Introduction to Database Systems IDBS – Spring 2024

- Week 7:
- Storage Hierarchy
- Physical Database Design
- Indexing

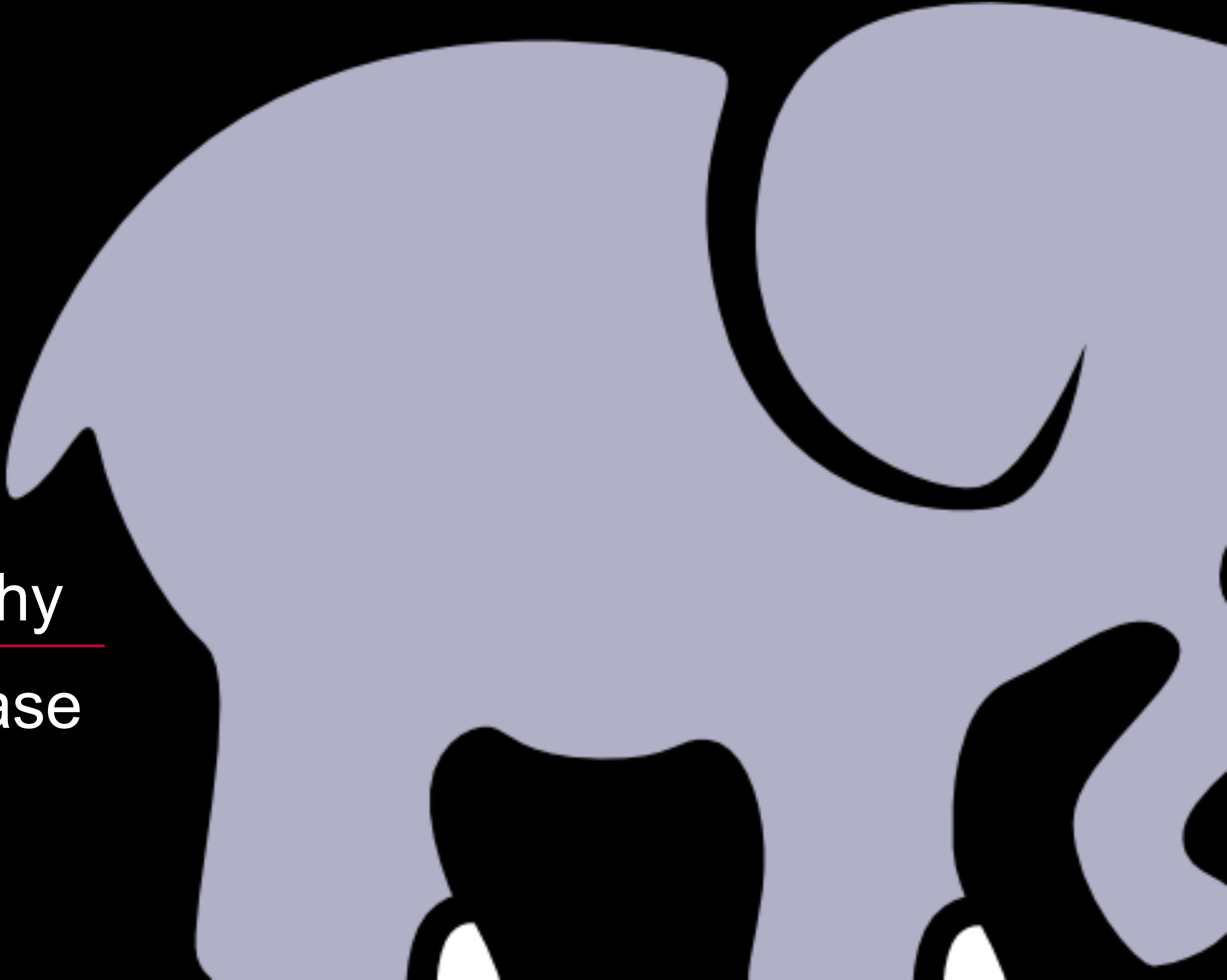
Eleni Tzirita Zacharatou

Readings:

PDBM 12.1, 12.2, 12.3.1 - 12.3.7

Storage Hierarchy

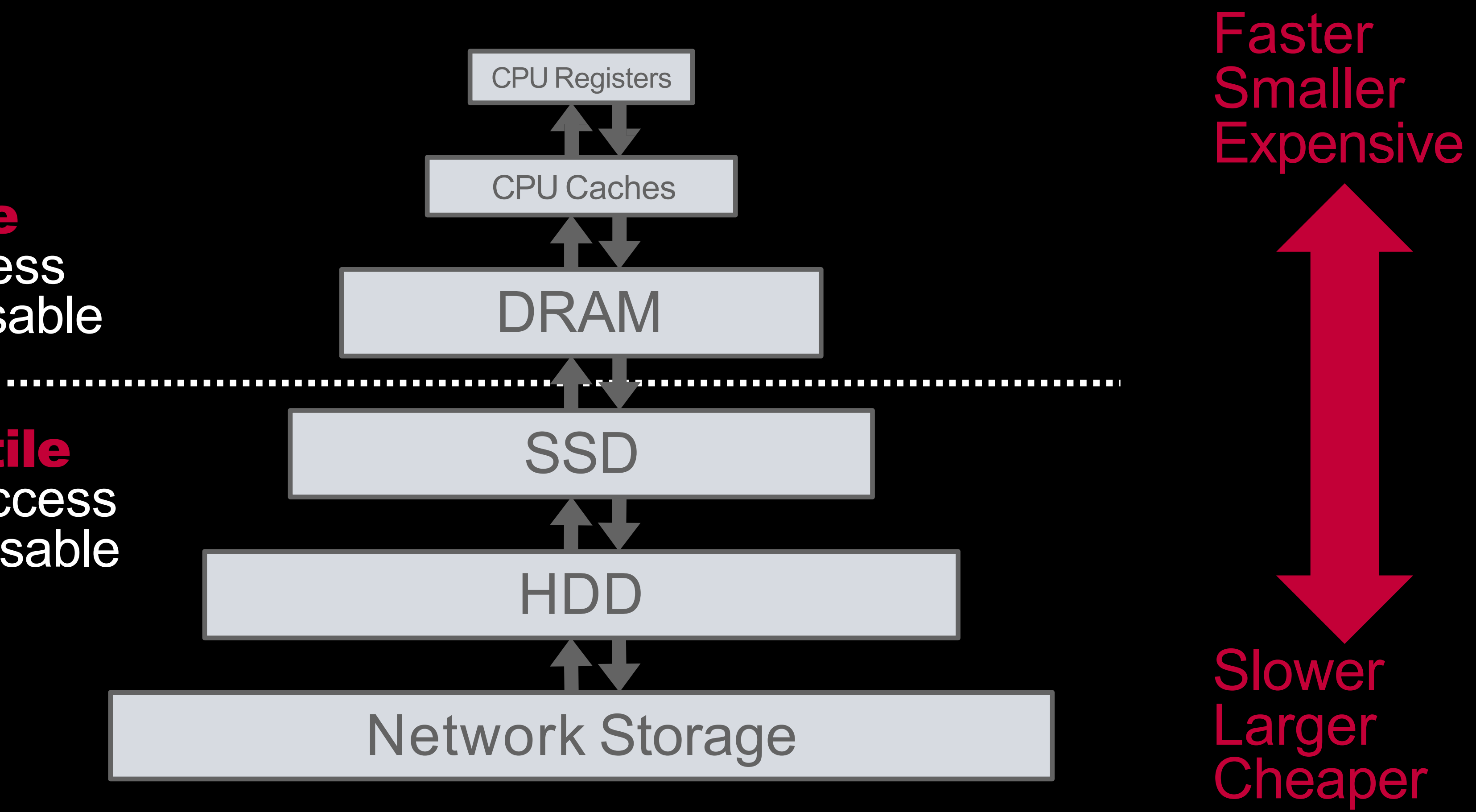
Physical Database Design



Storage Hierarchy

Volatile
Random Access
Byte-Addressable

Non-Volatile
Sequential Access
Block-Addressable



Access Times

1 ns	L1 Cache Ref	⬅	1 sec
4 ns	L2 Cache Ref	⬅	4 sec
100 ns	DRAM	⬅	100 sec
16,000 ns	SSD	⬅	4.4 hours
2,000,000 ns	HDD	⬅	3.3 weeks
~50,000,000 ns	Network Storage	⬅	1.5 years
1,000,000,000 ns	Tape Archives	⬅	31.7 years

Source: Colin Scott

Disk-Based Architecture

- The DBMS assumes that the primary storage location of the database is on non-volatile disk.
- The DBMS's components manage the movement of data between non-volatile and volatile storage.

