UPPSALA UNIVERSITET

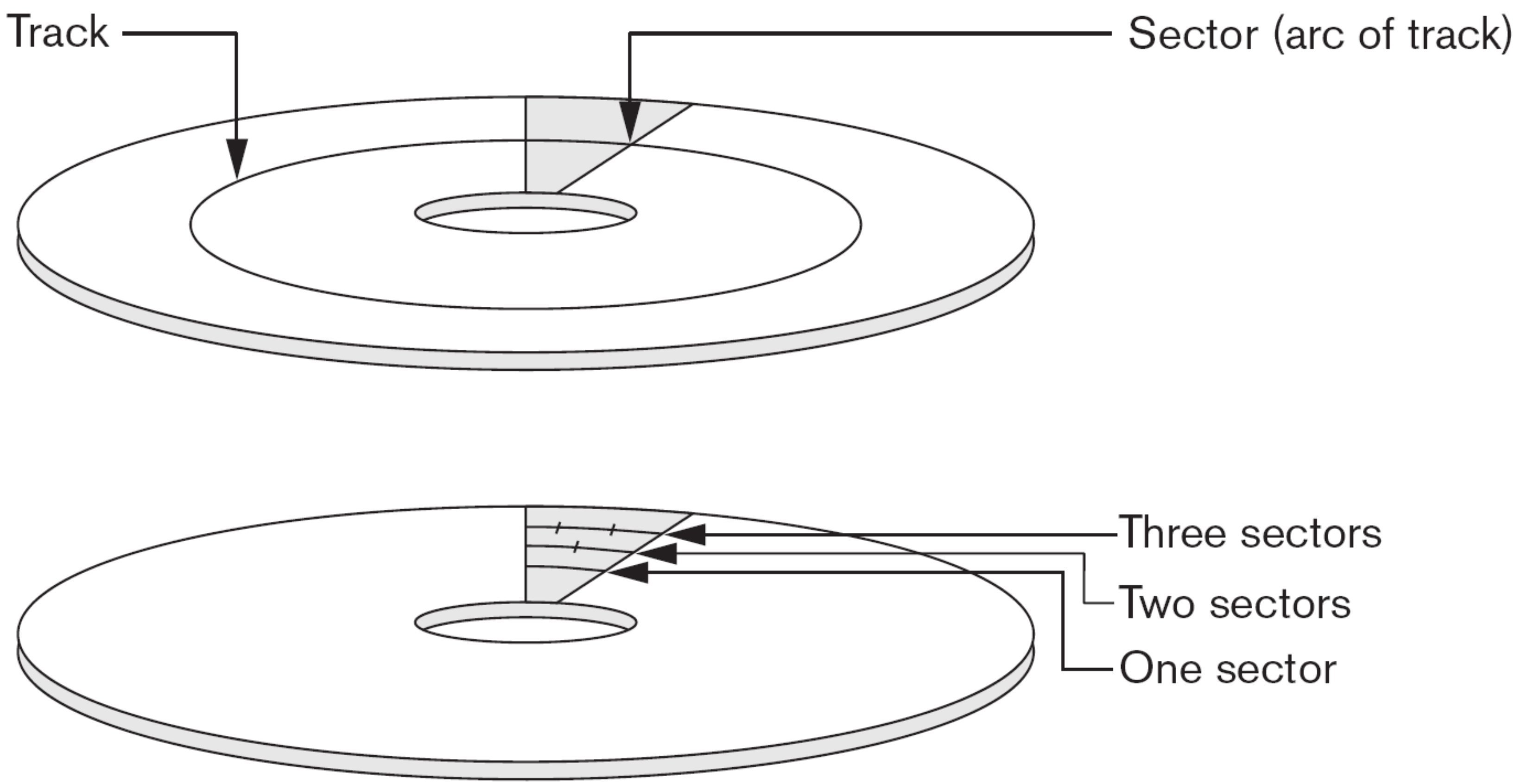
Introduction to physical database tuning (disks, indexes, recommendations)

Lecturer: Neena Thota
neena.thota@it.uu.se

Intended learning outcomes

- Understand the implications of storing data **on disks** for efficiency of data management;
- Know basic **data structures** to store relational data on secondary memory;
- Create **indexes** in SQL;
- Understand the **pros and cons** of different types of indexes;
- Decide when (**not) to create** indexes;
- Make **tuning** decisions.

