# Database Structures

Oscar Romero

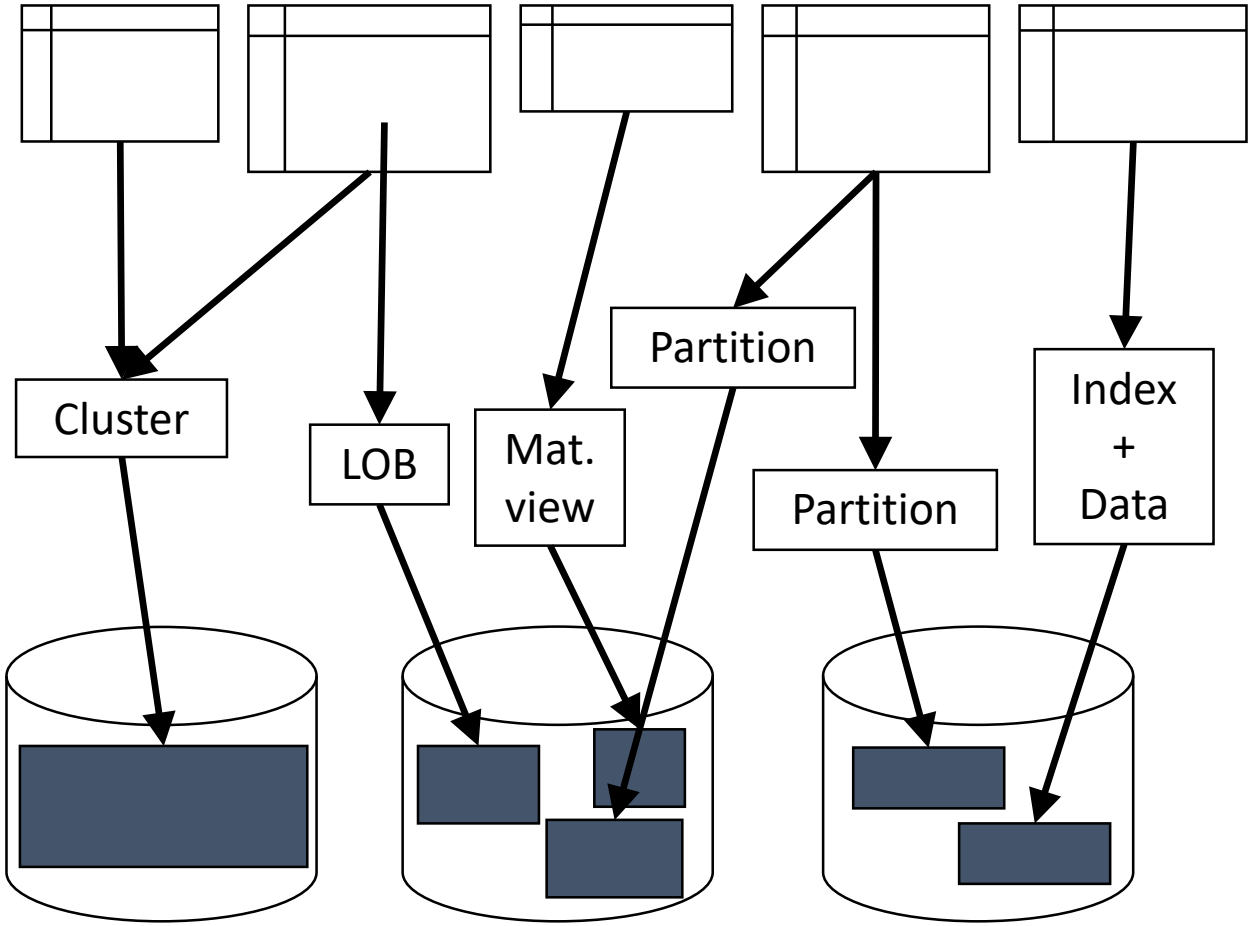
DTIM RESEARCH GROUP (http://www.essi.upc.edu/dtim/)

UNIVERSITAT POLITÈCNICA DE CATALUNYA - BARCELONATECH

# Table of contents

- Database internal structure
  - Three spaces
- Main Data Structures
- Table files
  - B-tree
  - Index cluster
  - Hash
- Access patterns
- Sequential access
- Random access
- Index-range access

# Three Spaces

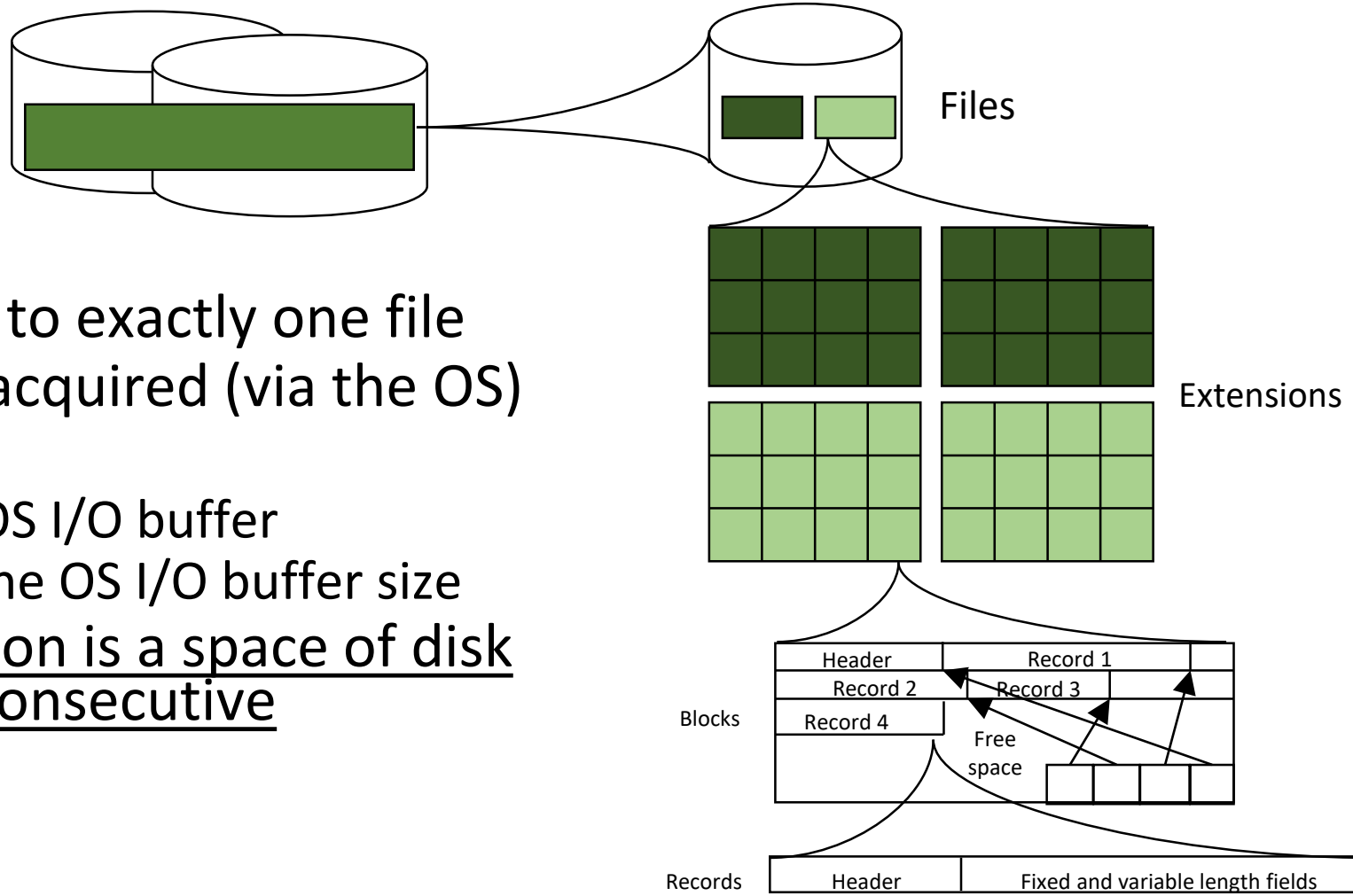



- Logic space
  - Tables (relations)
    - Rows (tuples)
    - Cols (attributes)
- Virtual space (database objects)
  - Pages
    - Records
      - Fields
  - Partitions
  - Views
    - Not materialized
    - Materialized
  - Indexes
  - Clusters
  - Tablespaces
- Physical space
  - Files
    - Extensions
      - Blocks

# Three Spaces

- A database exposes the logic space to the user. A relational database uses the relational model as metaphor to communicate with the user (tables, columns, rows, etc.).
- The physical space is where the data resides and it is managed by the operating system. At disk we store files, which contain files which, in turn, contain blocks.
  - A block is the minimum I/O unit. When reading data from disk we read block-wise.
  - When the operating system (OS) creates or appends space to an existing file it does so by means of extensions. An extension is a set of consecutive blocks in disk. But two extensions might not be consecutive. Therefore, a file is nothing else than a list of extensions.
- The virtual space is the database main memory. A database does not deal with files directly but with database objects. Each database object creates physical files with different structure. The database management system is the one in charge to read a file from disk into memory and interpret it as a certain kind of object: tables, indexes, materialized views, clusters, etc.
  - The mininum management unit in memory is the page. A page contains records (i.e., the logical counterpart of the table rows), which contain fields (columns).
  - The database objects are nothing else than a way to organize the physical files in an understandable manner and facilitate its management. Note, however, the database objects do not really exist at the physical space, but only in the database virtual space.