Sequential vs. Random Access

- Random access on non-volatile storage is almost always much slower than sequential access.

- DBMS will want to maximize sequential access.
  - Algorithms try to reduce number of writes to random pages so that data is stored in contiguous blocks.

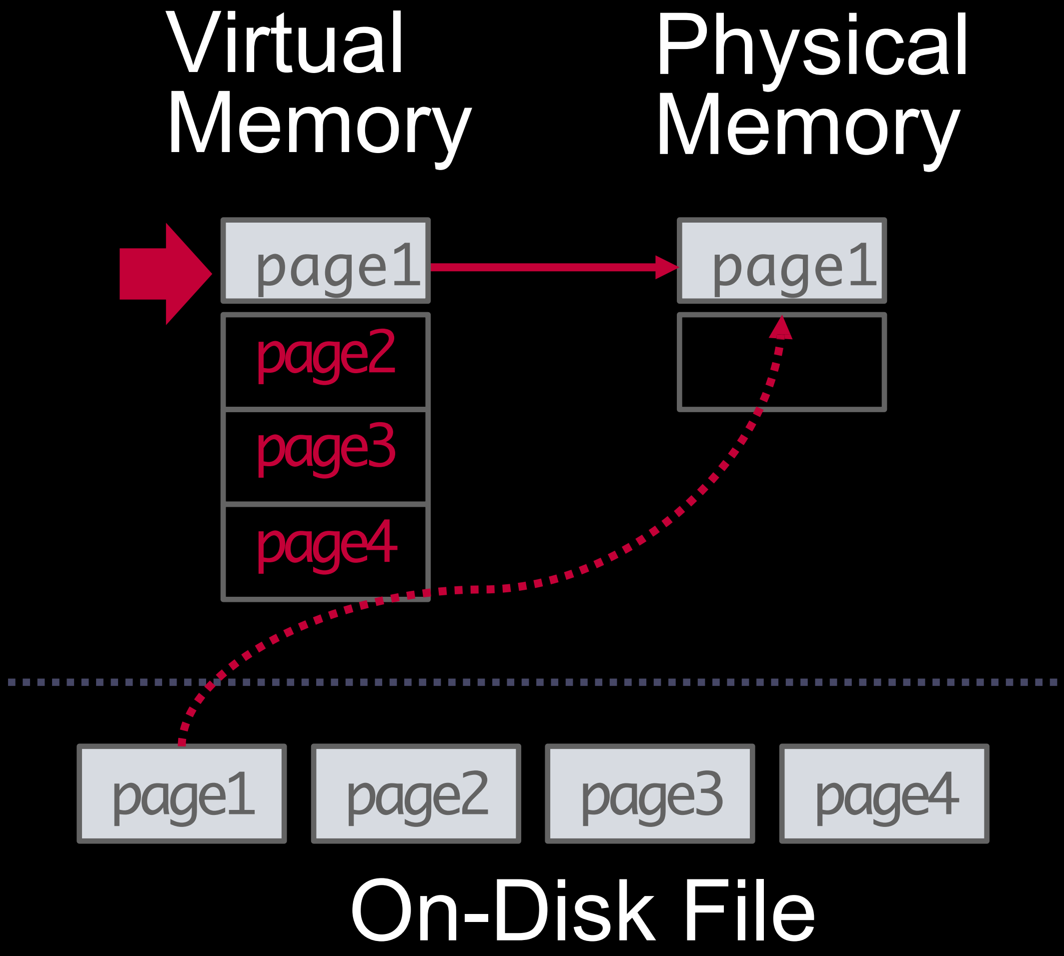
System Design Goals

- Allow the DBMS to manage databases that exceed the amount of memory available.
- Reading/writing to disk is expensive, so it must be managed carefully to avoid large stalls and performance degradation.
- Random access on disk is usually much slower than sequential access, so the DBMS will want to maximize sequential access.

