CSC 261/461 – Database Systems
Lecture 17

Spring 2018

Did you study?

- Chapter 16 and 17

BASICS

Page

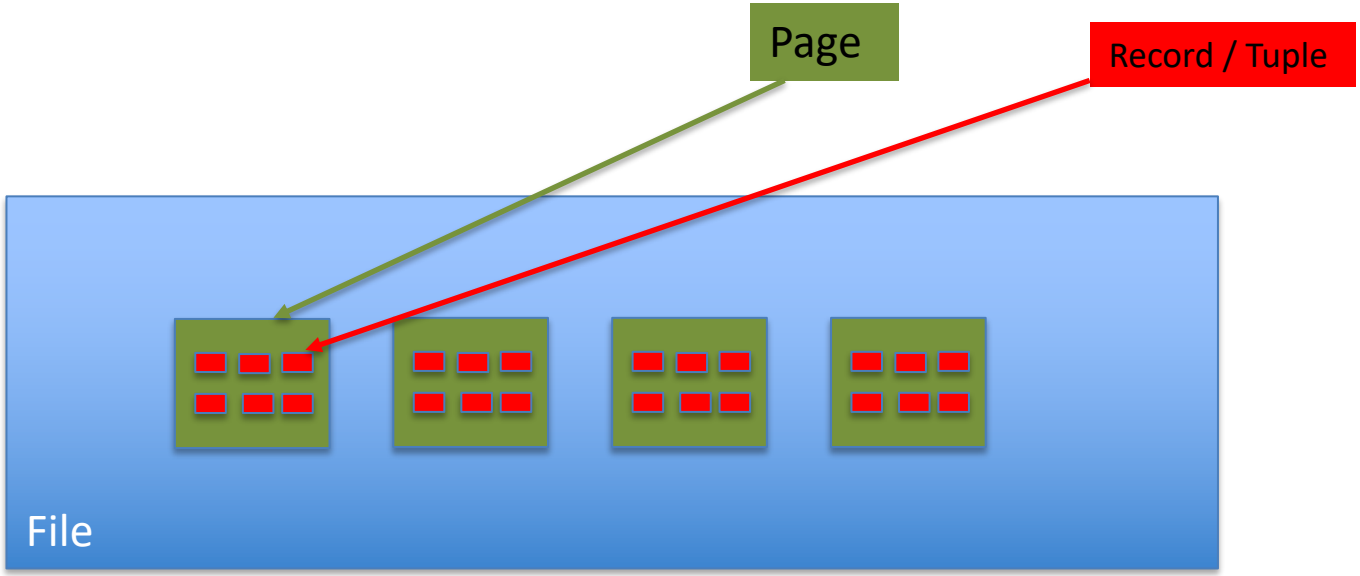
- The unit of information read from or written to the disk

How do we measure efficiency?

- The cost of page I/O
  - i.e., Input from disk to main memory and output from memory to disk

How is a relation (table) stored?

- As a file of records



Blocking Factor

- How many records a block can hold

- $bfr = \lfloor \frac{B}{R} \rfloor$

- B = Block size
- R = fixed length record

Records and Record Types

Data in Database

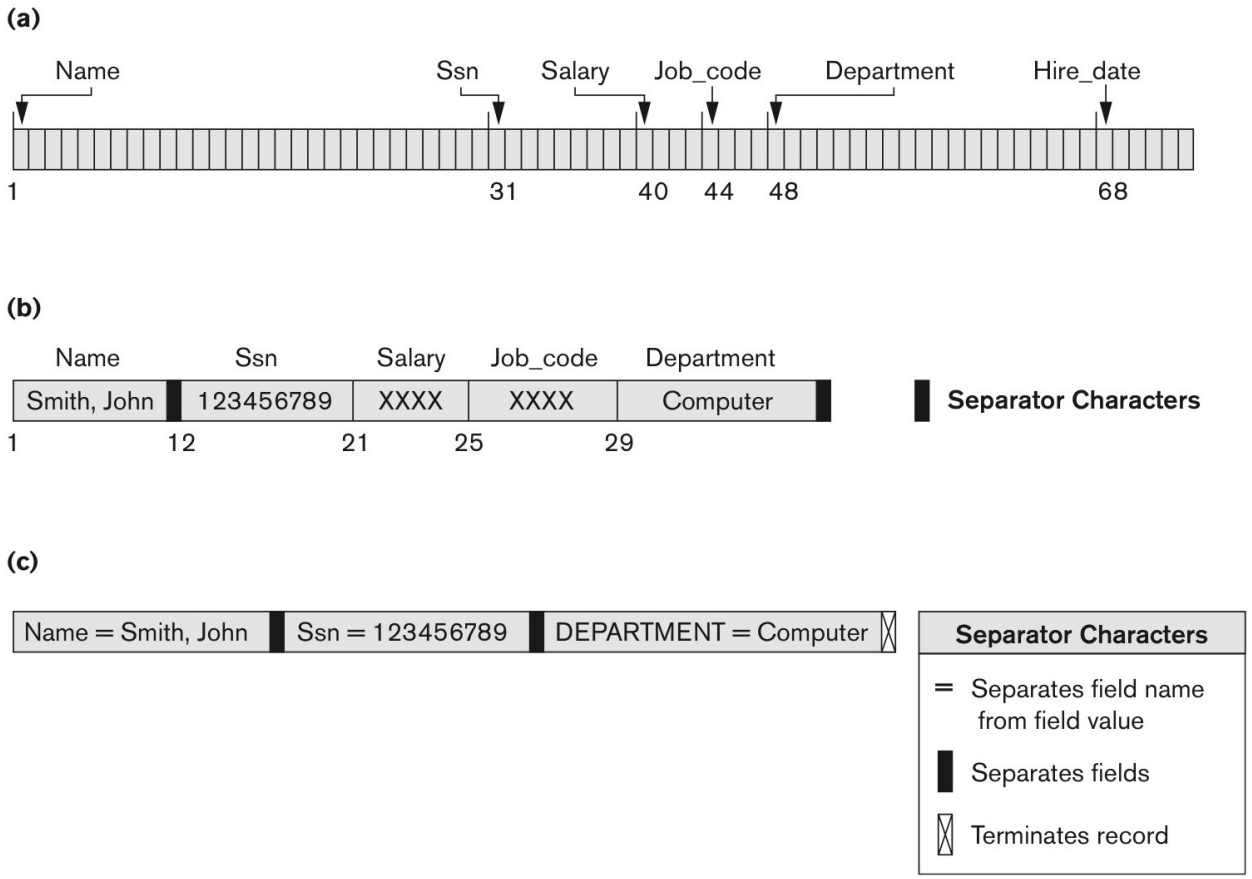
=

A set of **records** organized into a set of **files**

- Each record consists of a collection of related data values or items
- Each value corresponds to a particular attribute
  - Takes one or more bytes

Fixed Length Records vs Variable Length Records

- Fig 16.5 (from textbook)



Record Blocking

- Block Size = B bytes
- Fixed Record Size = R bytes

- With $B > R$:
  - Records per block = $bfr = \lfloor \frac{B}{R} \rfloor$
  - Unused space per block = $B - bfr * R$ = B mod R

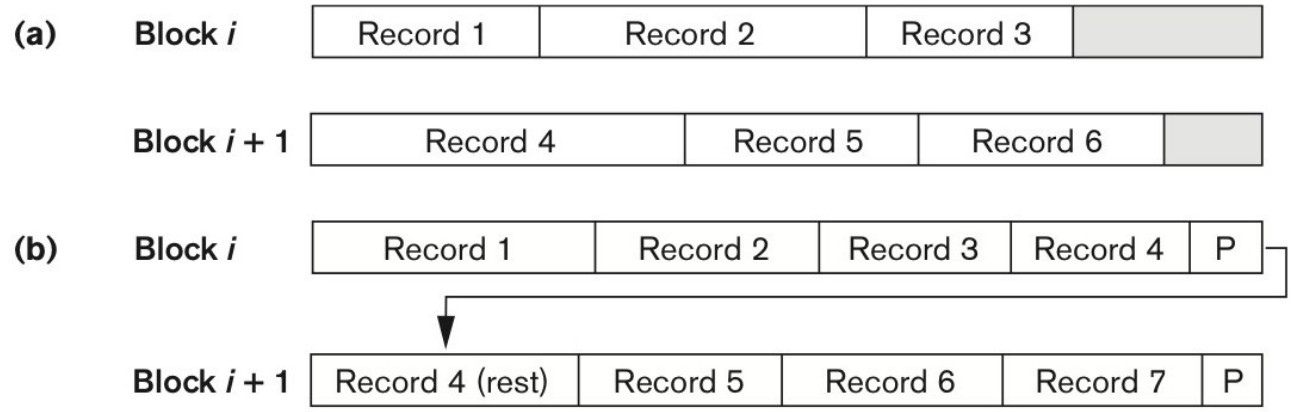
- Number of Blocks required = $\lceil (R/bfr) \rceil$