Structure of a hard disk (platter)



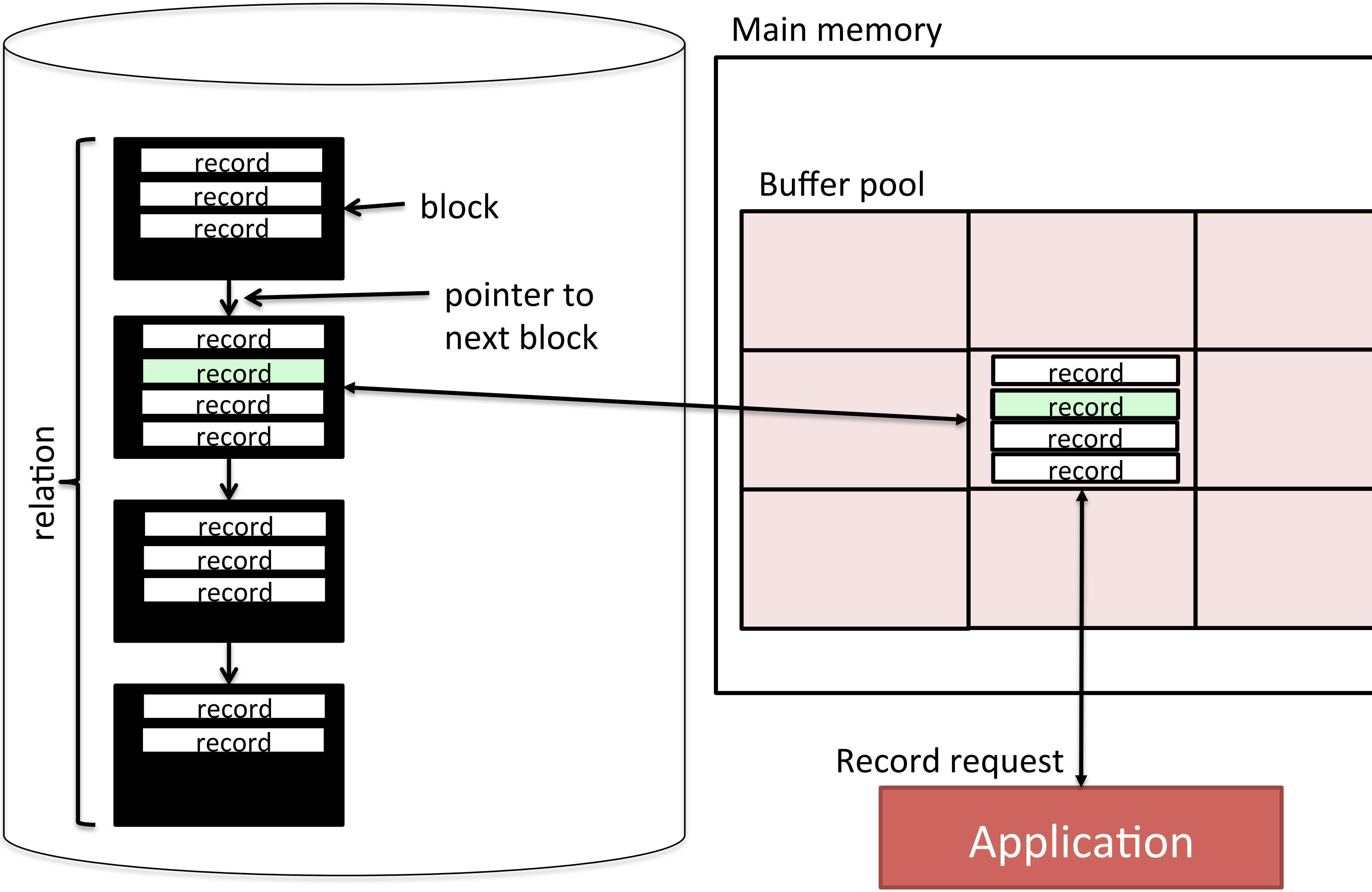
UPPSALA UNIVERSITET

Disk blocks

- A **disk block** corresponds to one or more sectors.
- Blocks are the basic "**physical**" storage structure where relations are stored.