# Logical - Physical Space Relationship




Extensions are assigned to exactly one file

They are automatically acquired (via the OS)

Their size is typically:

- Much bigger than the OS I/O buffer
- A number multiple of the OS I/O buffer size

By definition, an extension is a space of disk (i.e., blocks) physically consecutive

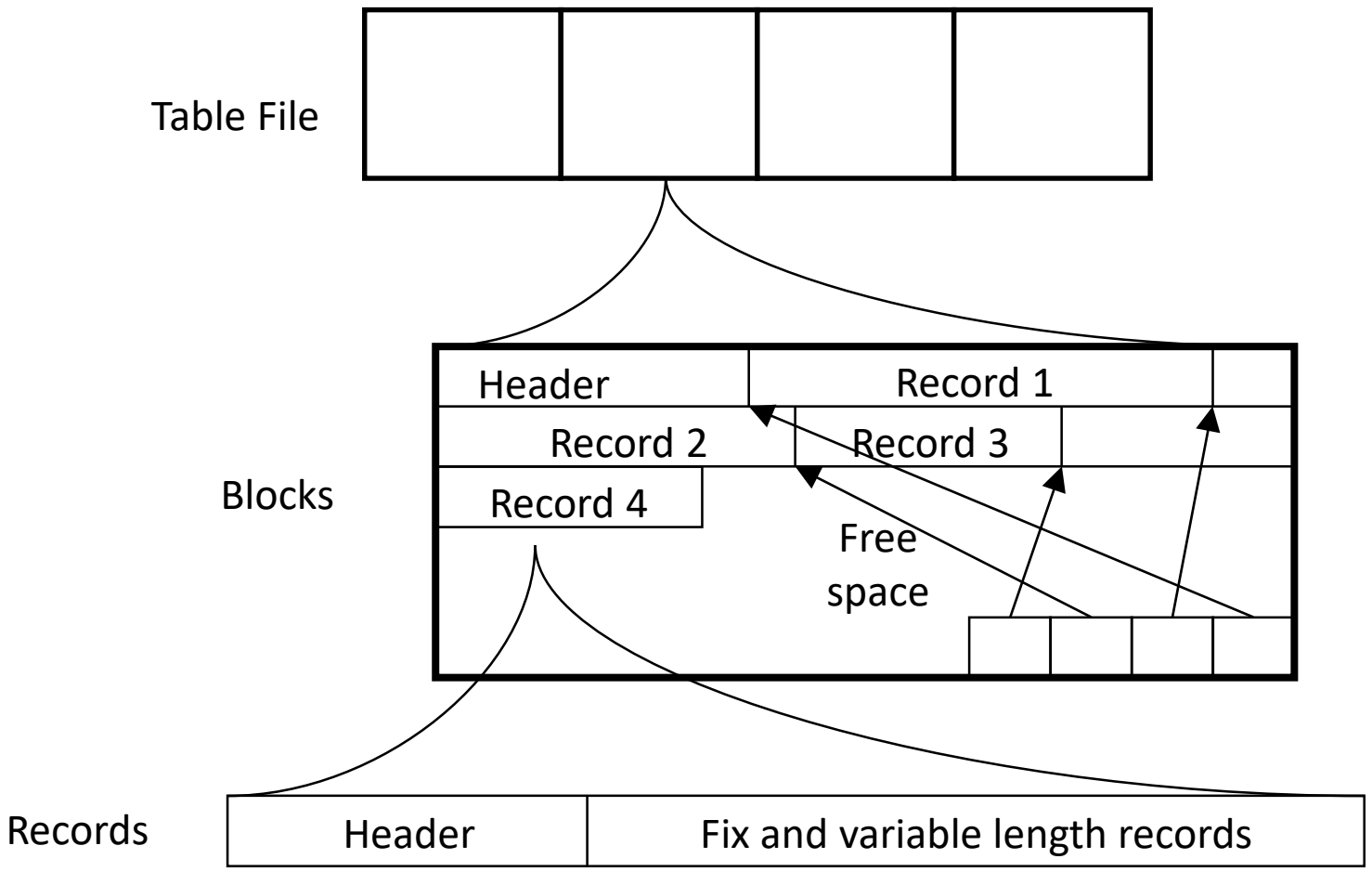
# Main Data Structures

# Main Data Structures

- The table file is the usual structure to store a table
  - They can only be traversed and therefore, the database typically explores a **table file** by means of sequential accesses
- Indexes are structures used to avoid traversing the whole file and instead accesing a specific block at disk
  - They do not rely on sequential accesses but in random accesses
  - The main indexing structures in a database are **B-trees**, **hash indexes** and **bitmaps**
    - **B-tree:** https://www.cs.cornell.edu/courses/cs3110/2012sp/recitations/rec25-B-trees/rec25.html
    - **Hash tables:** https://www.cs.cornell.edu/courses/cs2112/2020fa/lectures/lecture.html?id=hashtables

Read about b-trees and hash tables before moving forward with the slides

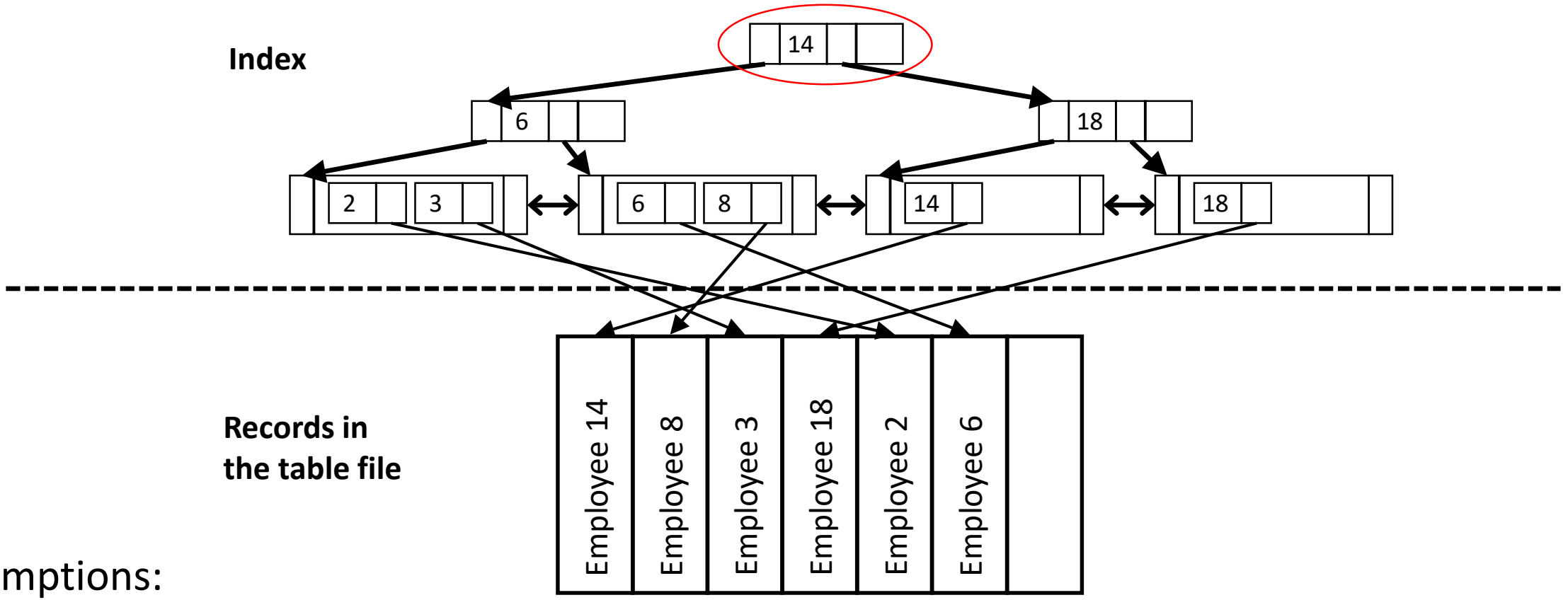
# Table Files




***Note***: *For the sake of simplicity, this slides ignore extensions and only show files and blocks*

# B-Tree (B+)

Every tree node is a block with a very specific format!