- Two scenarios:
  - Spanned: records can span more than one block
  - Unspanned: records can't span more than one block

Unspanned vs Spanned

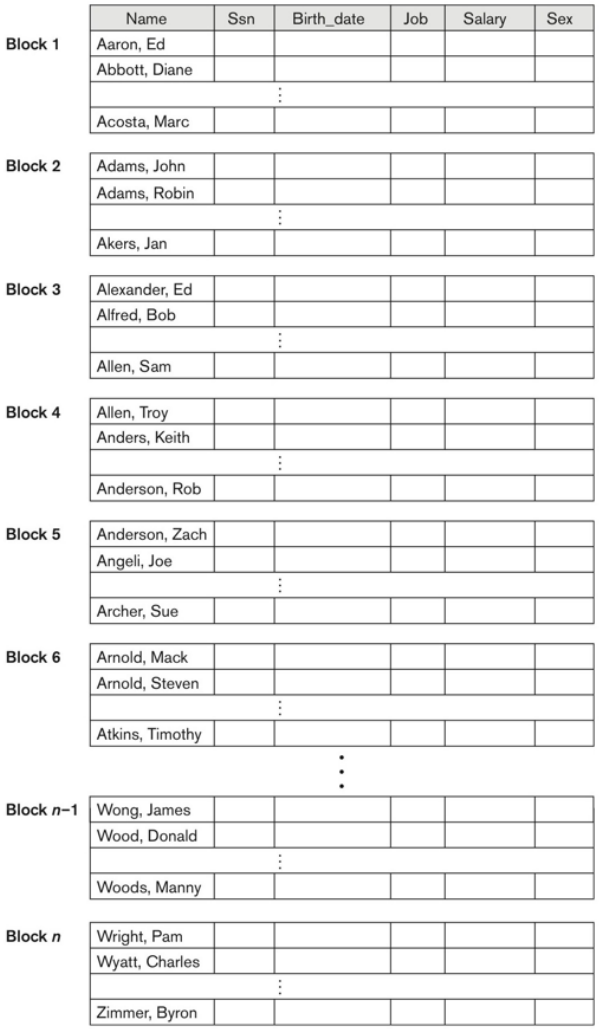
Unspanned

Spanned



Sorted Files

- Fig 16.7



Sorted Files (zoomed)

- Fig 16.7

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					

Heap Files vs Sorted Files

- Insertion
  - (Heap vs Sorted)
  - For Sorted files: Overflow
- Deletion
  - Deletion Marker
- Modifying
  - For Sorted files: May consist of Deletion and Insertion
- Searching
  - How many block access?
  - (by record number) What if the records are numbered and are of fixed size?
  - Searching by range (Heap vs Sorted)

Another variety

- Another type of files beyond:
  - Heap Files and Sorted Files
  - **Hashed Files**

Primary File Organization

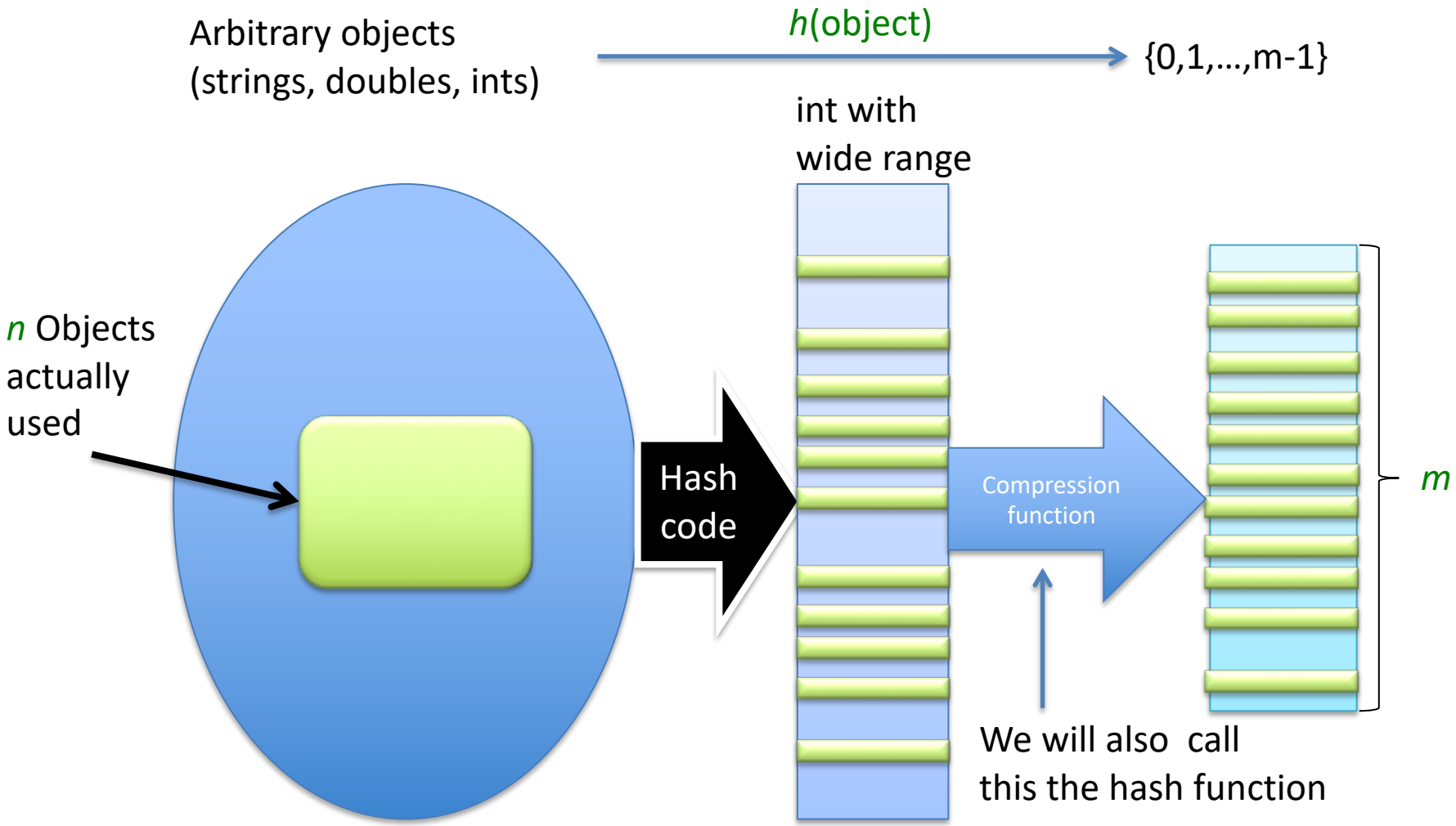
- How the file records are physically placed?
- **Heap File**
  - No particular order
- **Sorted File**
  - Ordered by the value of a particular field
- **Hashed File**
  - A hash function applied to a particular field to determine a records placement.

HASHING TECHNIQUES

- Hashing first proposed by Arnold Dumey (1956)
- Hash codes
- Chaining
- Open addressing

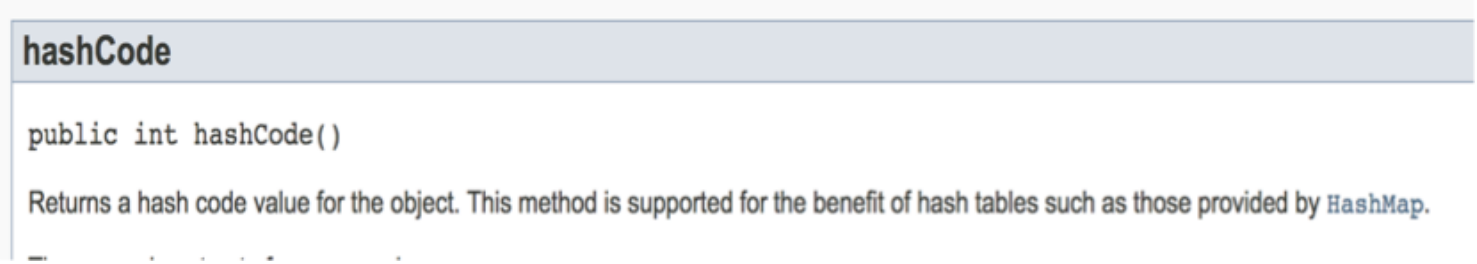
HASHING – GENERAL IDEAS

Top level view



Example (Once again!)

Java HashCode



hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The hash code for a String object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

True / False

- Unequal objects must have different hash codes
- Objects with the same hash code must be equal
- Both Wrong
  - We would like this but it's impossible to achieve
  - Still possible for a particular set of values but impossible if input values are unknown prior to applying the hashcode.

Good Hash Function

- If $key_1 \neq key_2$, then it's extremely unlikely that $h(key_1) = h(key_2)$
  - Collision problem!