Why Not Use the OS?

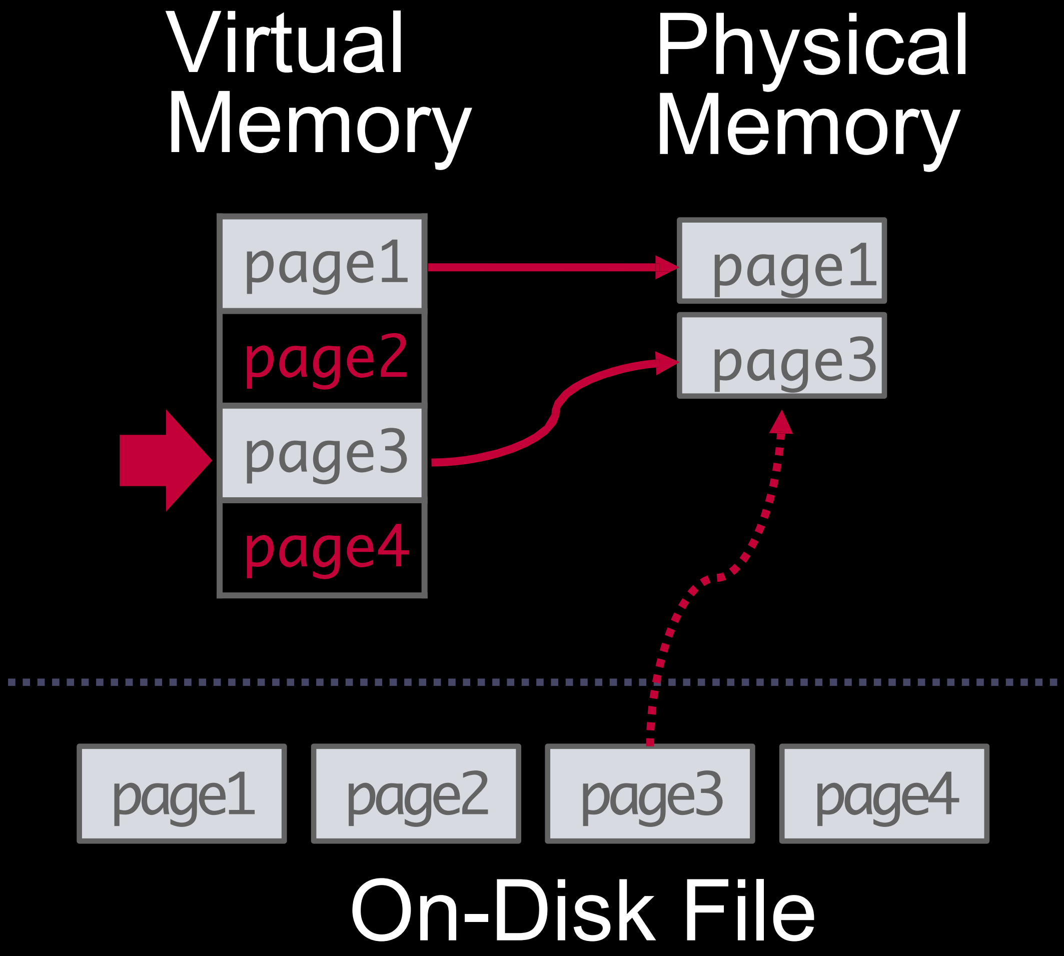
- The DBMS can use memory mapping (*mmap*) to store the contents of a file into the address space of a program.

- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



Why Not Use the OS?

