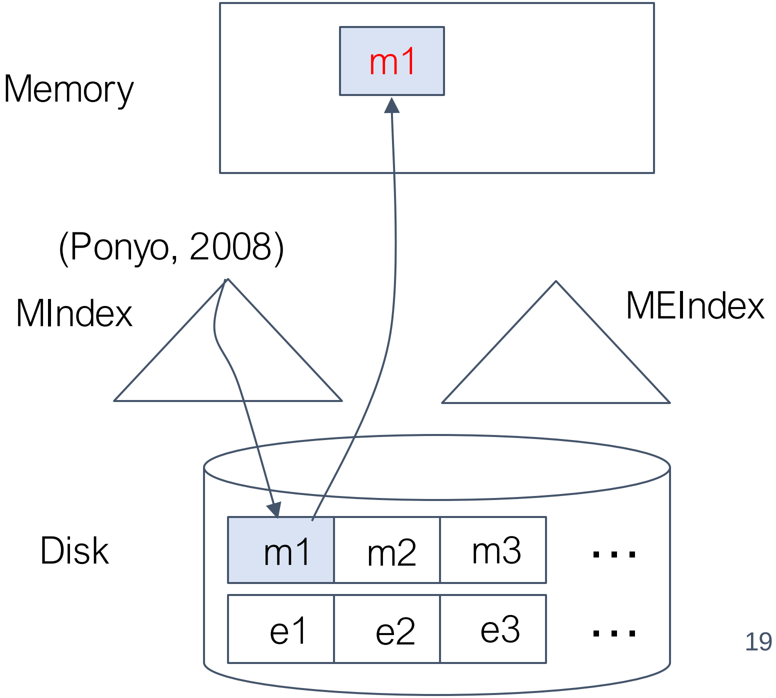
With the right indexes, the join below only requires 2 page reads for the tuples

- 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

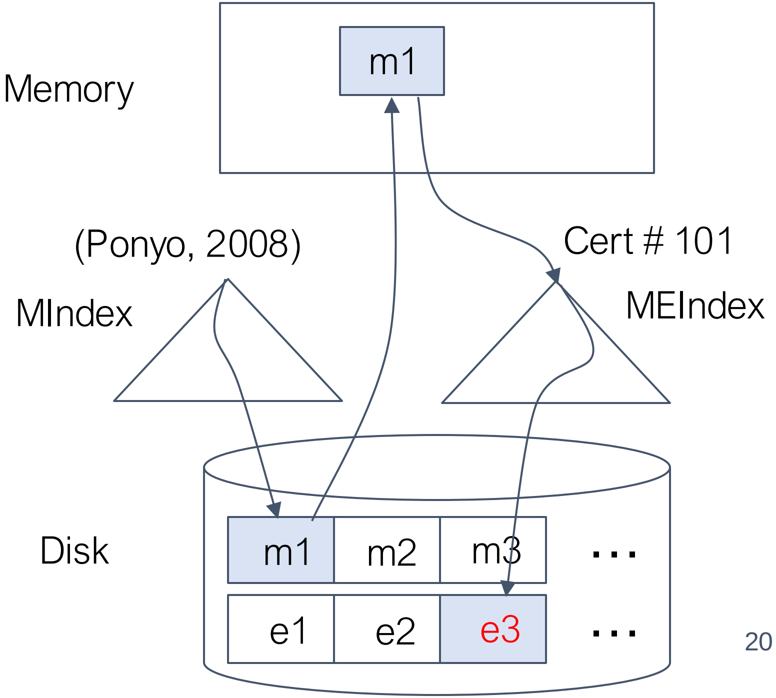
With the right indexes, the join below only requires 2 page reads for the tuples

- 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

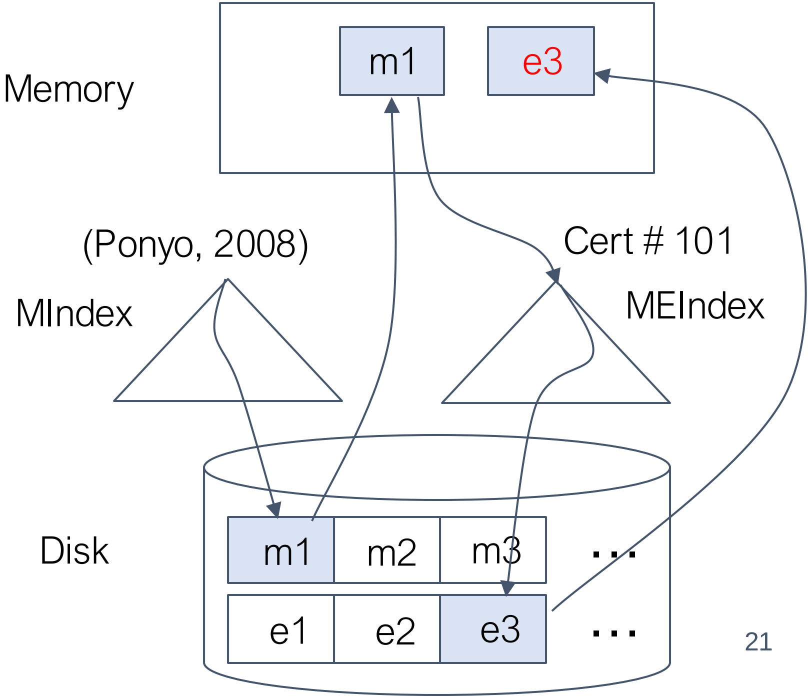