- Pigeonhole principle
  - K+1 pigeons, K holes → at least one hole with ≥ 2 pigeons

Compression: Division method

$$h(s) = s \mod m$$

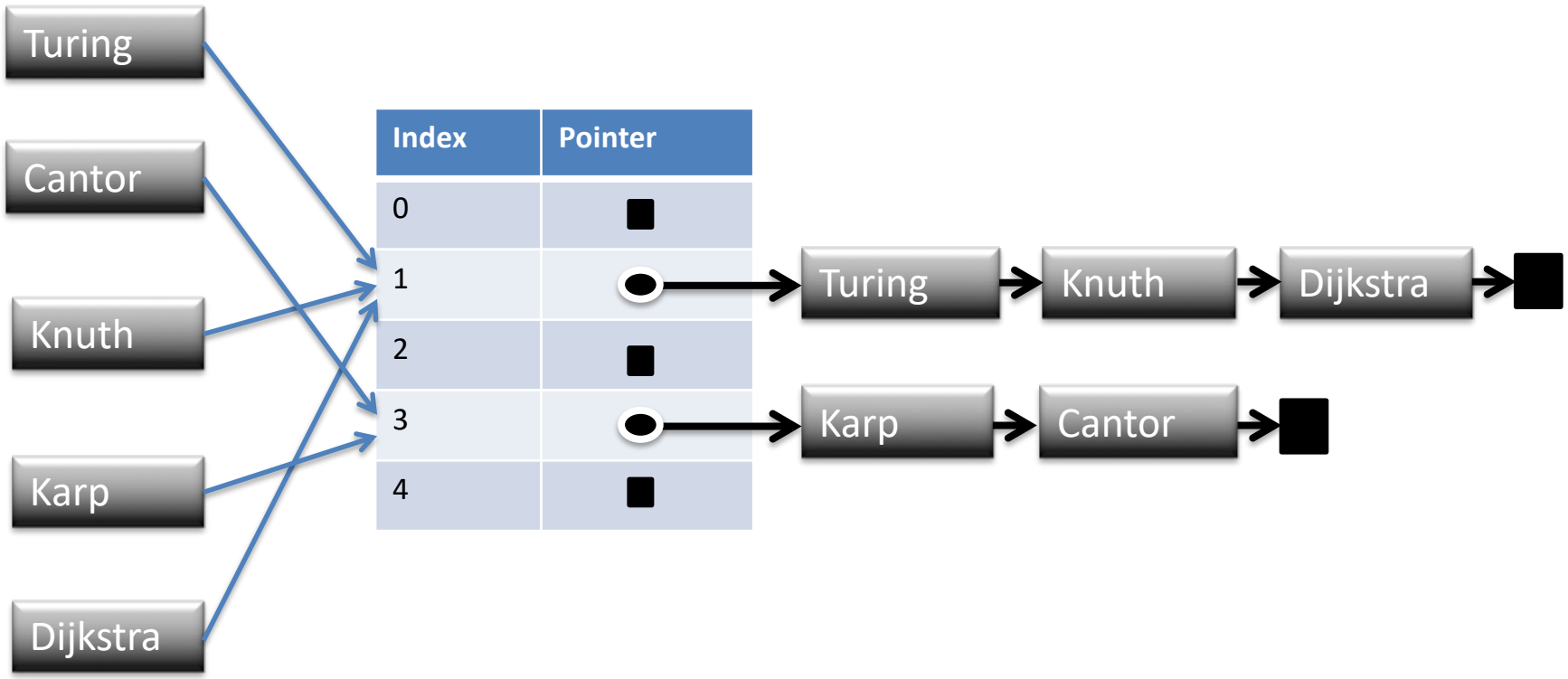
- How does this function perform for different *m*?

Separate chaining

Open addressing

COLLISION RESOLUTION

Separate Chaining



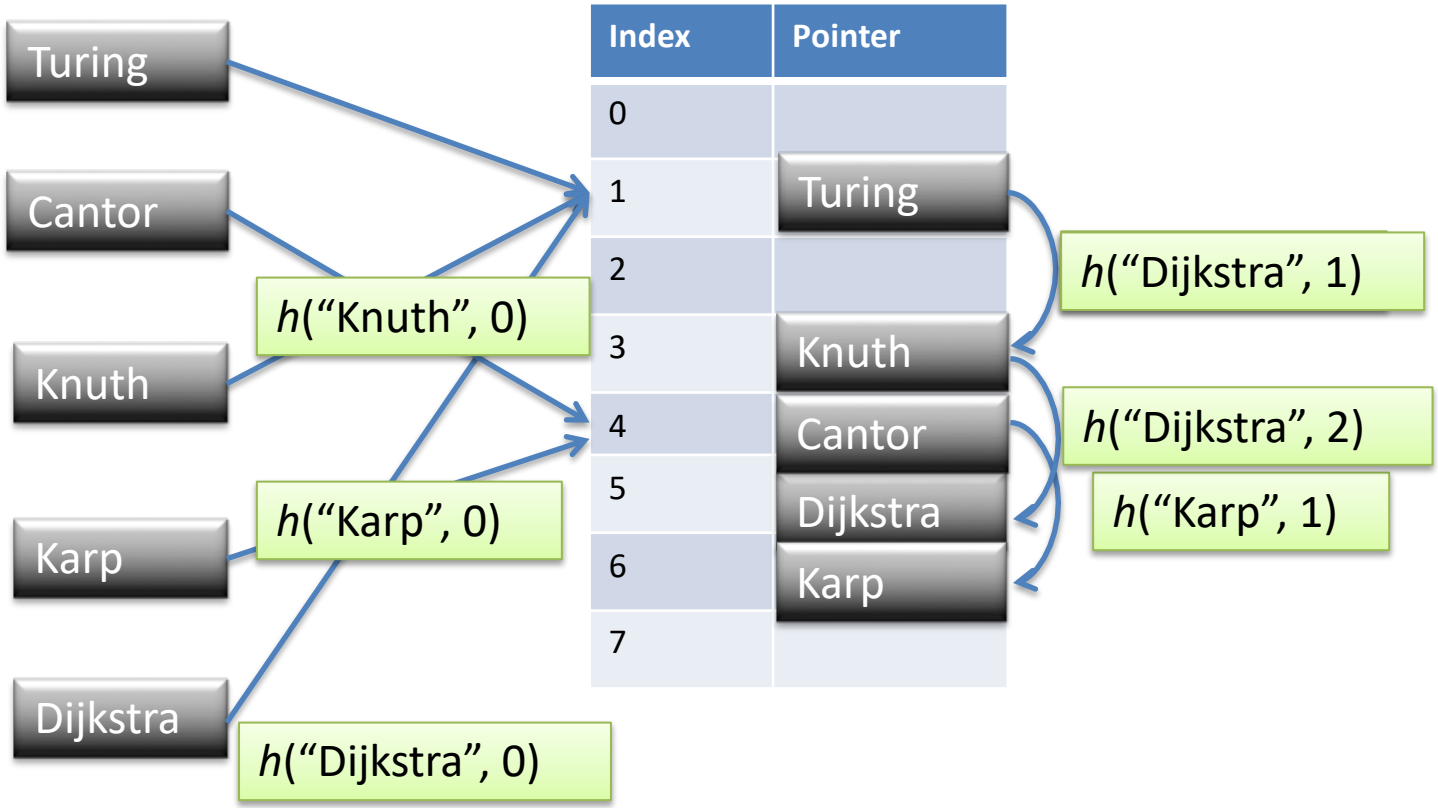
Open Addressing

- Store all entries in the hash table itself, no pointer to the “outside”