Data access on disk involves a whole block at a time (or multiple blocks)

Overview of relational data storage and access



Buffer manager

- Programs send a request to the buffer manager when they need a block from disk.
  1. Requesting program given address of block in main memory, if it is already present in buffer.
  2. If block is not in buffer, buffer manager allocates space in buffer for block, replacing (throwing out) some other block, if required, to make space for new block.
  3. Block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
  4. Once space is allocated in buffer, buffer manager reads in block from disk to buffer, and passes the address of block in main memory to requester.

Disks vs Main memory

- To access a block (i.e., a set of contiguous sectors) we need around 10*ms*.
- In 10*ms* we can perform around 1 000 000 data accesses on main memory, or execute around 1 000 000 CPU instructions (order of magnitude).

Manipulation time is dominated by data access time when data is on secondary storage

UPPSALA UNIVERSITET

File organization

- **Database**: collection of files.
- **File**: Sequence of records.
- **Record:** sequence of fields.
  - Can have constant (simplest) or variable length.
- A file can store records of same type (simplest) or of different type;
- Specific files used to store specific relations (simplest) or same file can store different relations (maybe even whole database).

Organization of Records in files

- **Heap** – a record can be placed anywhere in file where there is space.
- **Sequential** – store records in sequential order, based on value of search key of each record.
- **Hashing** – a hash function is computed on some attribute of each record; the result specifies in which block of the file the record should be placed.
- **Clustering** – records of several different relations can be stored in same file; related records are stored on the same block.

E.g. Heap files with unordered records

- New records are added to the **end** of the file. Such an organization is called a heap file.
  - Suitable when we don't know how data shall be used.
- **Insertion** of a new record is very **efficient**.
- **Search** after a specific record is **expensive** (linear to the size).
- **Delete** of a record can be **expensive** (search - read into - delete - write back).
  - Instead of physically removing a record one can mark the record as deleted. Both methods require periodically reorganization of file.
- **Modification** of a record of variable length can be **hard**.
- **Retrieval** according to a certain order requires that the file must be **sorted** which is **expensive**.

Why index? Example

- Relation: **EMPLOYEE** (NAME, SSN, ADDRESS, JOB, SAL, ...)
- Suppose that:
  - record size R=150 bytes; block size B=512 bytes; r=30000 records.
- Then, we get:
  - blocking factor Bfr= B div R= 512 div 150= 3 records/block
  - number of file blocks b= (r/Bfr)= (30000/3)= **10000 blocks**

- For an **index on the SSN field**, assume the field size V_{SSN}=9 bytes, assume the record pointer size P_R=7 bytes. Then:
  - index entry size $R_I=(V_{SSN}+ P_R)=(9+7)=16$ bytes
  - index blocking factor Bfr_I= B div R_I= 512 div 16= 32 entries/block
  - number of index blocks b= (r/ Bfr_I)= (30000/32)= **938 blocks**

- **Block accesses**
  - average linear search cost: (b/2)= 30000/2= **15000 block accesses**
  - if file records are ordered, binary search cost would be: $\log_2 b = \log_2 30000$= **15 block accesses**
  - binary search of index blocks $\log_2 bI = \log_2 938$= **10 block accesses**

Calculating the best attribute to index - Example

- Relation: **StarsIn**(movieTitle, movieYear, starName)
- **Frequent queries**
  - *Q*i: We look for **the title and year** of movies in which a **given star** appeared.
  - *Q*2: We look for the stars that appeared in a **given movie**.
  - We **insert** a new tuple into StarsIn.
- **Assumptions:**
  - StarsIn occupies **10 blocks.**
  - On the average, **a star has appeared in 3 movies and a movie has 3 stars**.

Example continued…

- Costs of each of the three operations in terms of block access:
  - *Q*i (query given a star),
  - *Q2* (query given a movie)
  - *I* (insertion).