With the right indexes, the join below only requires 2 page reads for the tuples

- 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#);
```

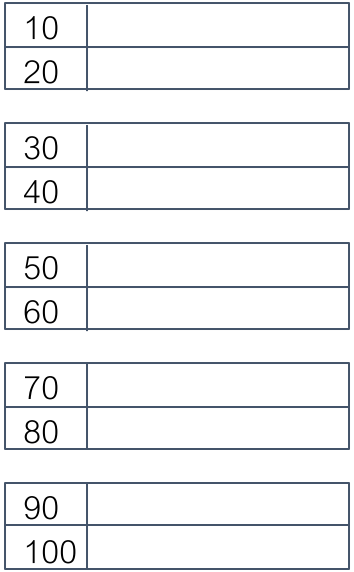
```
SELECT name
FROM Movies, MovieExec
WHERE title = ‘Ponyo’ AND year = 2008
  AND producerC# = cert#;
```



2. Index Structure Basics

Sequential file

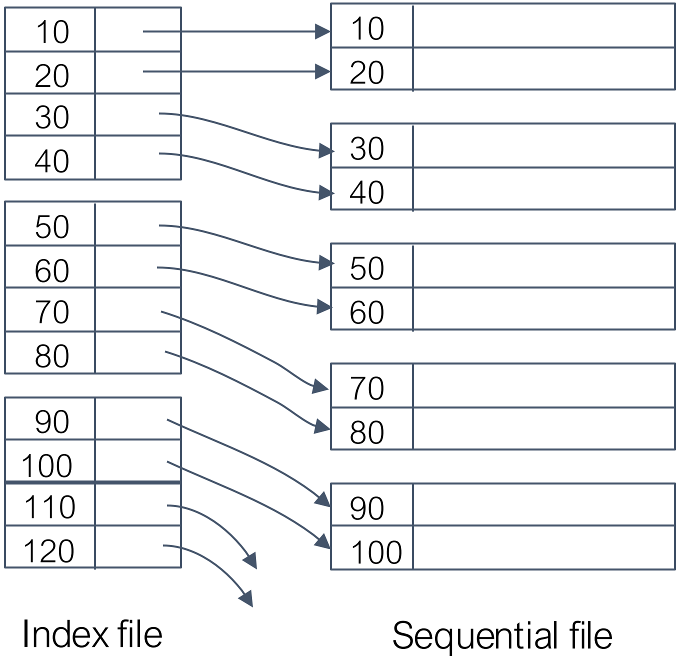