- Advantage
  - Less space waste
  - Perhaps good cache usage

- Disadvantage
  - More complex collision resolution
  - Slower operations

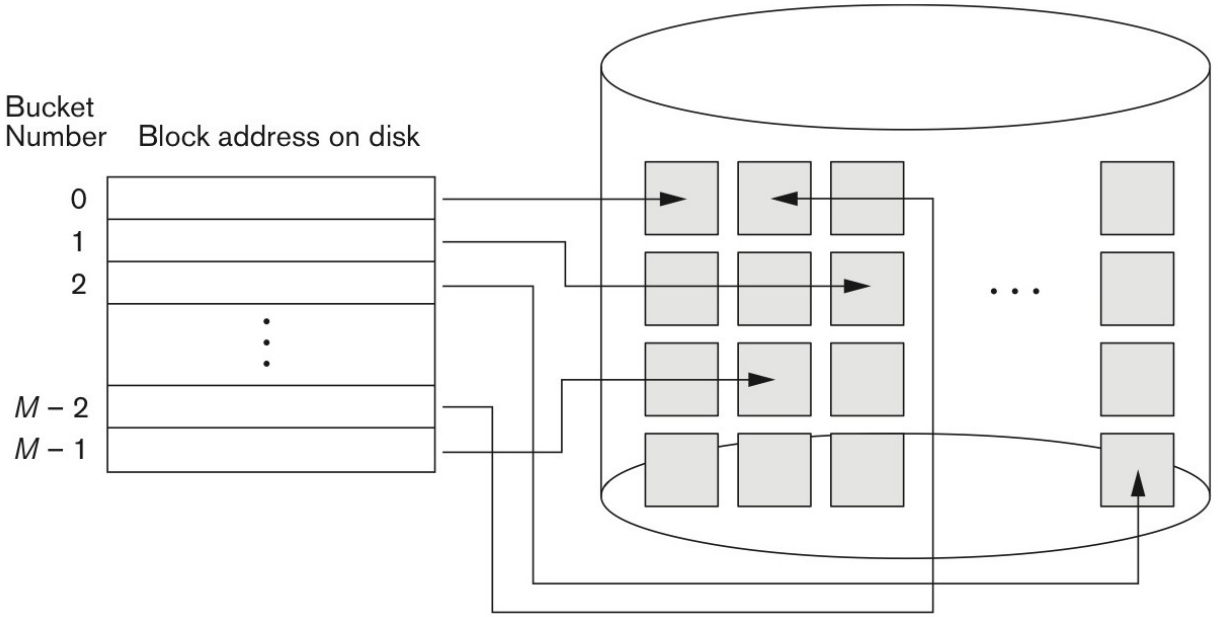
Open Addressing



External Hashing

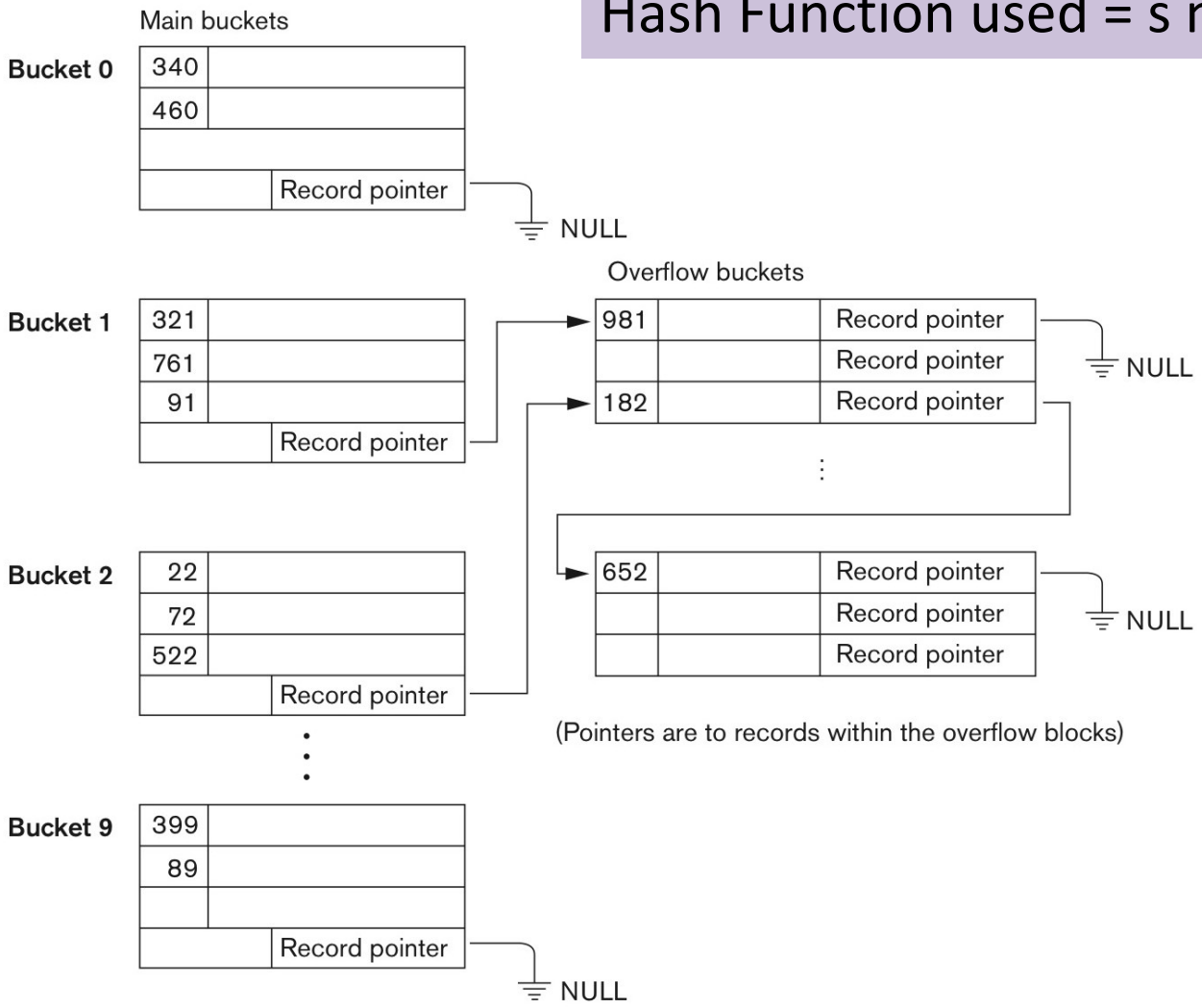
- Hashing for disk files
- Target address space is made of buckets
- Hashing function maps a key into relative bucket number
- Convert the bucket number into corresponding disk block address

Bucket Number to Disk Block address



Bucket Number to Disk Block address

Hash Function used = s mod 10



REVIEW

What Did We Learn

- Disk Storage
  - Hardware Description of Disk Devices
    - Sections to study: 16.2.1 and 16.2.2
- Buffering of Blocks
    - Buffer Management
    - Buffer Replacement Strategies
    - Sections to study: 16.3
- Placing File Records on Disk
    - Records and Record Types
    - Fixed-Length, Variable-Length, Spanned, Unspanned
    - Sections to study: 16.4.1 to 16.4.4

What did we learn

- Operations on Files
  - Insert, Modify, and Delete
  - And others.
  - Sections to study: 16.5
- Heap Files vs Sorted Files
  - Sections to study: 16.6 and 16.7
- Hash Files
  - Internal Hashing and External Hashing
  - Sections to study: 16.8.1 and 16.8.2

INDEXING

Types of Indexing

- Primary Indexes
- Clustering Indexes
- Secondary Indexes
- Multilevel Indexes
  - Dynamic Multilevel Indexes
- Hash Indexes

What you will learn about in this section

1. Indexes: Motivation
2. Indexes: Basics

Index Motivation (1)

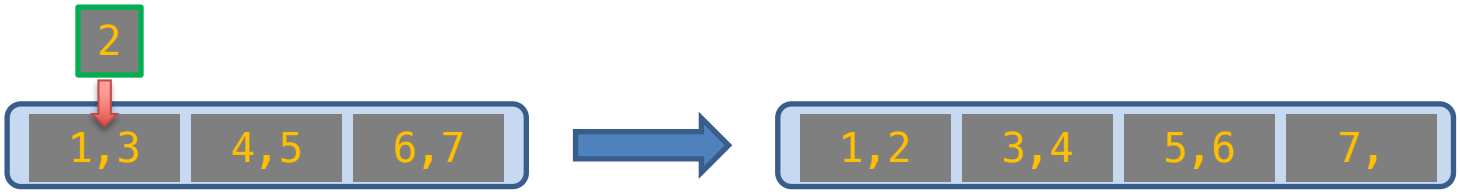
`Person(name, age)`

- 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 (2)

- 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 (3)

- 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?

Indexes

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!
  - Sometimes use B+ Trees, sometimes hash indexes

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
- An index can store:
  - Full rows it points to (primary index) or
  - Pointers to those rows (secondary index)

Operations on an Index

- Search: Quickly find all records which meet some condition *on the search key attributes*
  - More sophisticated variants as well. Why?