Assumptions:

- In every tree node (a disk block) 2d addresses fit; d is the tree order
- Usually, the tree load is 66% (2/3 of its capacity)

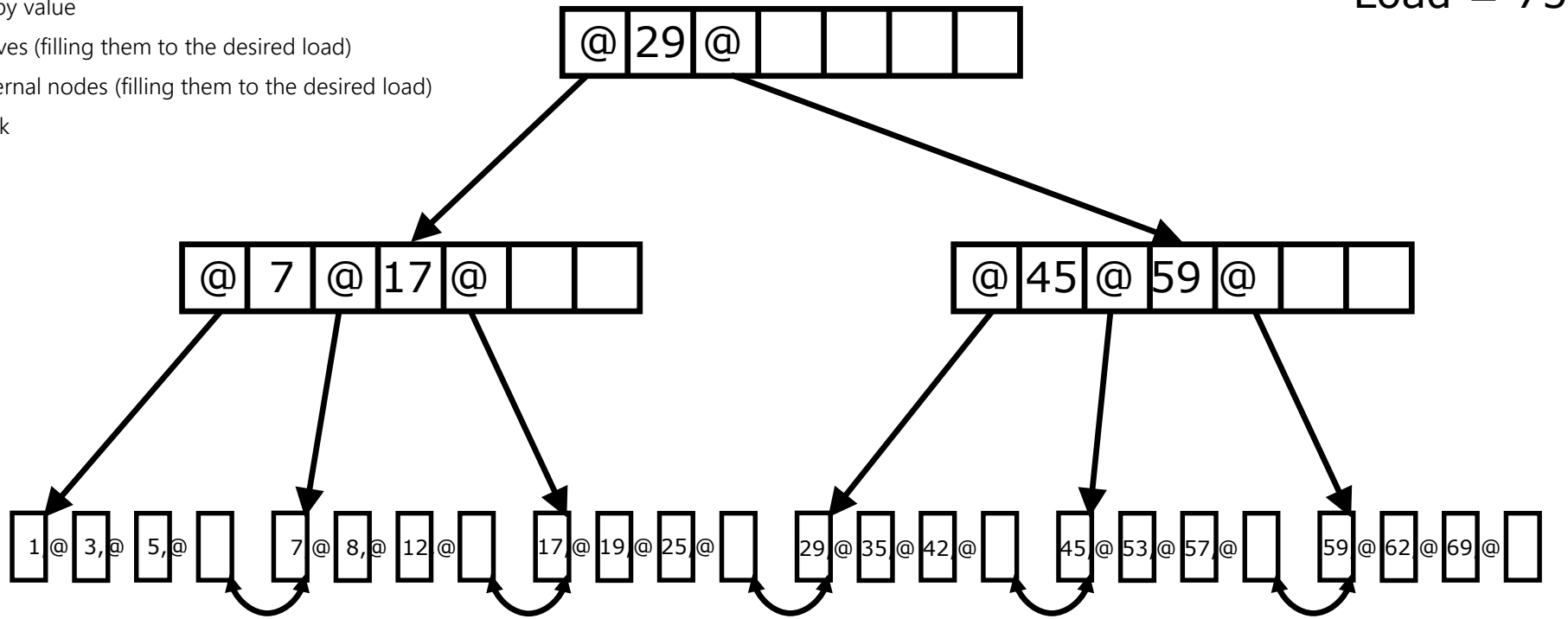
# Algorithm to build a B-tree (bulk mode)

1. Create a file with the entries [value,RID]
2. Sort the file by value
3. Build the leaves (filling them to the desired load)
4. Build the internal nodes (filling them to the desired load)
5. Save it to disk

# Example of Building a B-tree (bulk mode)

1. Create a file with entries [value,RID]
    1. The RID is the record physical address in the table file
2. Sort the file by value
3. Build the leaves (filling them to the desired load)
4. Build the internal nodes (filling them to the desired load)
5. Save it to disk

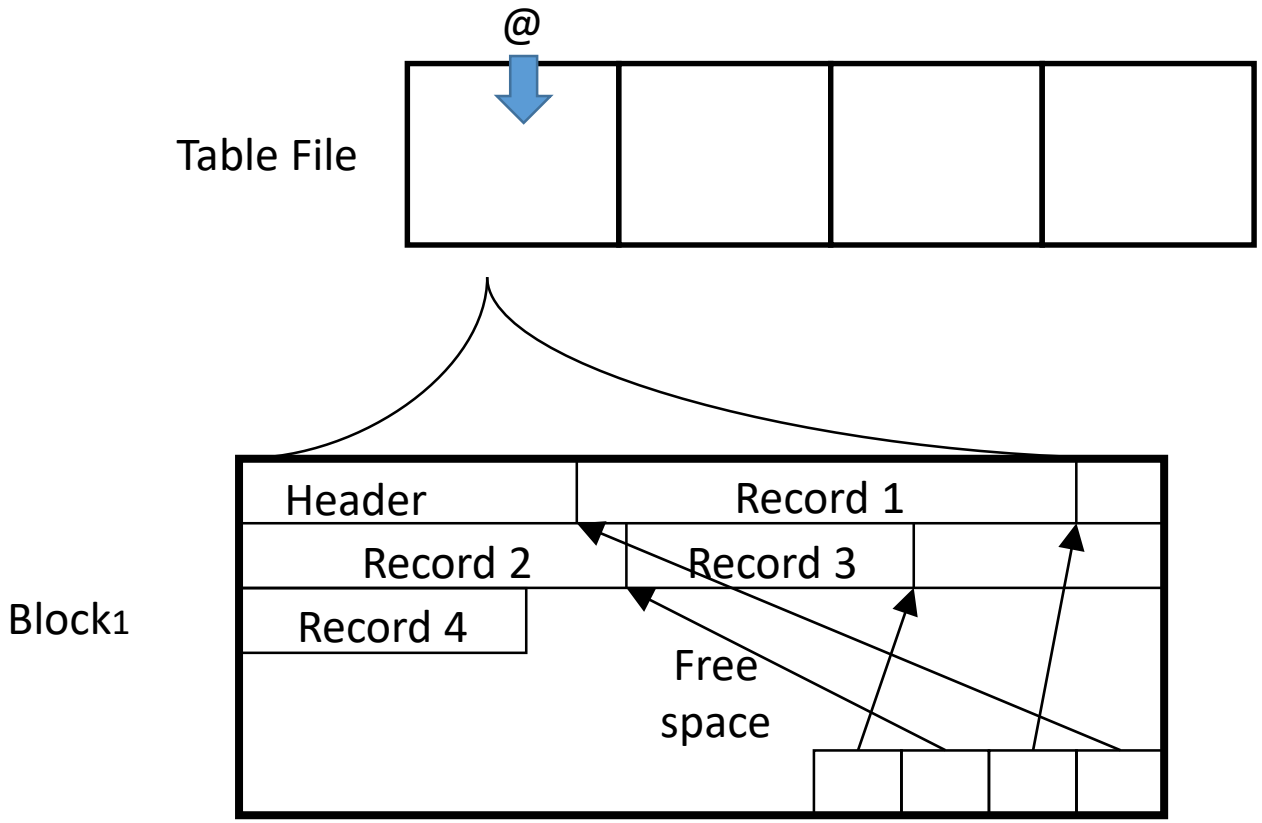
Order = 2
Load = 75%




[1,@],[3,@],[5,@],[7,@],[8,@],[12,@],[17,@],[19,@],[25,@],[29,@],[35,@],[42,@],[45,@],[53,@],[57,@],[59,@],[62,@],[69,@]

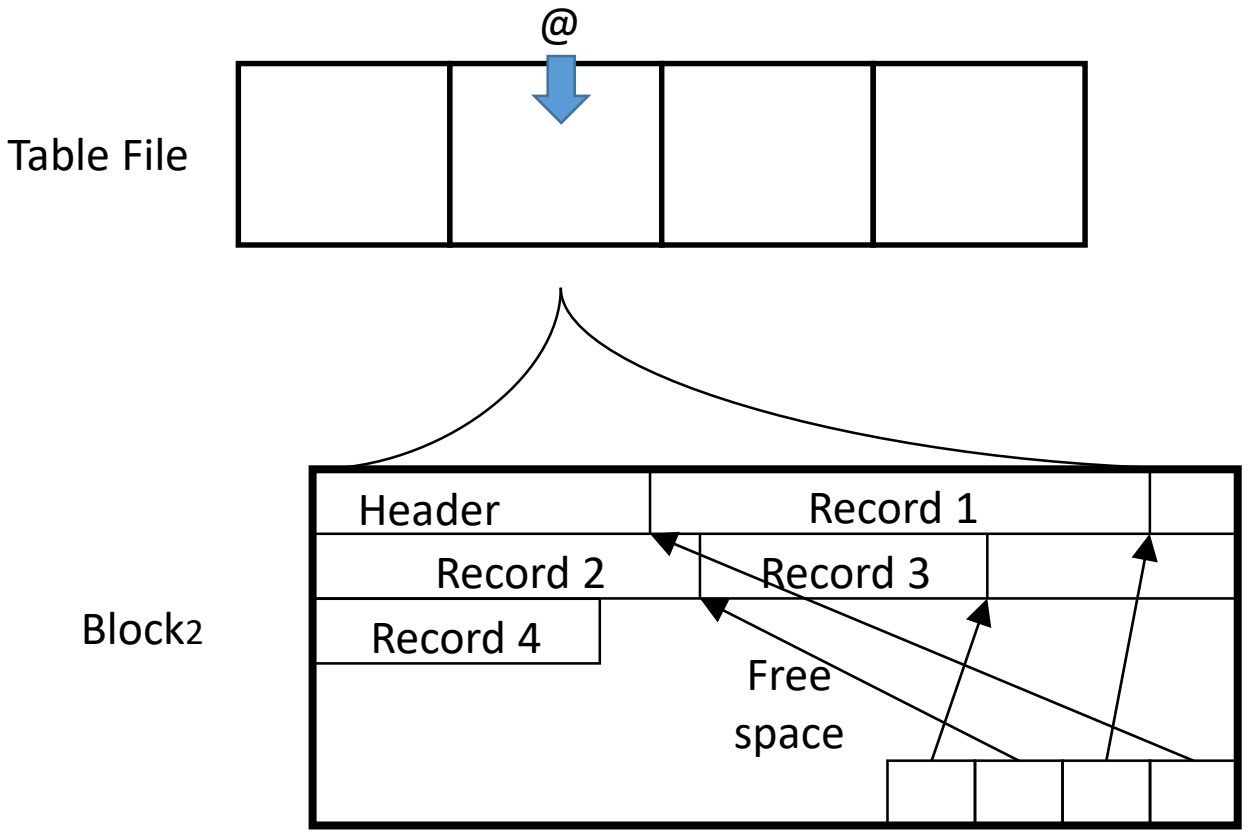
# Sequential Vs. Random Access

- Sequential Access:



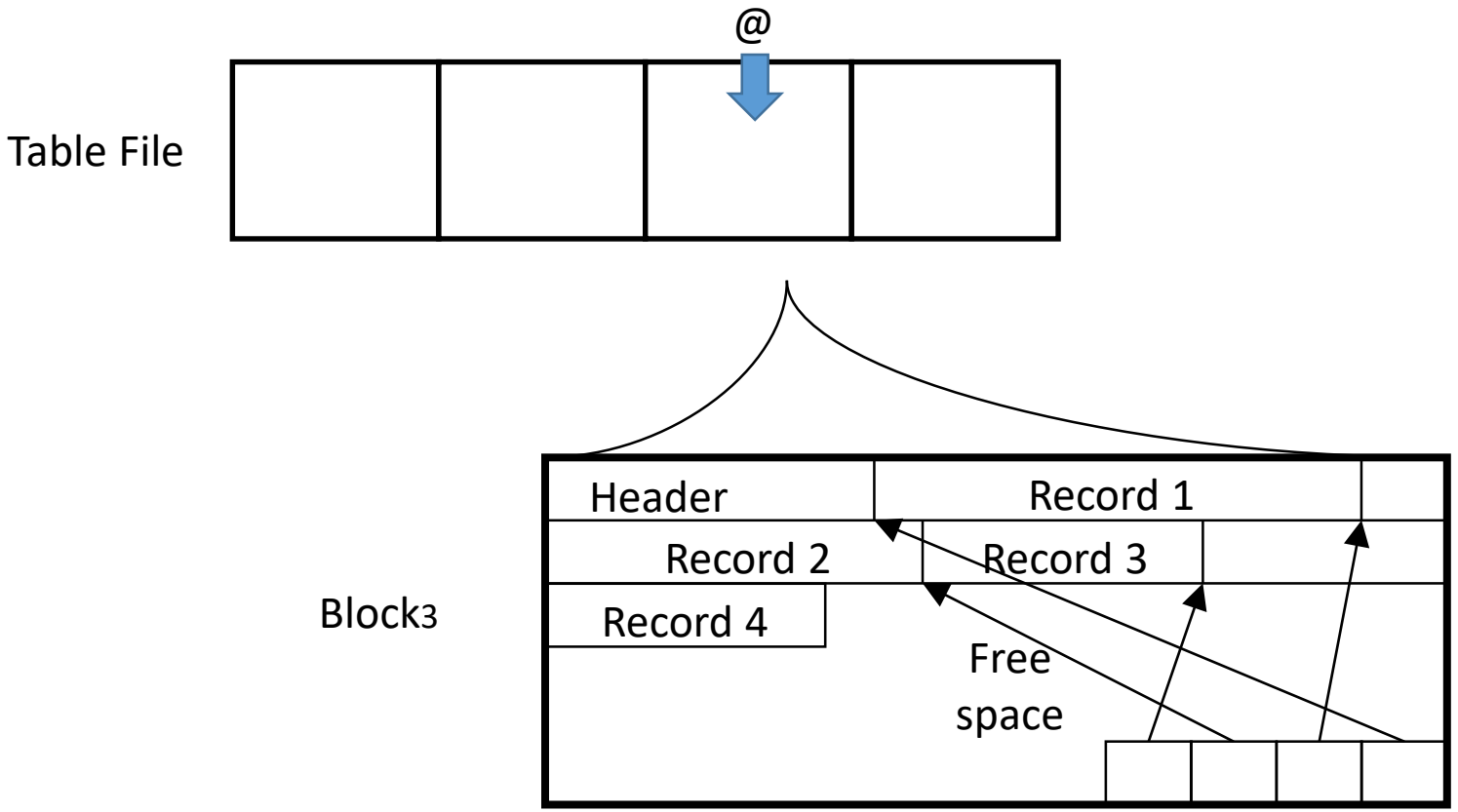
# Sequential Vs. Random Access

- Sequential Access:



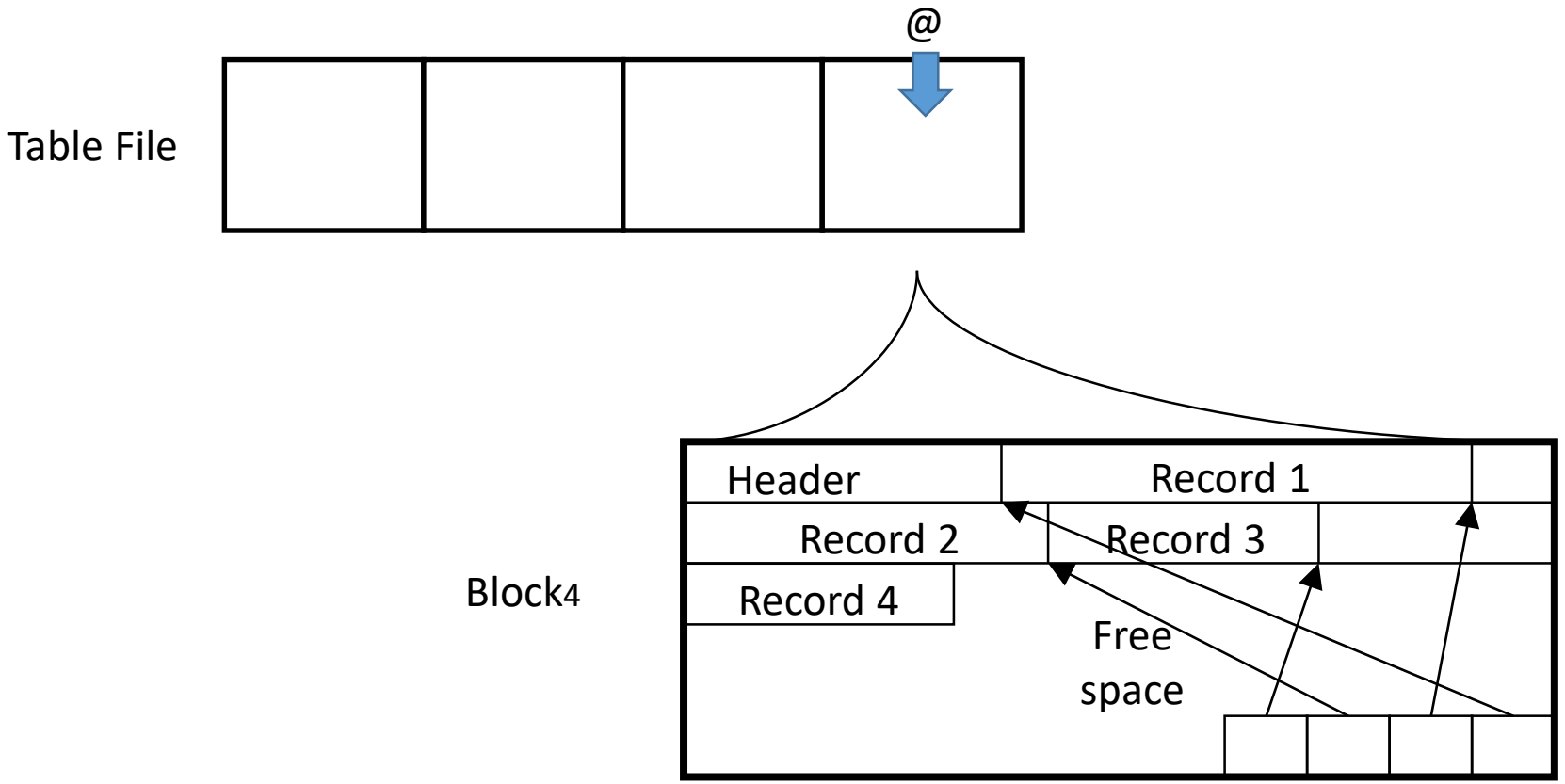
# Sequential Vs. Random Access

- Sequential Access:



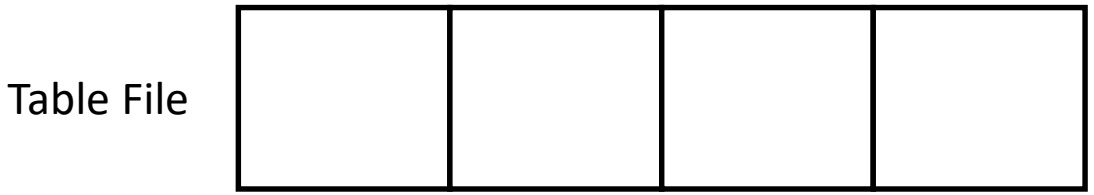
# Sequential Vs. Random Access

- Sequential Access:



# Sequential Vs. Random Access

- Sequential Access:

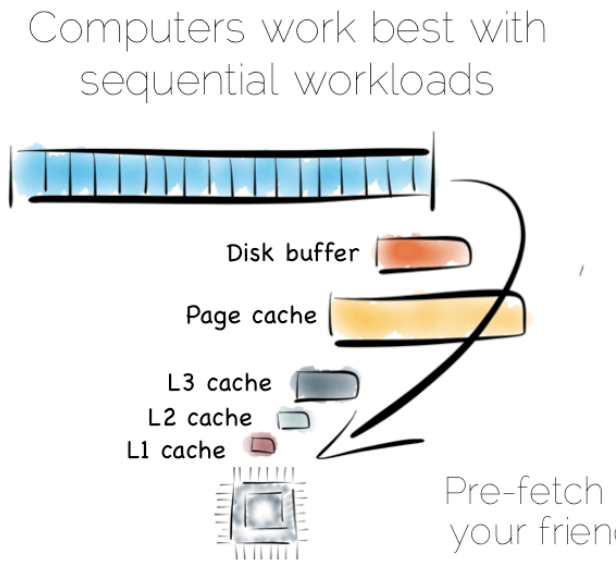



Total number of blocks read: 4
Total number of records read: the whole table

**Table scan: A sequential read of the whole table file retrieving all records**

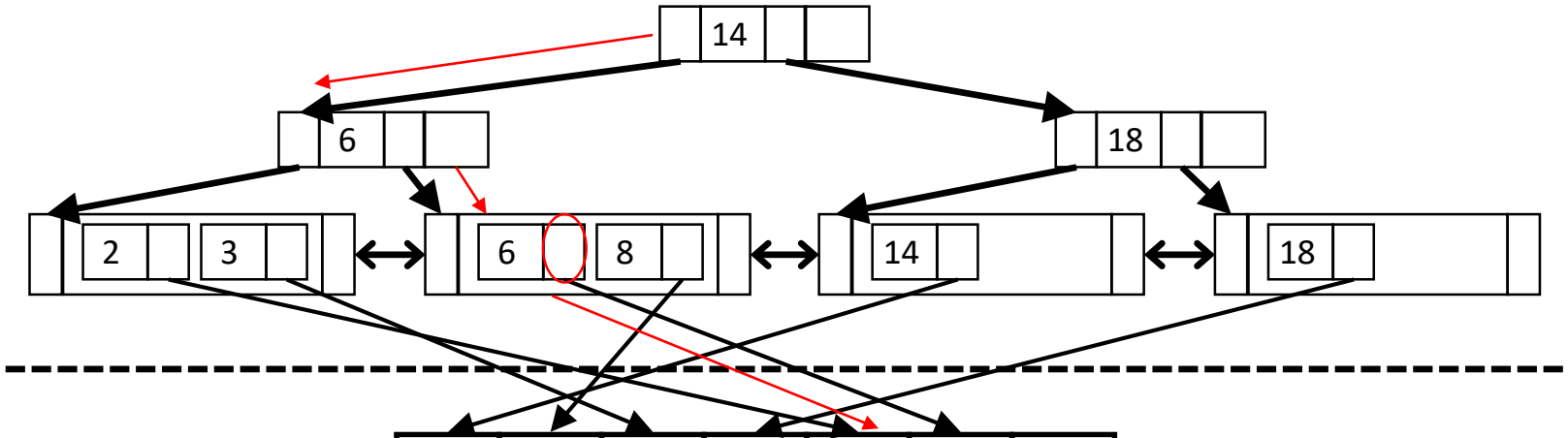
**Pros**: To read the whole table a table scan is the minimum number of I/O

**Cons**: But, if we want to read just one record, a table scan is extremely inefficient

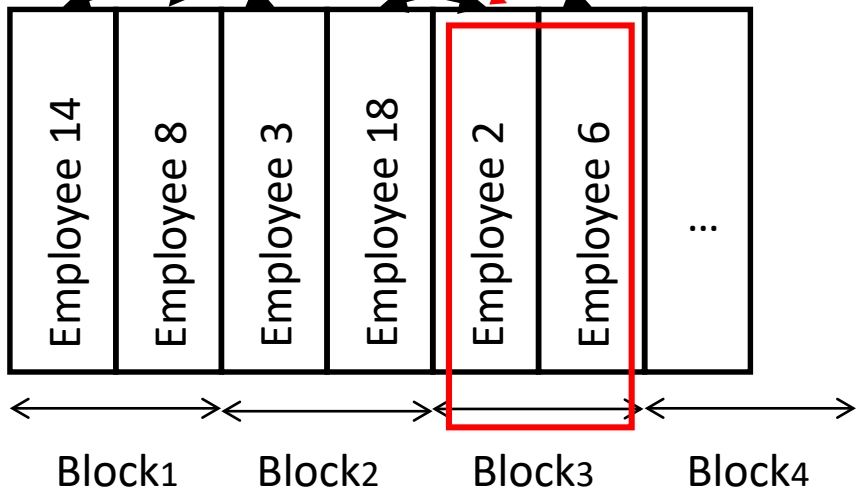


# Sequential Vs. Random Access

- B-tree:




**Records in the table file:**




**Query**. Employee where A = 6

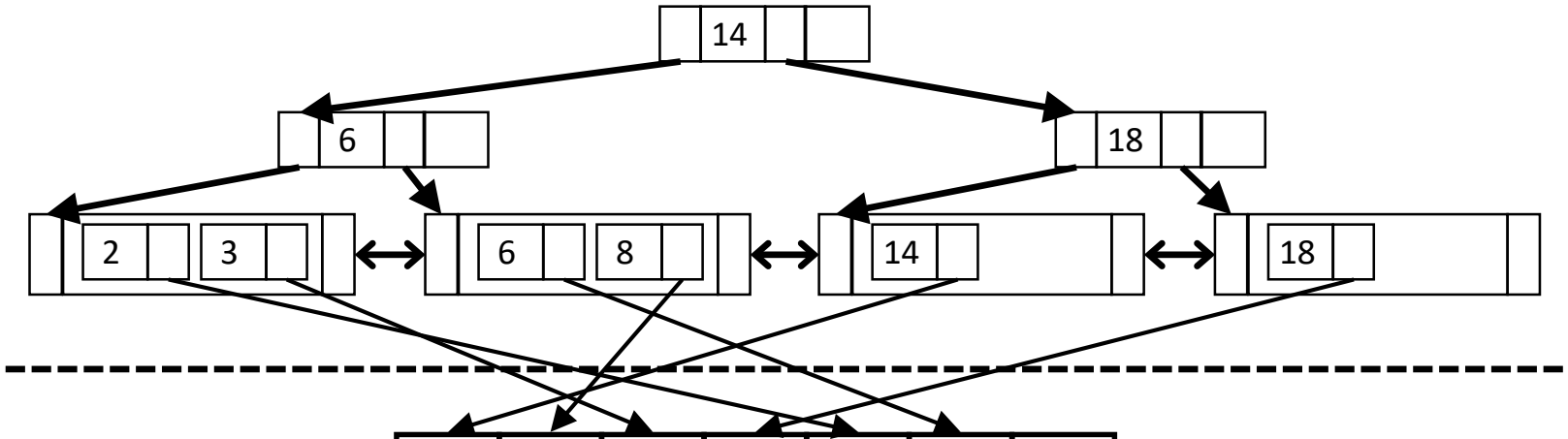
#Table blocks read with table scan? 4 (#B)

#Table blocks read using a B+? 1

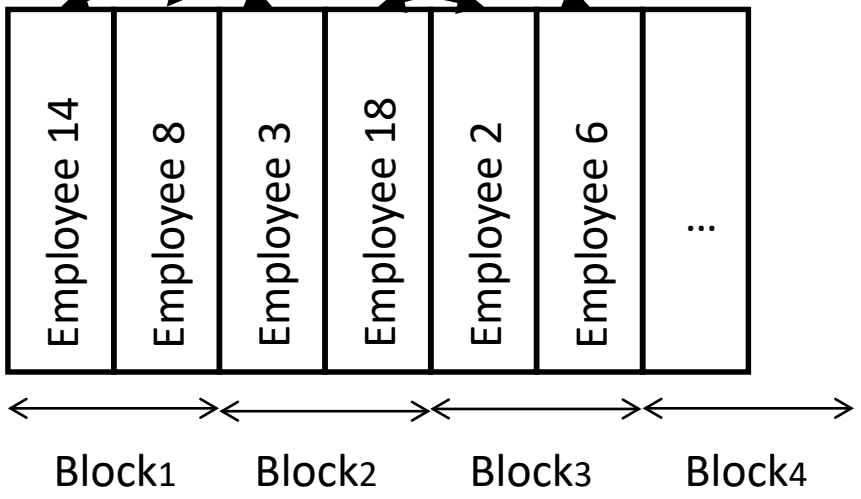
**Note**: *databases typically keep indexes in memory and therefore they do not need I/O operations to read their blocks*

# Sequential Vs. Random Access

- B-tree:




**Records in the table file:**




**Query**. Read the whole table

#Table blocks read with table scan? 4 (#B)