Action	NO Index	Star Index	Movie Index	Both Indexes
Q1	10	4	10	4
Q2	10	10	4	4
I	4	4	4	6

UPPSALA UNIVERSITET

Example cont. Lessons learned

- If only **one type of query is frequent**, create only the index that helps that type of query;
- If we are doing **mostly inser**tion, and very few queries, then we don't want an index;
- Structuring the data so that we know where to look for specific records can dramatically **speed up** query execution;
- But indexes require space and may **slow down** updates –> good to have a clear idea of **where indexes** can be useful.

UPPSALA UNIVERSITET

Index creation with SQL

```
CREATE [UNIQUE] INDEX IndexName ON TableName(AttributeList)
```

```
DROP INDEX IndexName
```

```
CREATE INDEX EmpPK ON Employee(EmpID)
```

```
DROP INDEX EmpPK
```

Not standard SQL, but de facto standard

When Is the Index Used?

- Perform **constant arithmetic**
  - SELECT * FROM EMP WHERE salary/12 > 4000 (may not use index)
  - SELECT * FROM EMP WHERE salary > 48000 (will use index)
- Use **built-in operations** rather than functions
  - SELECT * FROM EMP WHERE SUBSTR(name, 1, 1) = 'G‘ (may not use index)
  - SELECT * FROM EMP WHERE name LIKE 'G%‘ (will use index)
- **Nulls** are often not indexed
  - SELECT * FROM EMP WHERE salary IS NULL (may not use index)
- Nested sub-query;
- Selection by negation;
- Queries with OR.

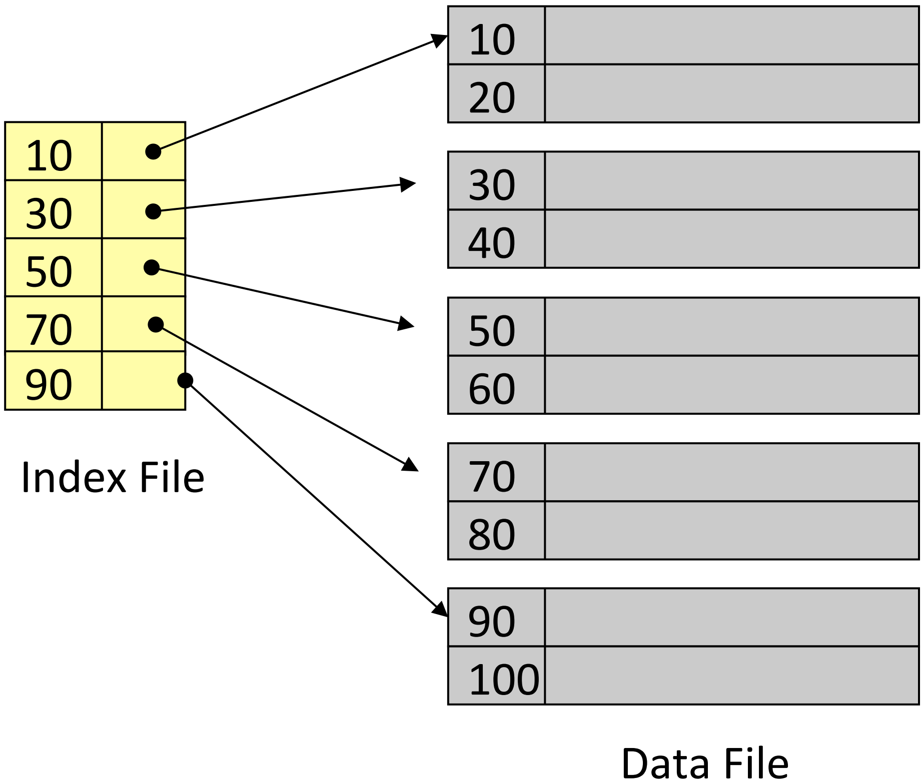
UPPSALA UNIVERSITET

Indexes as Access Paths

Value	Block/record pointer

- Index file usually occupies considerably less disk blocks than data file because its entries are much smaller;
- A binary search on the index yields a pointer to file block/record;
- Indexes can be characterized as:
  – **Dense index** has an index entry for every search key value (and hence every record) in the data file.
  – **Sparse (or nondense) index** has index entries for only some of the search values.