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

Conceptual Example

What if we want to return all books published after 1867? The above table might be very expensive to search over row-by-row…

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

```
SELECT *
FROM Russian_Novels
WHERE Published > 1867
```

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

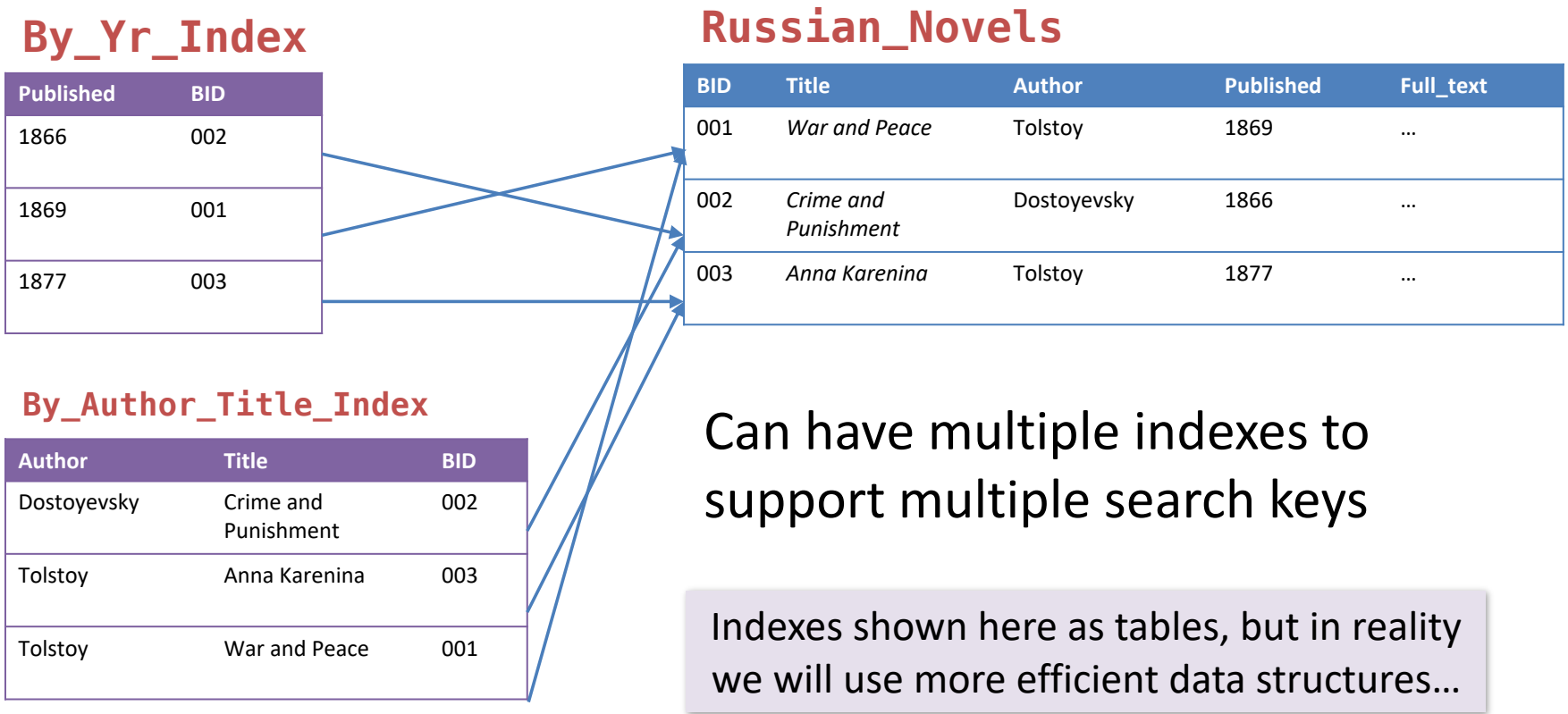
Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Conceptual Example



Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

Covering Indexes

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

We say that an index is **covering** *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

The "needed" attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```

TYPES OF INDEXES

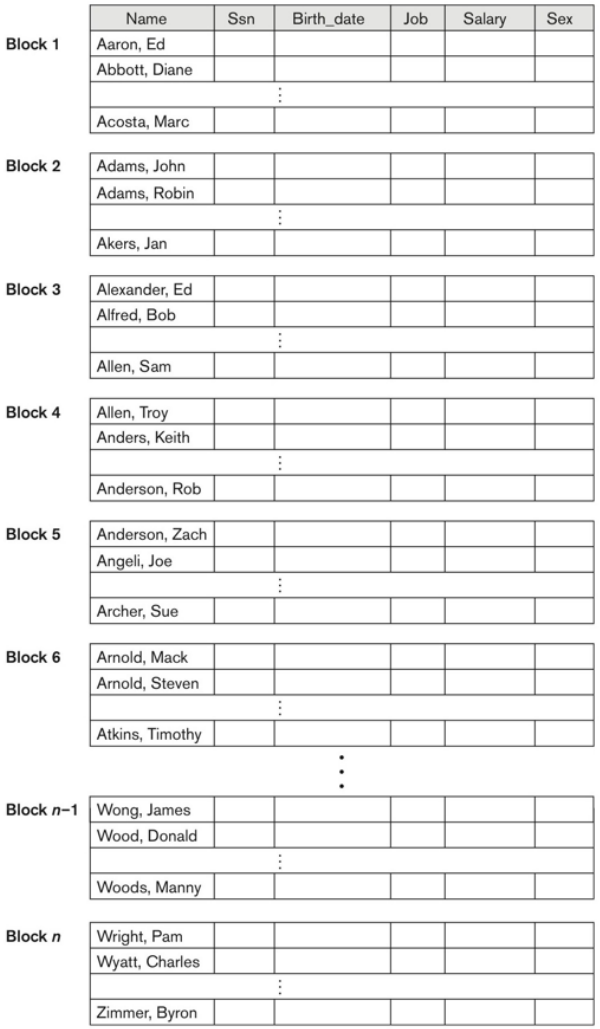
Types of Indexing

- Primary Indexes
- Clustering Indexes
- Secondary Indexes
- Multilevel Indexes
  - Dynamic Multilevel Indexes
- Hash Indexes
- Easy introduction: https://www.tutorialspoint.com/dbms/dbms_indexing.htm

Sorted Files

- Fig 16.7

Recap: No Indexing



	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
Block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
	⋮					
Block *n*−1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
Block *n*	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

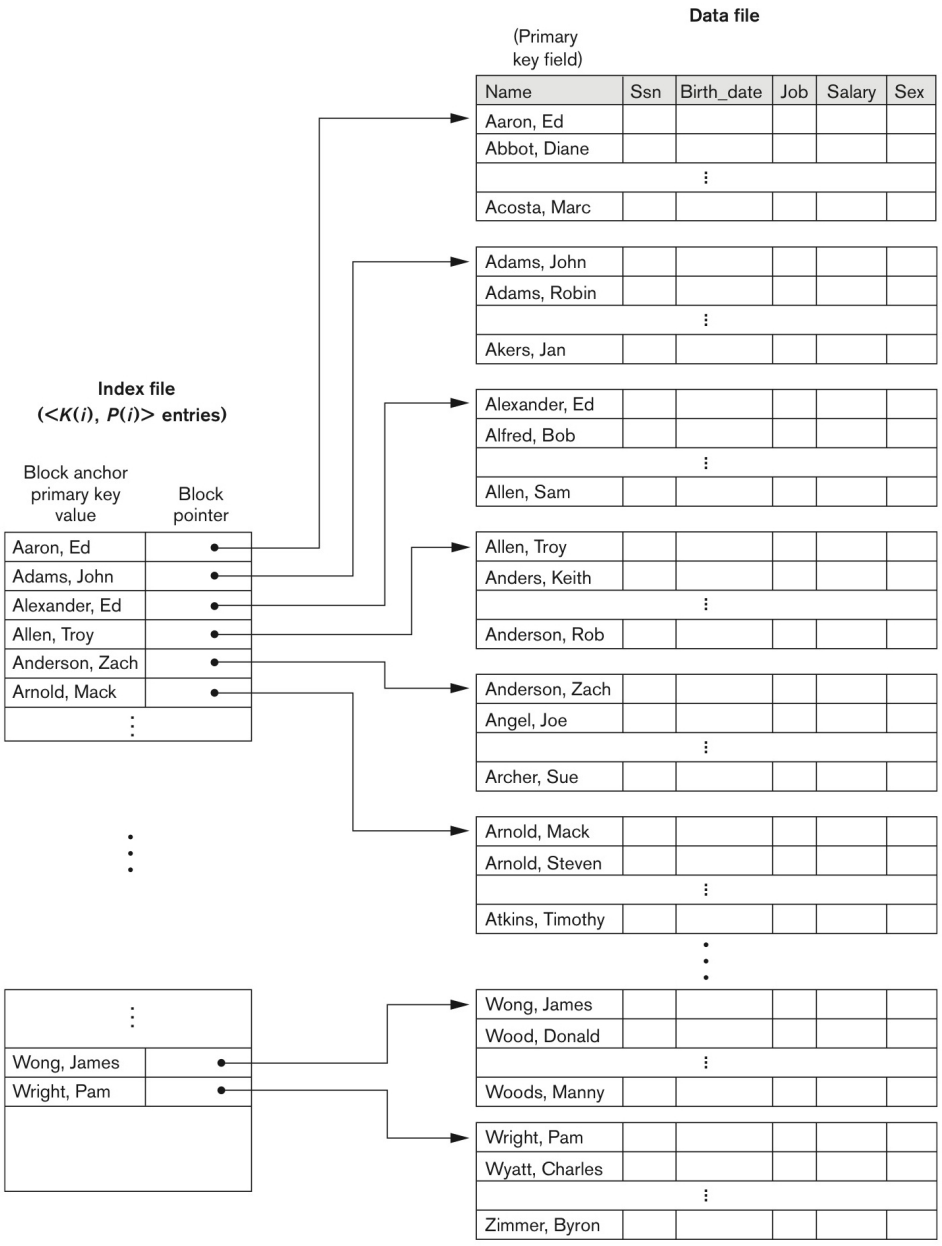
Sorted Files (zoomed)