#Table blocks read using a B+? 5

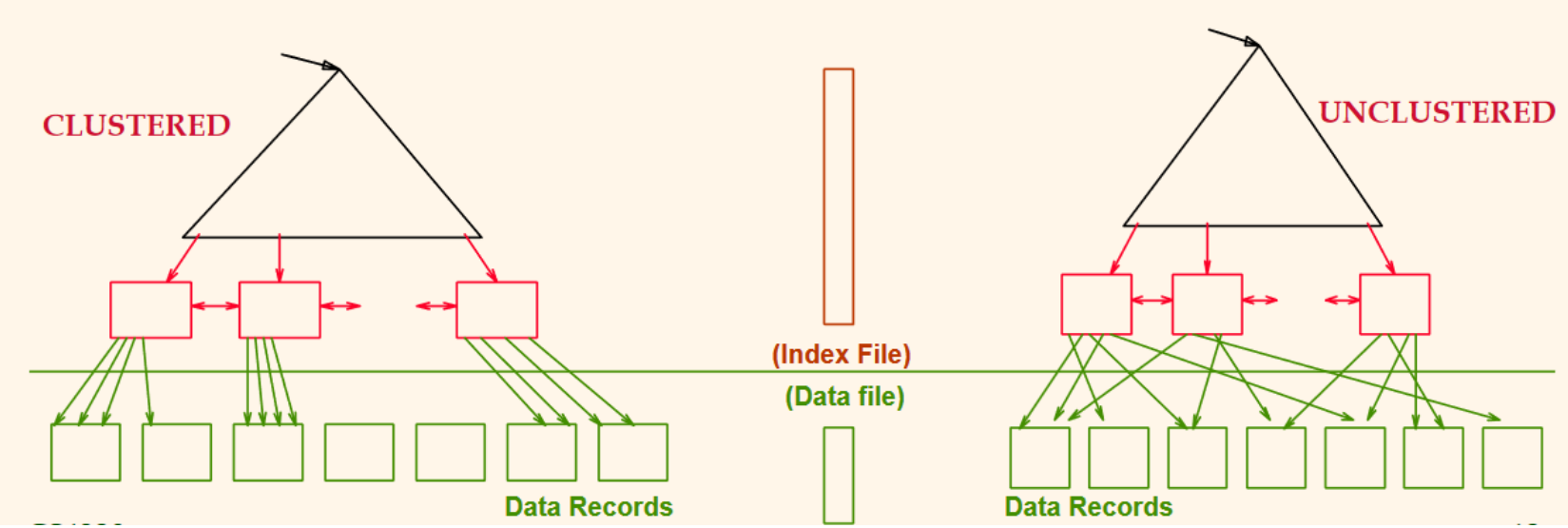
*(asume the virtual space only contains 1 page and assume it is equivalent to one physical block)*

# Sequential Vs. Random Access

- The key question is *what is the number of records from which it is better to do a sequential access (i.e., read the table file) instead of using random access (i.e., use an index)*? It depends on many factors; e.g.:
  - Record size,
  - Block size,
  - Tree order,
  - And in general on any physical parameter determining the amount of data in each I/O operation and the degree of parallelism achieved.
- In general, for relational databases, the following thumb rule applies:
  - Reading above 10% of the table records it is better to perform a sequential access,
  - Below 10%, better random accesses.
- However, realise this is a **heuristic**, not a rule. In Big Data systems, like HDFS/HBase random accesses above 2-3% of the records are already worse than a table scan

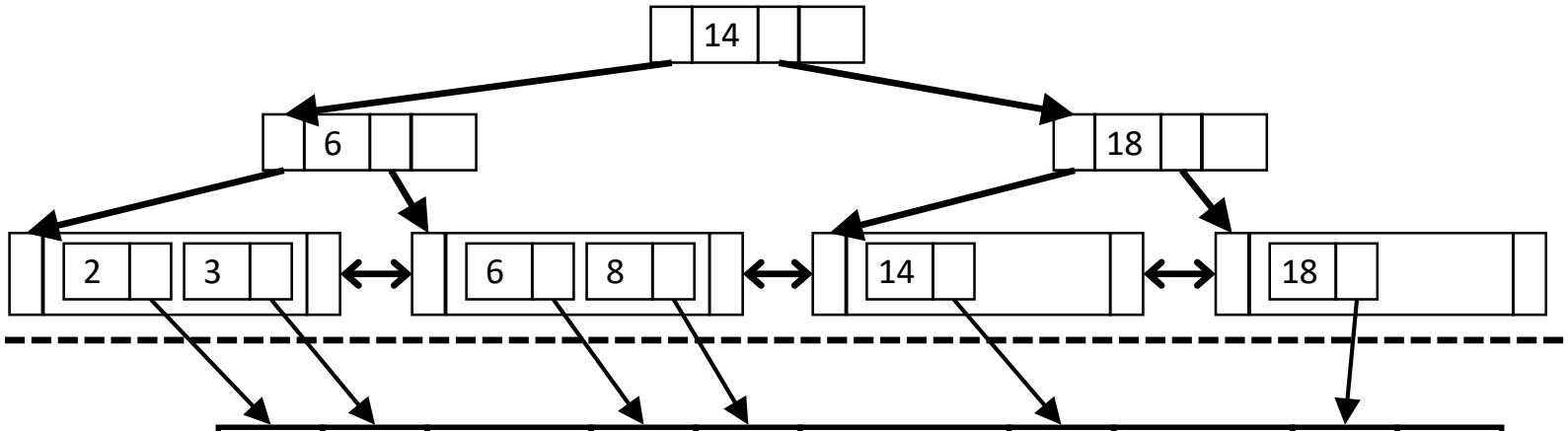
# Clustered Vs. Unclustered B-Trees

- A clustered B-tree, also known as index cluster, is a B-tree whose table file order is the same as the order of the B-tree leafs. Thus, it is a kind of B-tree that imposes changes in the table file
  - Accordingly, one table may have several unclustered B-trees but only one clustered B-tree
- To build an index cluster, you can follow the same batch algorithm presented for B-trees but additionally sorting the table file (with some extra space for insertions)

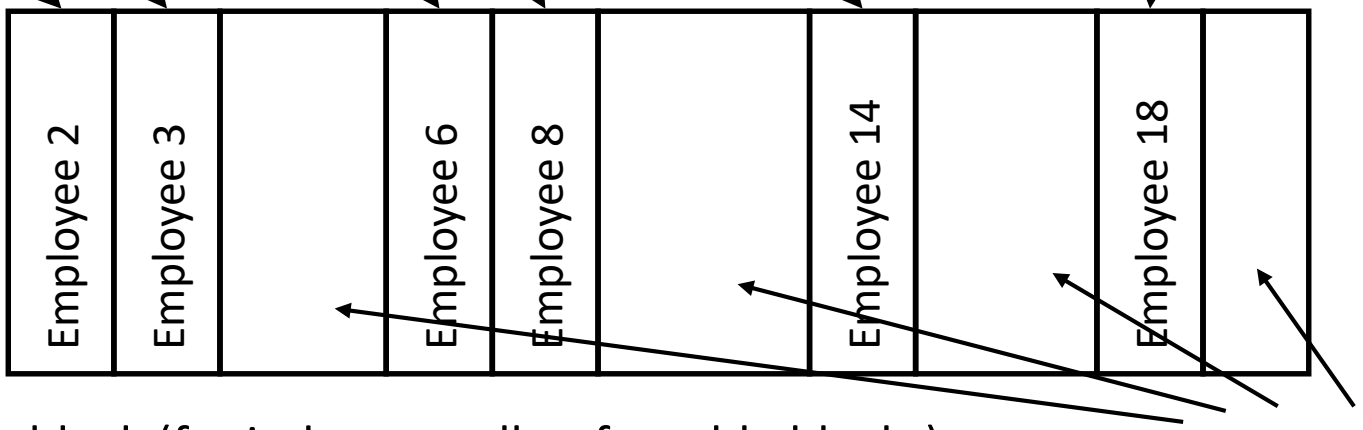


# Clustered B-Tree (Index Cluster)

**B-tree:**




**Records in the table file:**



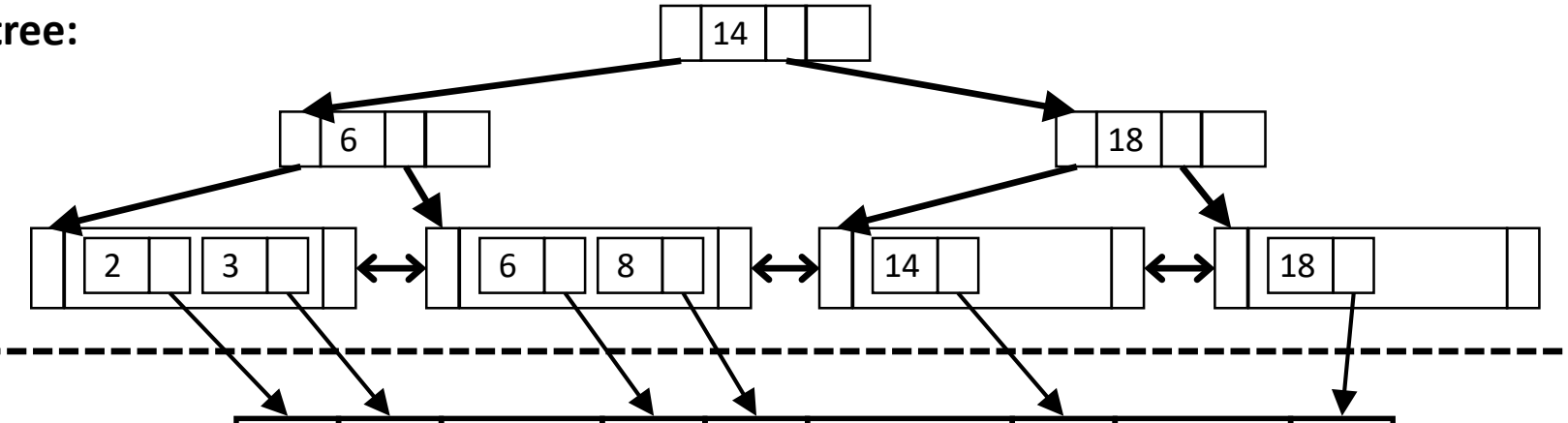

Since the table is sorted according to the tree data entries, we keep some extra empty space to facilitate inserting data without having to relocate older records.