- A file containing tuples of a relation sorted by their primary key



Sequential file

Dense index

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



Dense index

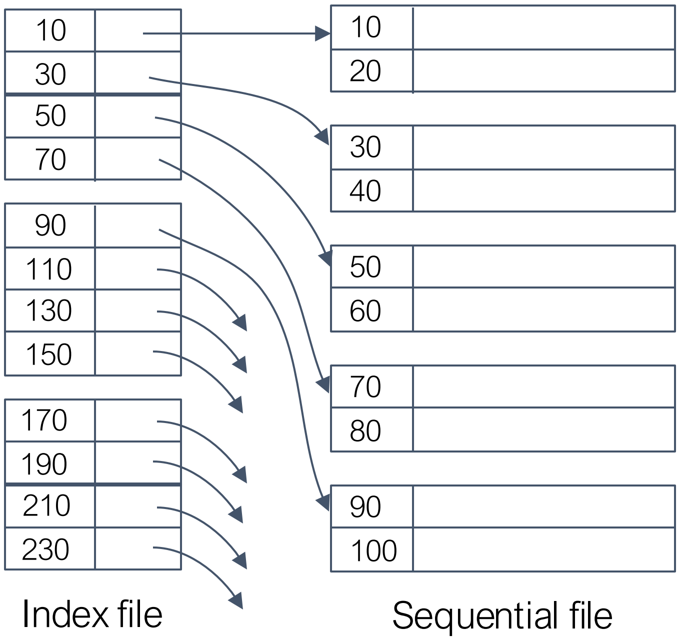
Given key K, search index blocks for K, then follow associated pointer

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



In-class Exercise

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

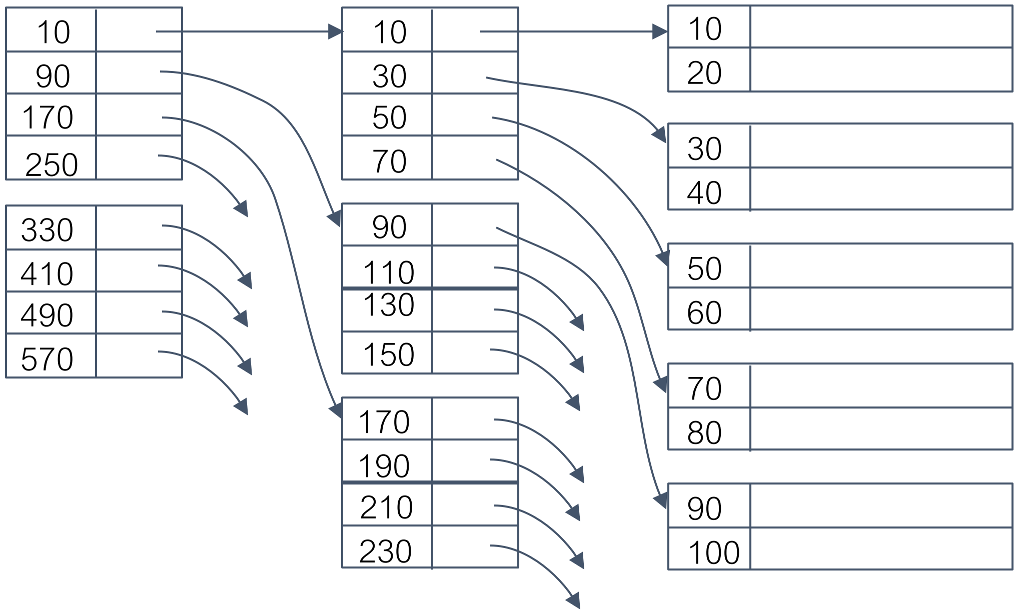