Primary Index - Sparse



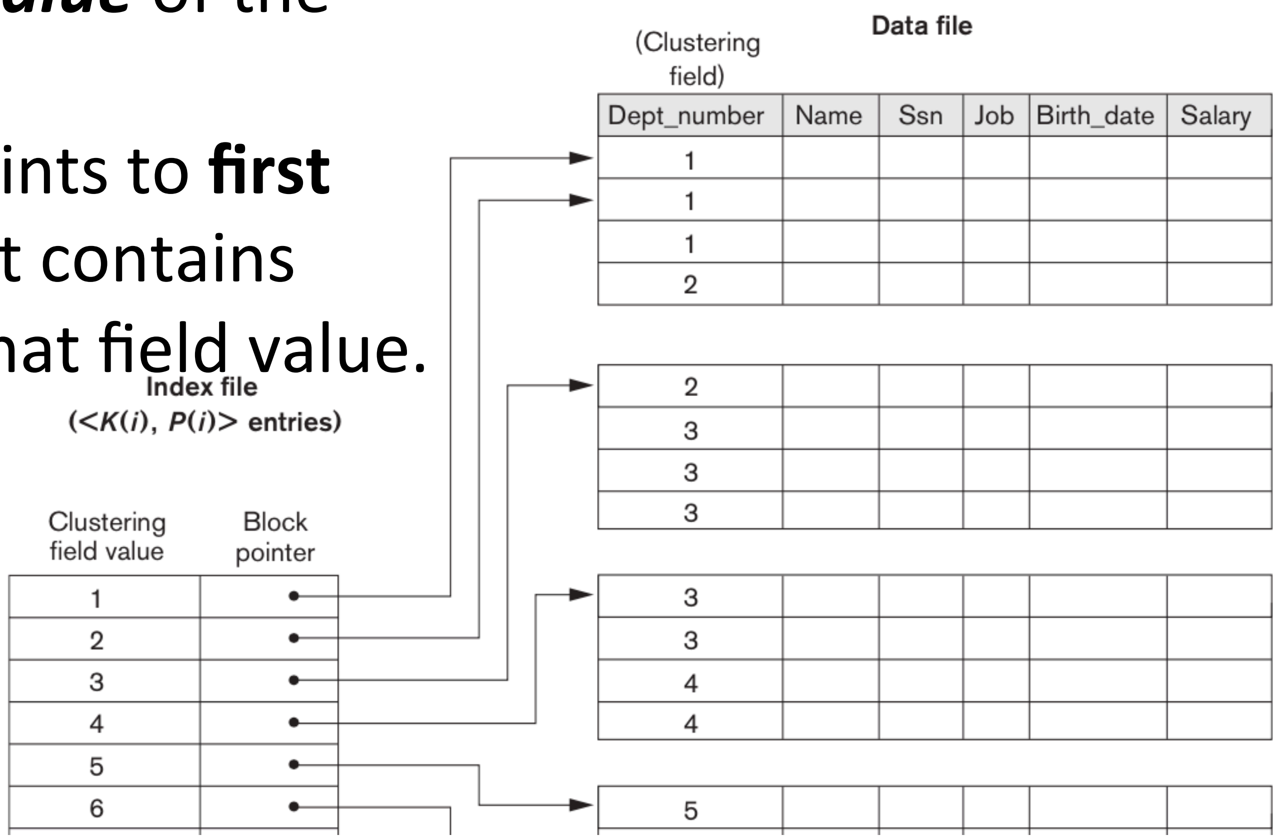
- Defined on an **ordered** data file with a **key field;**
- Includes one index entry ***for each block*** in data file;
- Index entry has key field value for *first record* in block, called ***block anchor.***

Primary index - pros and cons

- Require **much less space** than the data file.
  - a) There are much fewer index records than records in the data file.
  - b) Every index record needs less space ($\Rightarrow$ fewer memory blocks).
- Problem with **insertion and deletion** of records.
  - To insert a record in its correct position in data file, need to move records to make space for the new record.
  - If anchor records are changed the index file must be updated.

Clustered index - Sparse

- Defined on data file ordered on a ***non-key field.***
- Includes one index entry *for* ***each distinct value*** of the field;
- Index entry points to **first data block** that contains records with that field value.