Assumptions:

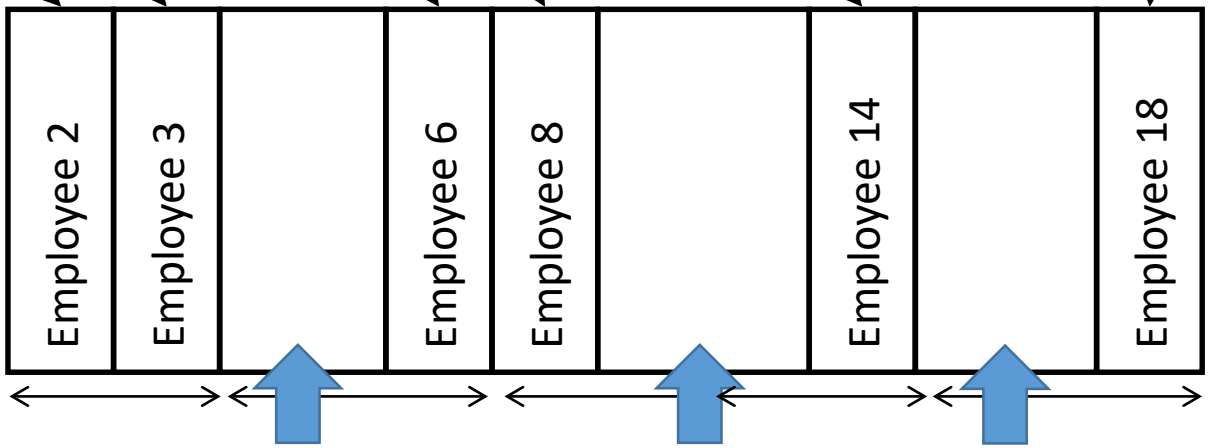
Only 2/3 are used in every block (for index as well as for table blocks)

# Clustered B-Tree (Index Cluster)

**B-tree:**




**Records in the table file:**




**Query**. Read records where A > 8

#Table blocks read with table scan? 5 (#B)

#Table blocks read using index cluster? 2

Note the index cluster is pretty efficient for ranges. Once it finds the first element > 8 in the tree leaves (i.e., 14), it accesses the table and **continues from there** scanning the following blocks (taking advantage of the table sorting)

When performing table scan, the empty spaces are extra space read!

# Multi-Attribute Vs. Single-Attribute B-Tree

- A multi-attribute tree contains several values in its data entries (leaves)
  - Instead of [value, RID] pairs it uses [{$value_1$, …, $value_N$}, RID]
- Consider the following example:

| | idPerson | Name | Surname | Address | DoB | Company | Dept. | Salary |
|---|---|---|---|---|---|---|---|---|
| @1 | 1 | Joan | Gamper | Rue d'Avenir | 1/1/1956 | FCB | Sales | 100000 |
| | … | | | | | | | |
| @8 | 8 | Mercè | Blanc | c/ Aragó | 5/5/1976 | Heart | Marketing | 120000 |

- Consider a multi-attribute B-Tree (company, dept, salary). The data entries generated would be:

```
[{FCB, Sales, 100000}, @1], ..., [{Heart, Marketing, 120000}, @8]
```

# Multi-Attribute Vs. Single-Attribute B-Tree

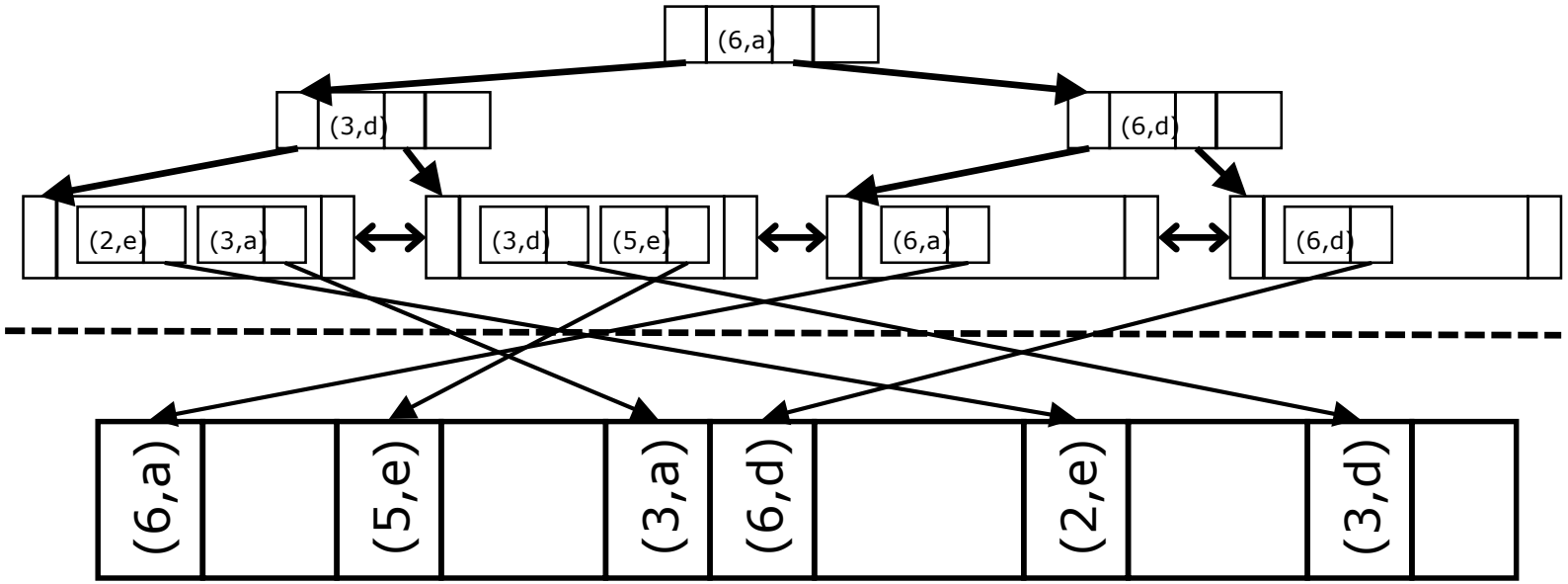
How to explore a multi-Attribute B-tree?

- The only difference is on how to perform the data entries comparison
- The comparison of data entries is done in order, starting from the first value and looking for the first inequality. Consider two data entries of size N: $[\{v_1, v_2, \ldots, v_N\}, @z]$, $[\{v_{1'}, v_{2'}, \ldots, v_{N'}\}, @w]$ then:
  - $[\{v_1, v_2, \ldots, v_N\}, @z] > [\{v_{1'}, v_{2'}, \ldots, v_{N'}\}, @w]$ if $((v_1 > v_{1'})$ OR $(v_1 == v_{1'}$ AND $v_2 > v_{2'})$ OR $(v_1 == v_{1'}$ AND $v_2 == v_{2'}$ AND $v_3 > v_{3'})$ OR ... OR $(\forall i\ i<N, v_i == v_{i'}$ AND $v_N > v_{N'}))$
  - They will be considered the same data entry only if $(\forall i\ i \leq N, v_i == v_{i'})$

- Accordingly, the order of the attributes in the data entry do matter
  - A multi-attribute index can be used as a regular B-tree for the first element
  - For complex selection predicates, the multi-attribute index might not be useful

# Multi-Attribute Tree

Consider the following query predicates. Would this index [{Num, Let}, @] be useful to access the required data?