If there are n records in a data file, how many blocks are needed to hold

- 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)



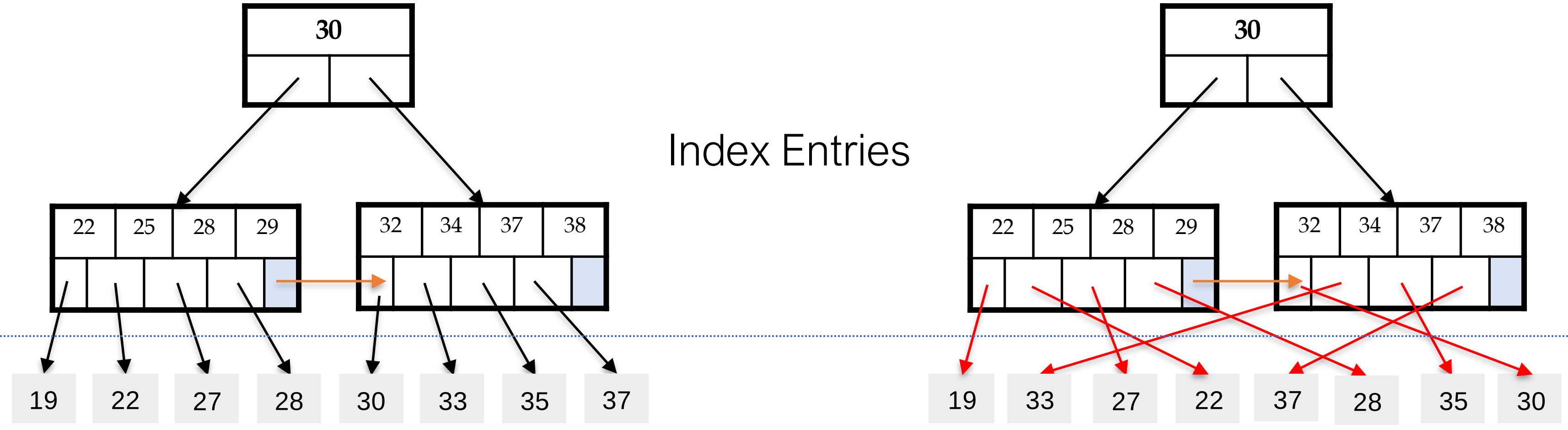
Q: Should the blocks of additional levels be dense or sparse?

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

Sometimes also referred to as primary vs secondary index



Index Entries

Data Records