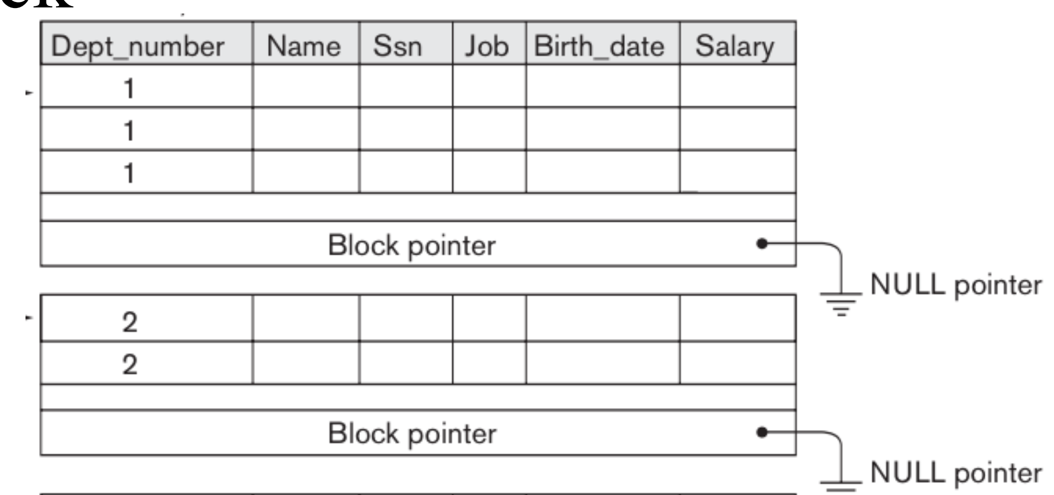


Clustered Index – Pros & Cons

- **Speeds up retrieval** of all records that have same value for clustering field.
- Record insertion and deletion still **cause problems** because data records are physically ordered.

Solution:

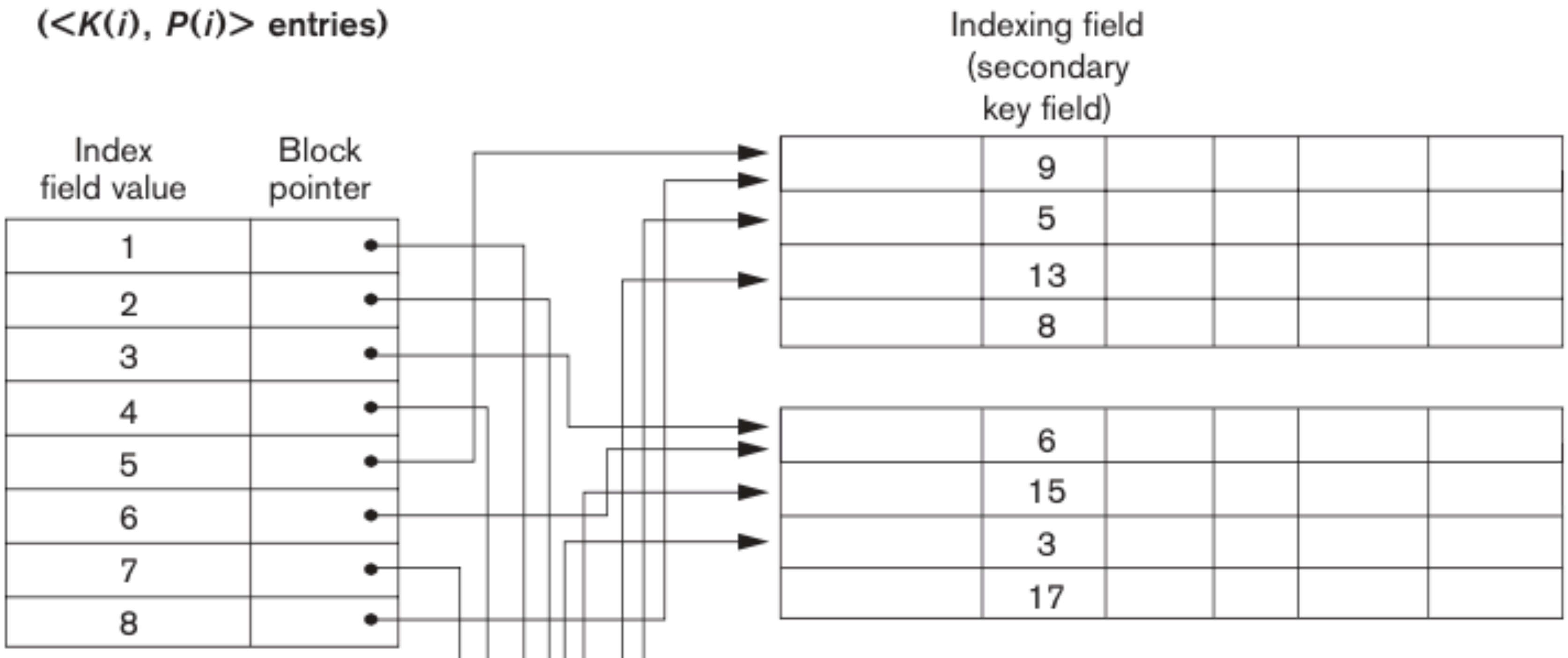
- Reserve a whole block (or a cluster of contiguous blocks) for ***each value*** of clustering field;