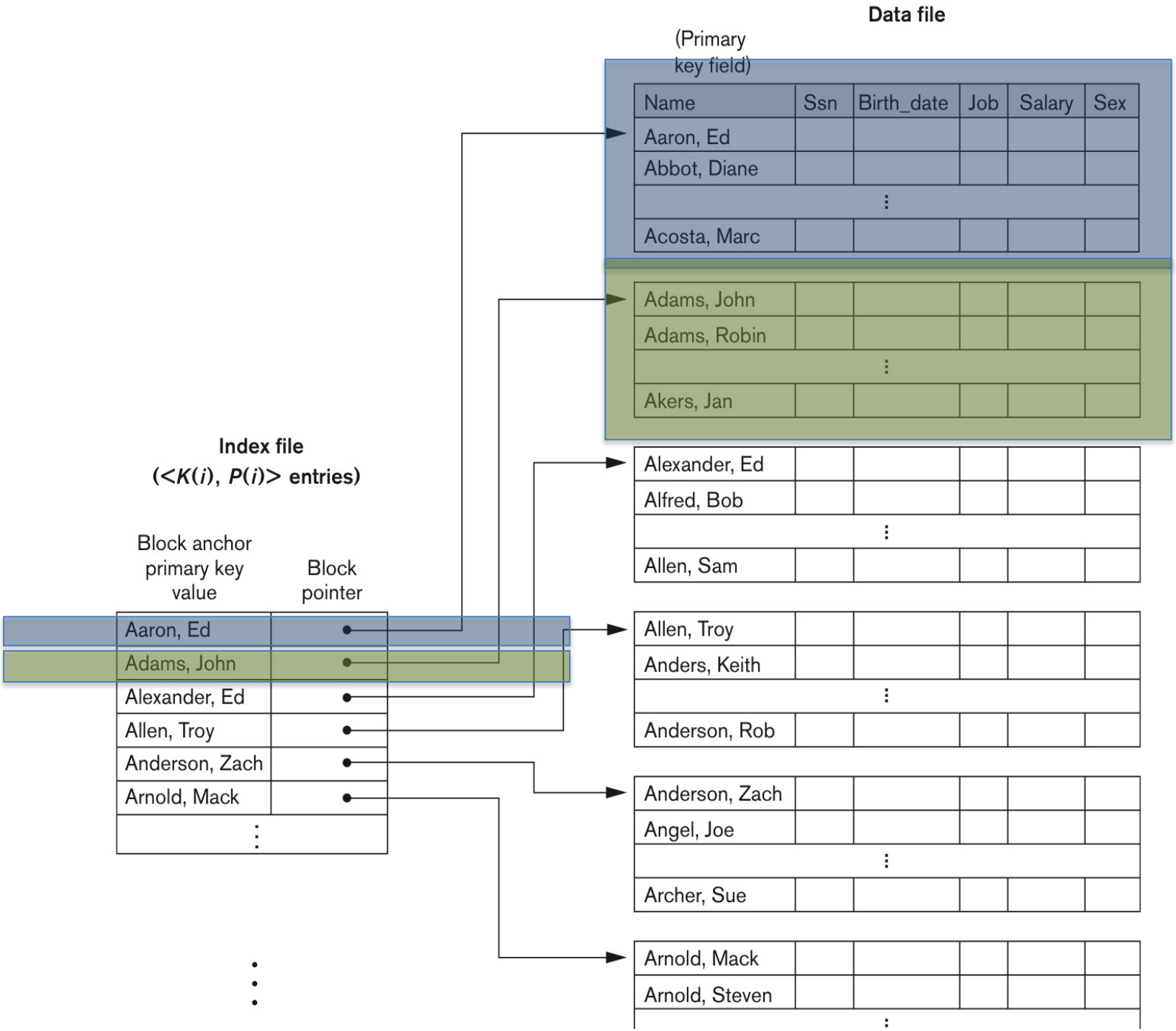
- Fig 16.7

Recap: No Indexing

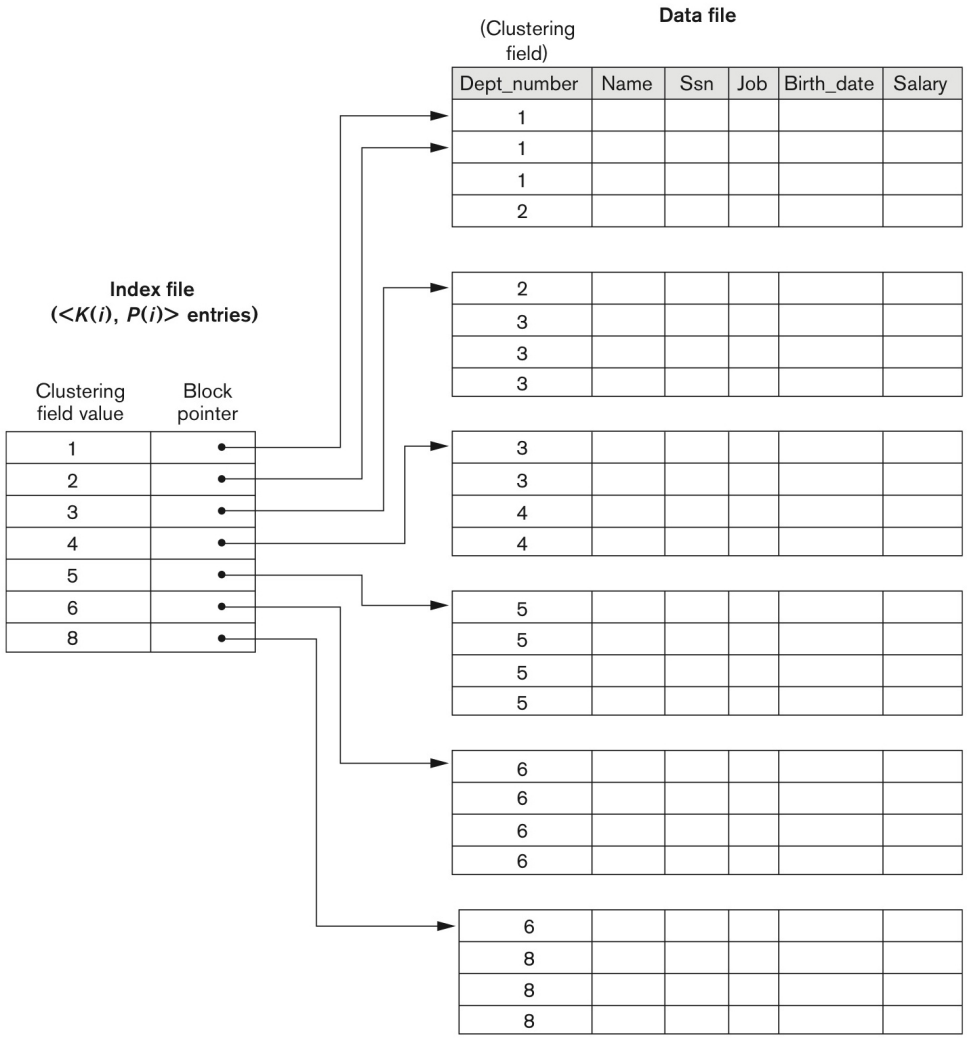
	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					

Primary Indexes: Index for Sorted (Ordered) Files




Data file
(Primary key field)
Name
Ssn
Birth_date
Job
Salary
Sex
Aaron, Ed
Abbot, Diane
Acosta, Marc
Adams, John
Adams, Robin
Akers, Jan
Index file
(<K(i), P(i)> entries)
Block anchor primary key value
Block pointer
Aaron, Ed
Adams, John
Alexander, Ed
Allen, Troy
Anderson, Zach
Arnold, Mack
Alexander, Ed
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angel, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven

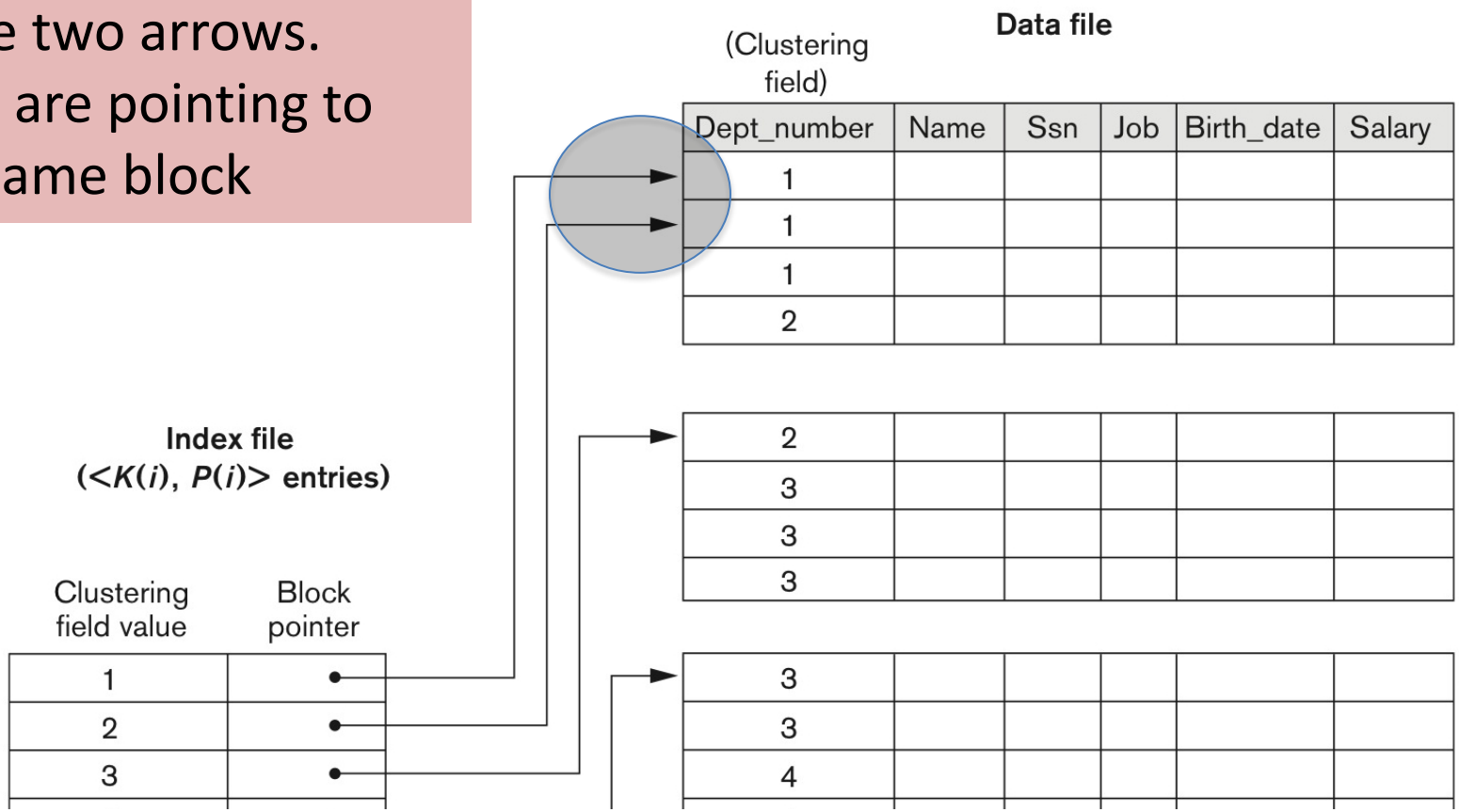
Clustering Indexes (Index for Sorted (on non-key) Files)



Clustering Indexes (Index for Sorted (on non-key) Files)

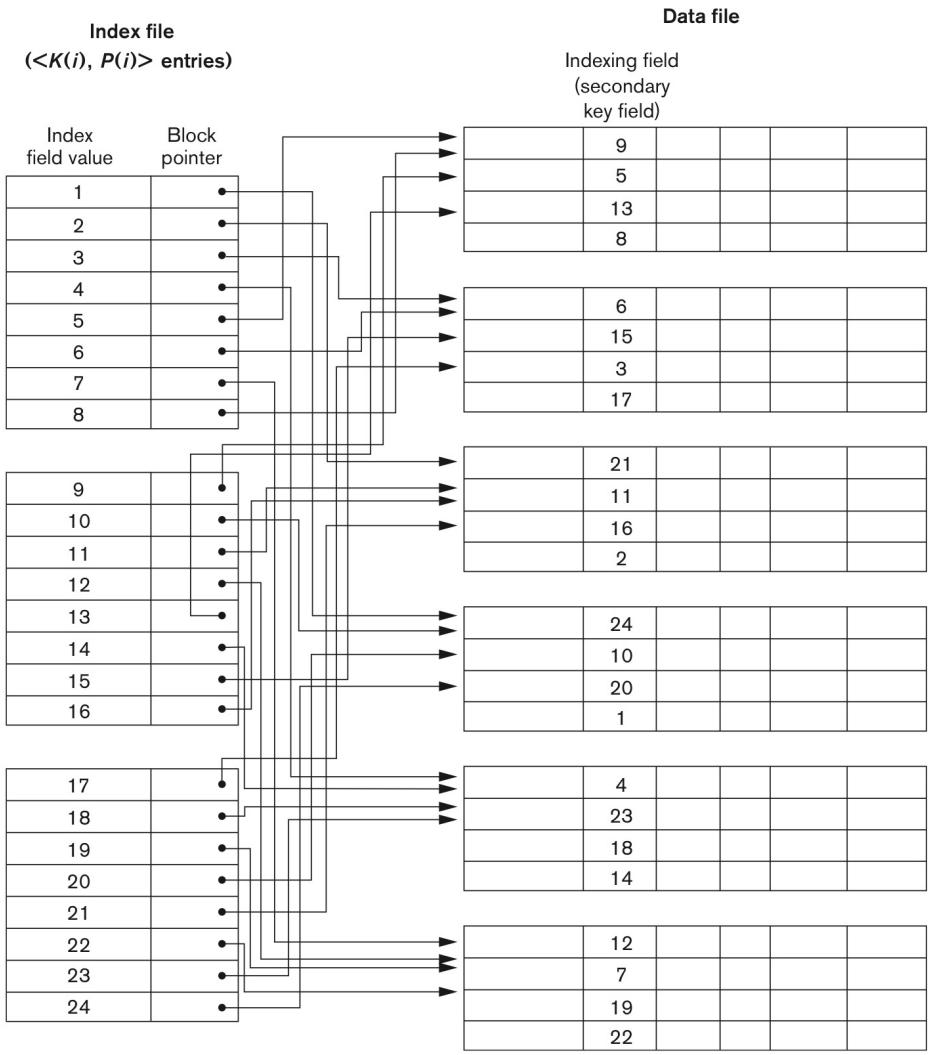
Points to the first block that contains the clustering field

Don't get confused by these two arrows. They are pointing to the same block



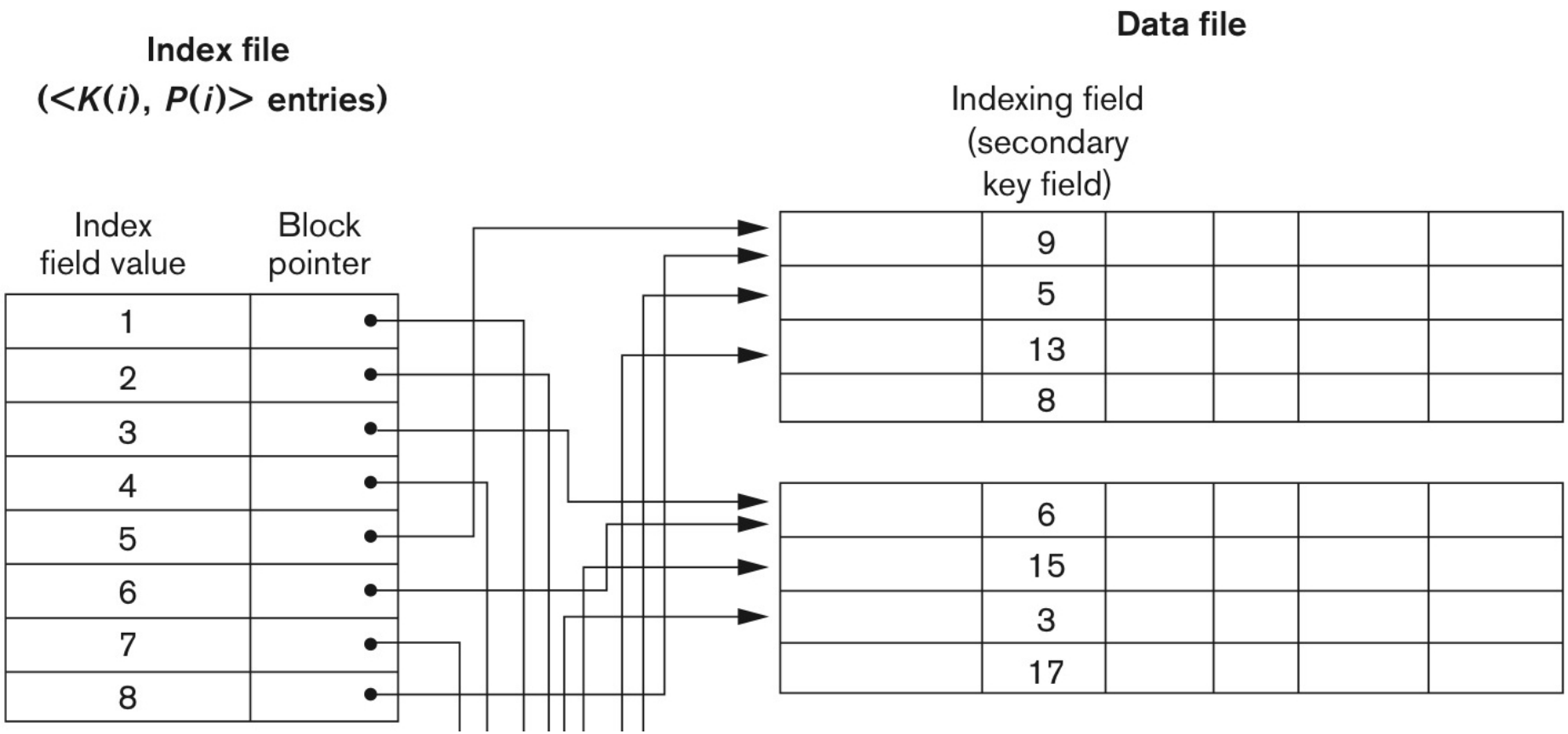
Secondary Indexes (on a key field)