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**

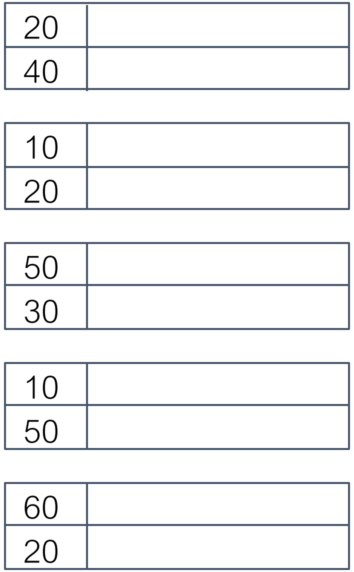
For point lookup, no difference between clustered / unclustered

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!**

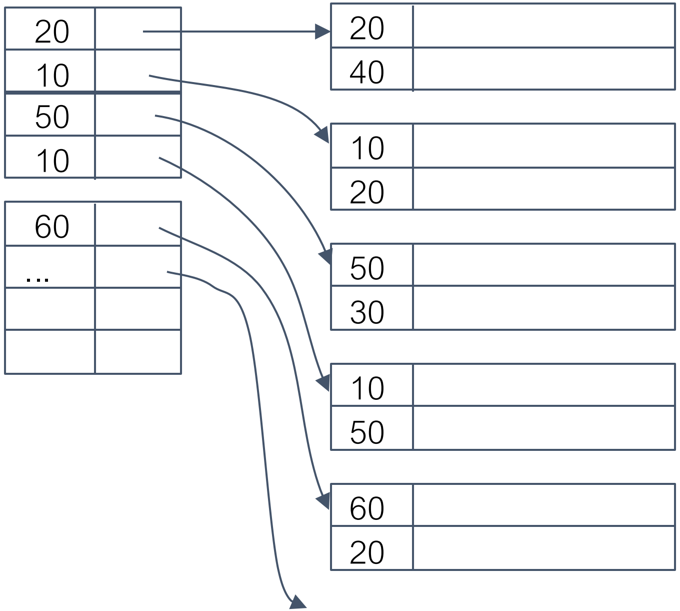
Non-clustered/Secondary index

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



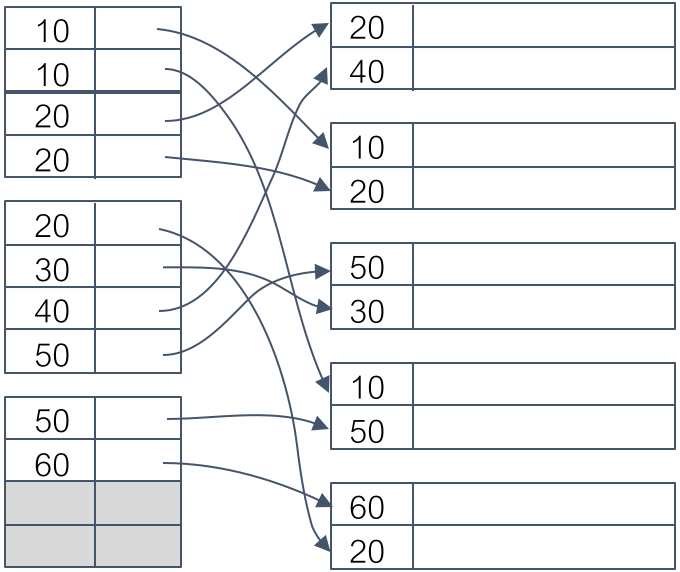
Non-clustered/Secondary index