Secondary Indexes

- For accessing a file for which some primary access already exists.
- May be on field which is **candidate key** and has unique value in every record, or a **non-key with duplicate values**.
- Can have **many secondary indexes** (and indexing fields) for same file.
- Secondary index is ordered file with two fields:
  - First field is of same data type as some *nonordering field* of data file that is an **indexing field**.
  - Second field is either a *block* pointer or a *record* pointer.

Secondary Index - Dense



Example: A **dense secondary index** (with block pointers) on a non-ordering key field of a file.

- One index entry for ***each record*** in the data file,
  - which contains the value of the field for the record and a pointer either to the block in which the record is stored or to the record itself.

UPPSALA UNIVERSITET

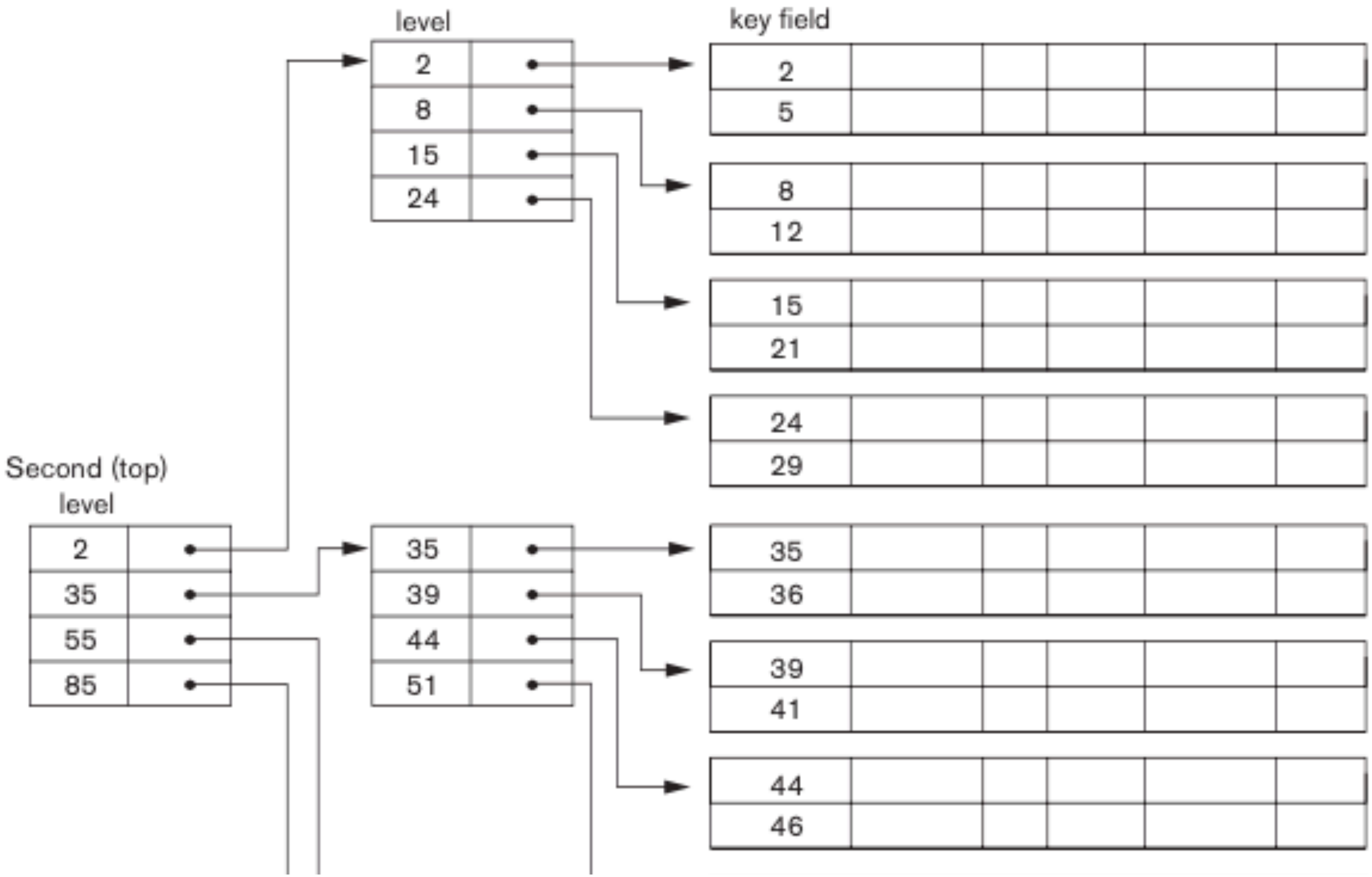
Secondary indexes – Pros & Cons

- **Improvement in search time** for an arbitrary record much greater for a secondary index than for a primary index, since we do a *linear search* on data file if secondary index did not exist;
- Sequential scan using secondary index is **expensive** (each record access may fetch a new block from disk);
- Needs **more storage space and longer search time** than does a primary index, because of its larger number of entries.

Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index ***to the index itself***;
  - Original index file is called ***first-level index*** and index to the index is called ***second-level index***.
  - Can repeat the process until all entries of the *top level* fit in **one disk block.**
- Can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of ***more than one*** **disk block.**

Example: Multilevel index



UPPSALA UNIVERSITET

Summary Types of indexes