Queries:

- Num='3' AND Let='d'
- Num='3' AND Let>'b'
- Num='3'
- Num>'3' AND Let='a'
- Num>'3' AND Let>'b'
- Num>'3'
- Let='e'
- Let>'b'
- Num='3' OR Let='a'

# Usefulness of Multi-Attribute B-Trees

- Need more space
  - For each data entry it stores k values $V_1$, .., $V_k$
  - May result in more levels, worsening access time
- Modifications are more frequent
  - Every time one of the attributes in the index is modified
- For multi-predicate selections, it is much more efficient than intersecting RID lists (to evaluate conjunctions)
- Can be used to solve several kinds of queries
  - Equality of all first *i* attributes
  - Equality of all first *i* attributes and range of the *i+1*
- The order of attributes in the index matters
  - We cannot evaluate condition over $A_k$, if there is no equality for $A_1$, .., $A_{k-1}$

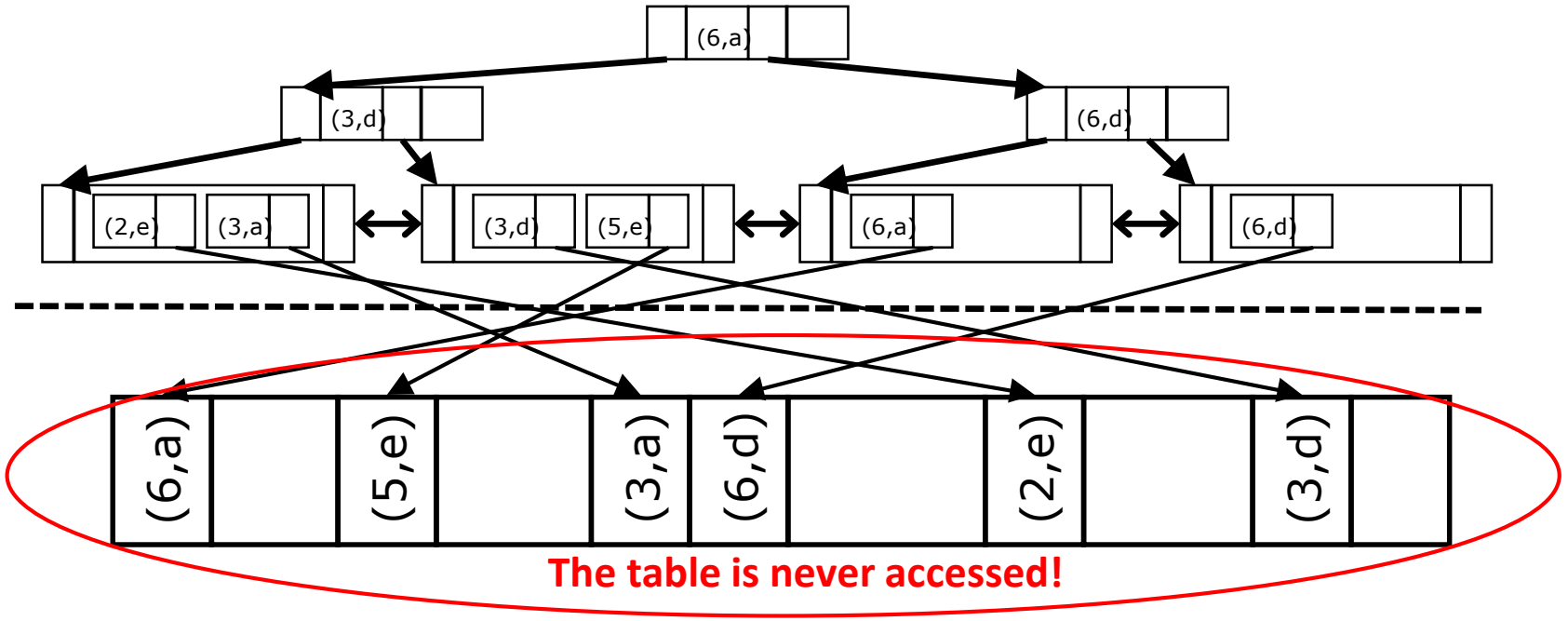
# Index-Only Query Answering

- We have discussed sequential and random access, but multi-attribute B-trees facilitate a third kind of access patern: **index-range access**
  - Note that index-only query answering may happen with just single-attribute B-trees
- Index-range access scans the B-tree leaves to answer the query (i.e., it does not access the B-tree as usual but avoids accessing the table file at all)
  - For this reason, this kind of access is also known as index-only query answering (IOQA)
- We may use IOQA depending on the SELECT clause of the query at hand:
  - All projected attributes in the query are contained in the index
  - All attributes participating in selections are contained in the index
  - DISTINCT and aggregates (count, min, max, avg, etc.) are allowed
  - If the query contains a natural join (e.g., employee.idcard = department.boss), both attributes must be indexed (in different indexes)
  - If the query contains a GROUP BY / ORDER BY clause, all attributes involve must be indexed

# Index-Only Query Answering

- Example:

**Query:**
SELECT COUNT(*)
FROM T
WHERE A > 6 AND B<>'d'



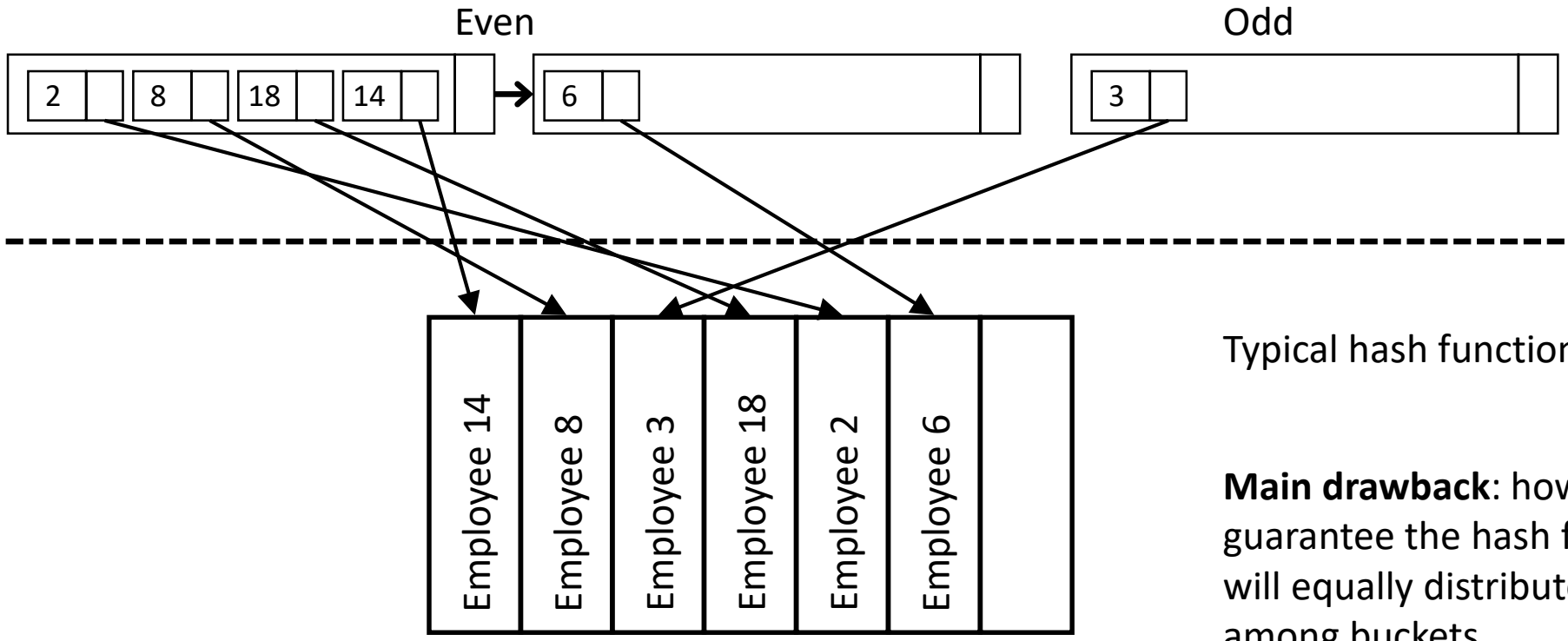

This query can be answered without accessing the table. The DB would use the B-tree as usual for the A>6 subpredicate. Then, from there on, it would scan the rest of leaves and apply the rest of the predicate on them. Finally, the count is performed.

#pages: 2 leaves
(Indexes are typically in memory)
(We assume a page has the same size as a block)

**What changes if the selection predicate was a disjunction?**

# Hash Index (Hash)

**Hash Index:**




**Records in the table file:**

Typical hash function: %

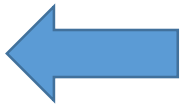
**Main drawback**: how to guarantee the hash function will equally distribute records among buckets

Assumptions:

There are no blocks for excess
In a bucket block the same number of entries fit as in a tree block
Usually, bucket blocks are used at 80% (4/5 of their capacity)

# Indexes Must Be Defined If…

- B-tree:
  - There is a very selective condition
  - The predicate is a range
  - Order of the results is relevant (e.g., GROUP BY, ORDER BY)
- Hash:
  - There is a very selective condition (with equality)
  - The table is not very volatile
  - The table is huge
- Clustered:
  - All benefits of B+tree (ranges, ordered results)
  - Benefits of sequential access for range queries
  - The table is not very volatile (managing the extra empty space is troublesome)



Initially, most scalable systems use(d) hashing for fast data accessing (although recently there are more systems implementing an index cluster since finding a proper hash function balancing data is not trivial)

# We Should NOT Define an Index If…

- Processing is massive
  - Batch processes reading large portions of data (tables) are not well fit for indexes
- The table has few blocks
  - The overhead of the indexing structure does not pay off
- The attribute appears in the predicate inside a function
- The attribute has few values
  - Small selective conditions