Using a sparse index doesn't make sense



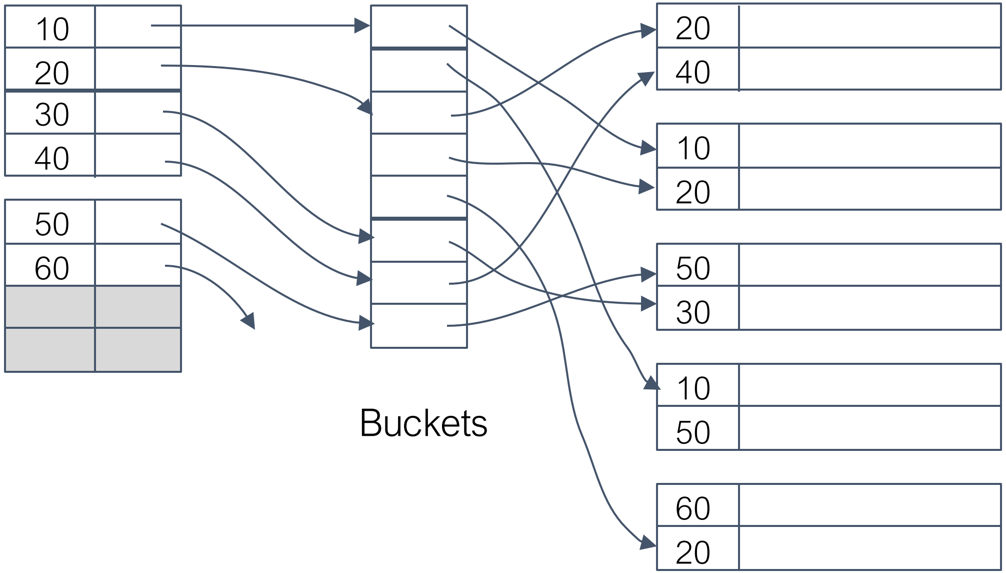
Non-clustered/Secondary index

As a result, secondary indexes are always dense



Non-clustered/Secondary index

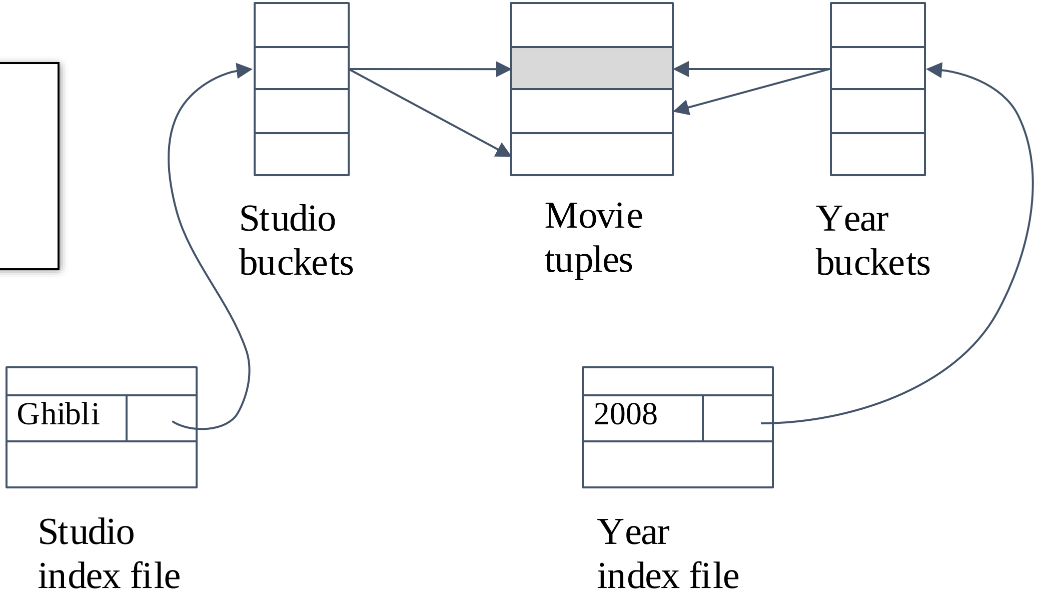
To remove redundant keys in index file, use level of indirection



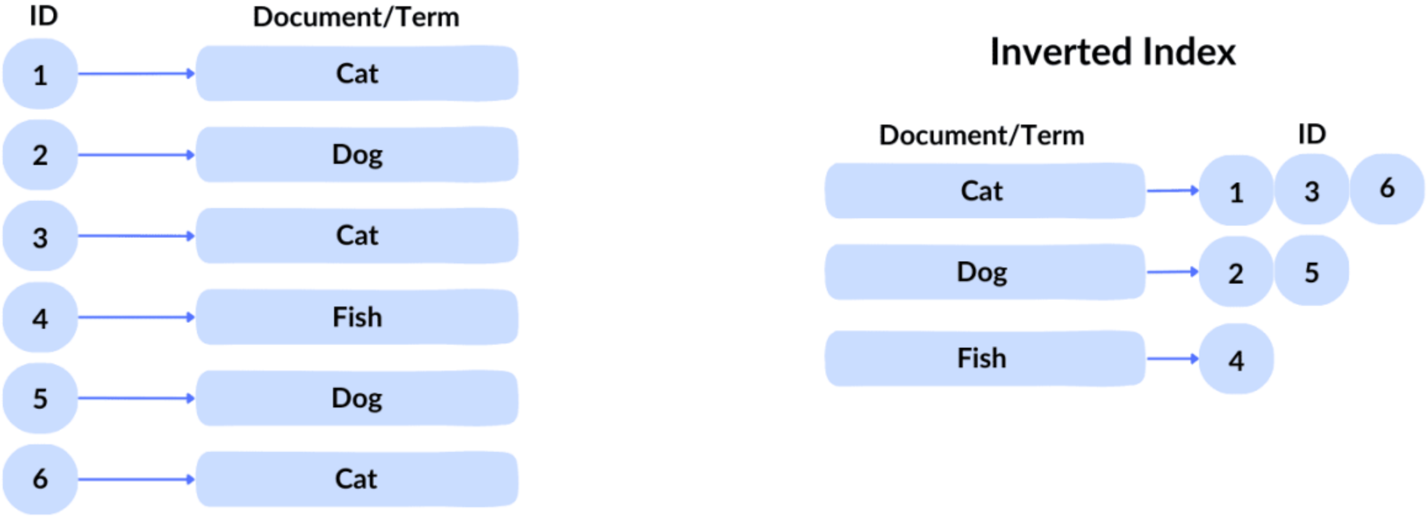
When is indirection and secondary index useful?