- Secondary means of accessing a data file
- File records could be ordered, unordered, or hashed



Note: The data file is a heap file, i.e., not sorted

Secondary Indexes (on a key field)

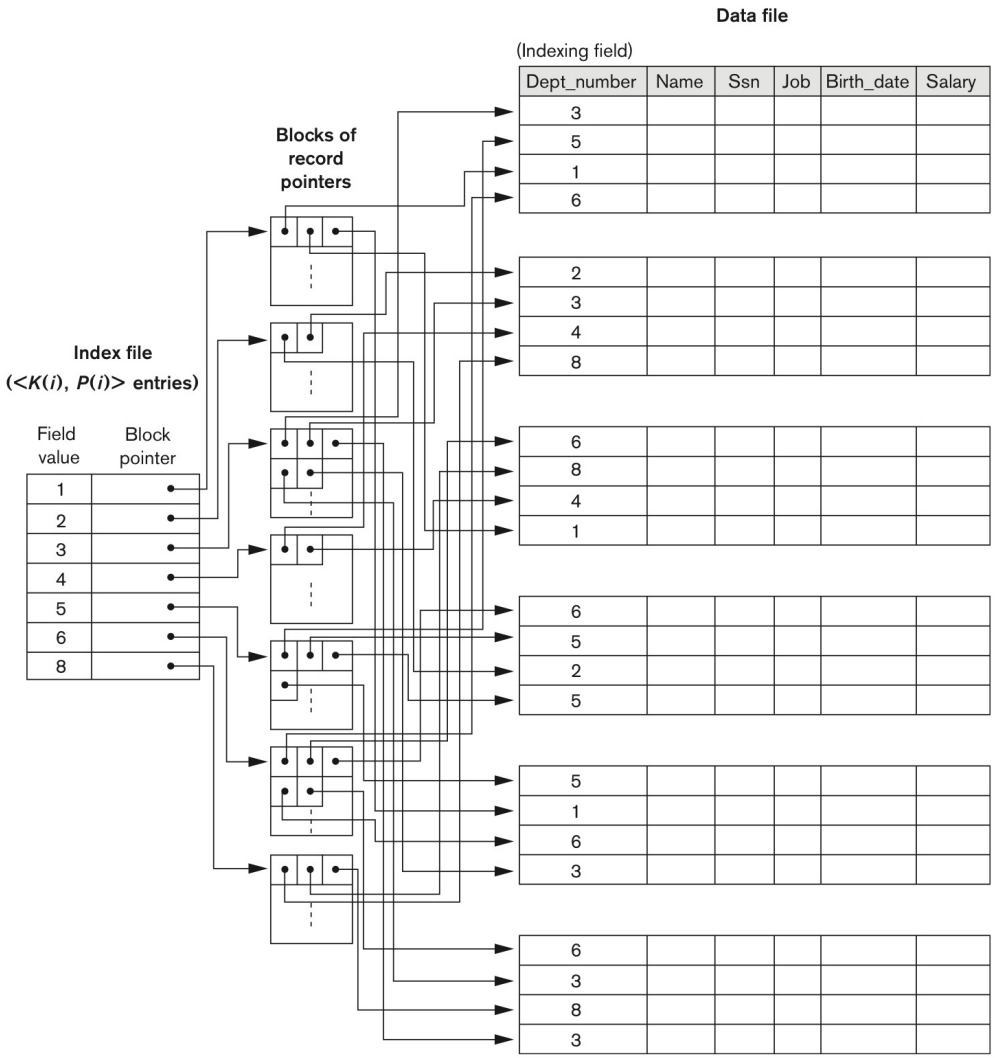


Note: The data file may be a heap file, i.e., not sorted

Secondary Indexes (on a non-key field)

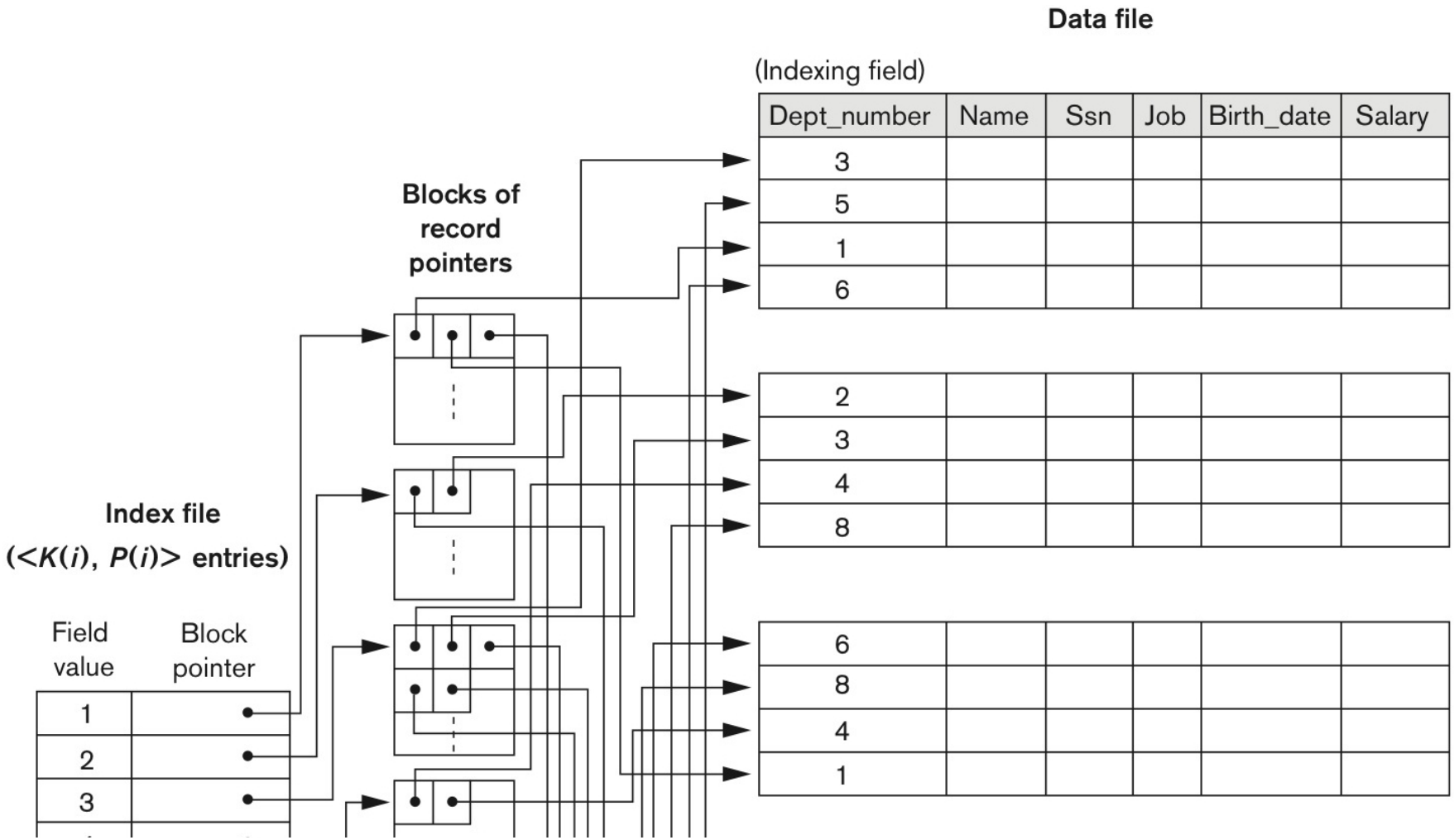
Extra level of indirection

- Provides logical ordering
  - Though records are not physically ordered



Secondary Indexes (on a non-key field)

Extra level of indirection



Reminder

- Please Study Chapter 16 and 17

Acknowledgement

- Some of the slides in this presentation are taken from the slides provided by the authors.
- Many of these slides are taken from cs145 course offered by Stanford University.