- An index is a collection of pairs: search key – pointer to record(s).
- **Contains primary key**?
  - YES: Primary index
  - NO: Secondary index
- **Has the same (or similar) order of the records found in the indexed relation?**
  - YES: Clustered index
  - NO: Unclustered index
- **Contains all values of the search key that are present in the indexed relation?**
  - YES: Dense
  - NO: Sparse

Which one is false?

1. Primary index can be used when index field is used for physical ordering of file.
2. Clustered index can be created on nonkey ordered indexing field.
3. Secondary index with key can be dense.
4. A sparse index can be clustered.
5. None

Some Tuning Decisions

Design decisions about indexing. Whether to

- index **an attribute**
- **or attributes** to index on
- set up a **clustered** index

Tuning Indexes

Reasons to tune indexes

- Certain queries may take **too long to run** for lack of an index;
- Certain indexes may **not get utilized** at all;
- Certain indexes may be causing **excessive overhead** because the index is on an attribute that undergoes frequent changes.

Options to tuning indexes

- **Drop or/and build** new indexes;
- Change a non-clustered index to a **clustered index** (and vice versa);
- **Rebuild** the index.

Other Tuning Considerations

- **Use enough RAM.**
  - 3-4 GB RAM costs very little these days.
  - Make sure the DBMS uses it.
- **Consider using more disks**.
  - More disks = more seeks per second.
- **Tune database design – use wisely!**
  - Combine commonly joined tables into one (de-normalization).
  - Avoid join with another table by adding to a table attributes that are needed for answering queries or producing reports.
  - Trade off between update and query performance
  - Repeat values in other tables (redundancy).
- **Tune queries.**
  - Check that queries use the right indexes.
  - Do not create unnecessary indexes.