- 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: where the name came from



ID	Document/Term
1	Cat
2	Dog
3	Cat
4	Fish
5	Dog
6	Cat

Inverted Index

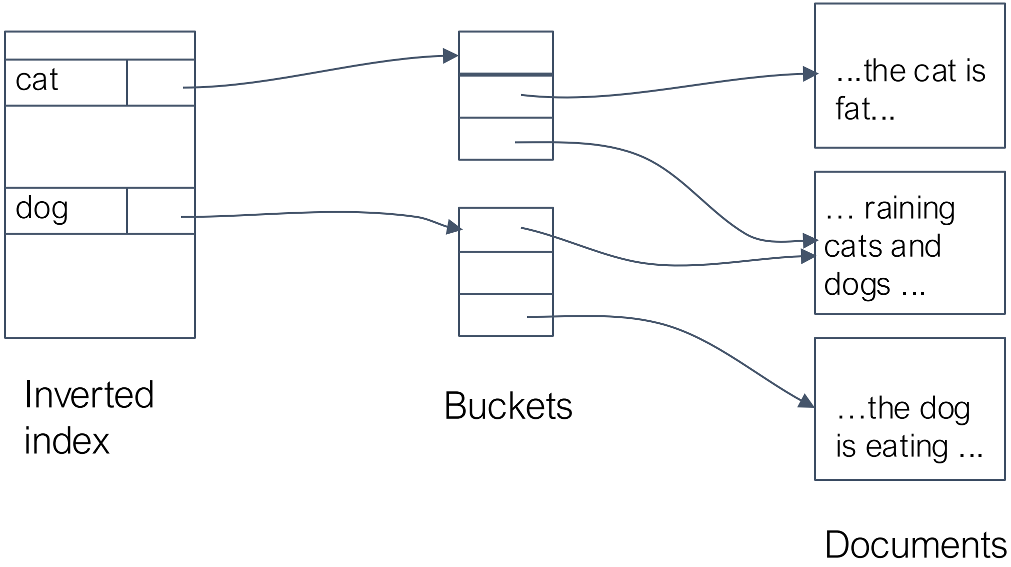
Document/Term	ID		
Cat	1	3	6
Dog	2	5	
Fish	4		

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

Inverted index

Essentially a secondary index, used in text information retrieval

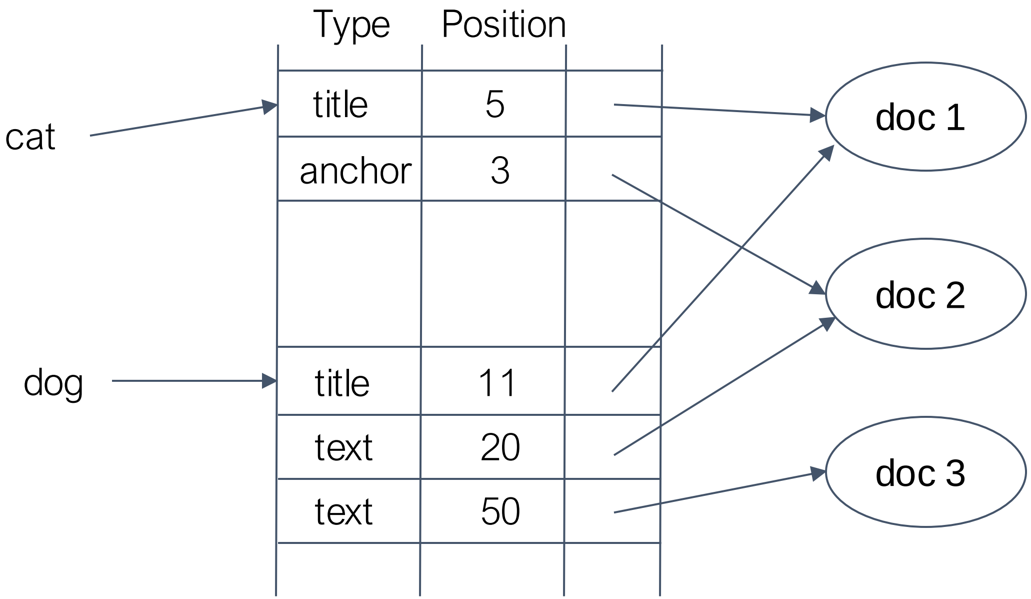
- e.g., Search for documents containing “cat” or “dog” (or both)



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



Type	Position	
title	5	
anchor	3	
title	11	
text	20	
text	50	