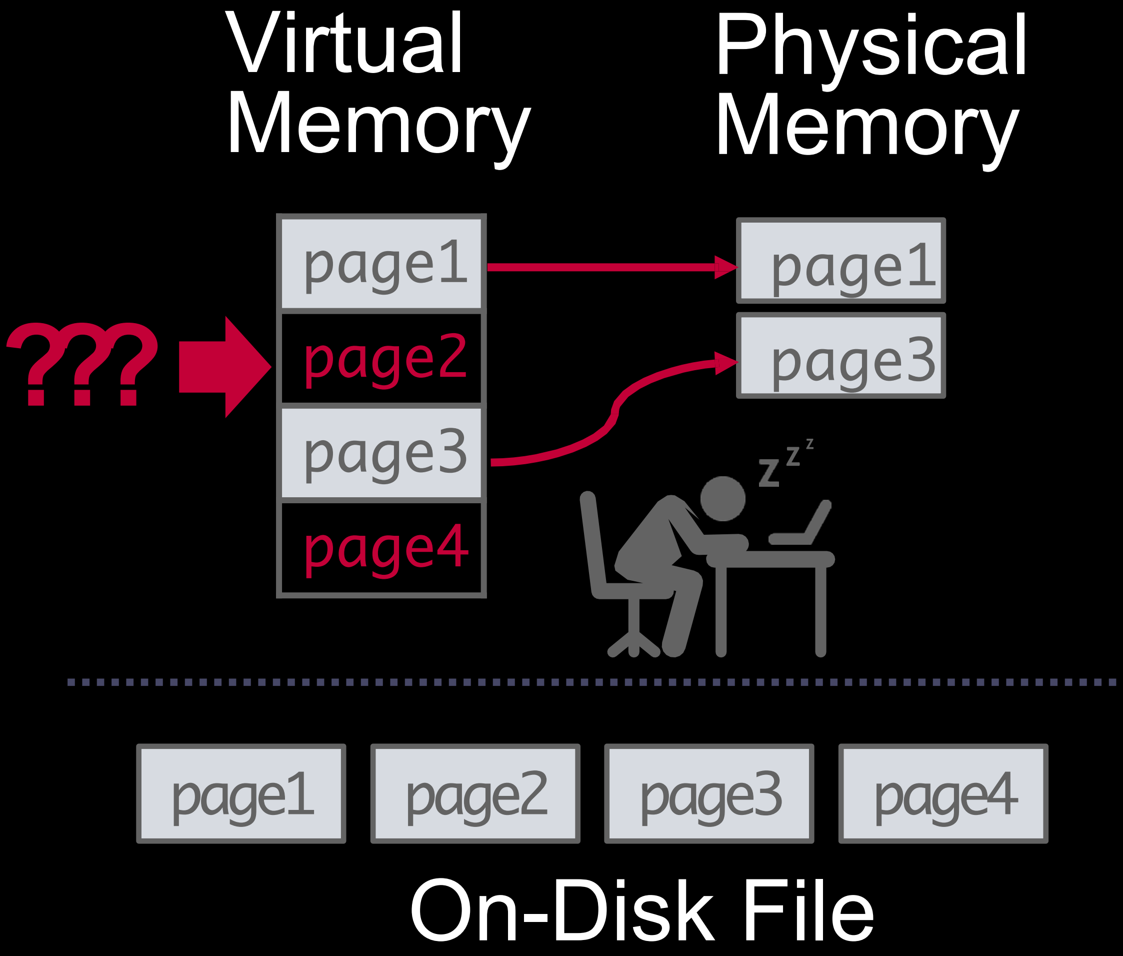
- The DBMS can use memory mapping (*mmap*) to store the contents of a file into the address space of a program.
- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



Why Not Use the OS?

- The DBMS can use memory mapping (*mmap*) to store the contents of a file into the address space of a program.

- OS is responsible for moving file pages in and out of memory, so the DBMS doesn't need to worry about it.



Memory Mapped I/O Problems

Problem #1: Transaction Safety

→ OS can flush dirty pages at any time.

Problem #2: I/O Stalls

→ DBMS doesn't know which pages are in memory. The OS will stall a thread on page fault.

Problem #3: Error Handling

→ Difficult to validate pages. Any access can cause a **SIGBUS** that the DBMS must handle.

Problem #4: Performance Issues

→ OS data structure contention. TLB shootdowns.

Why Not Use the OS?

- DBMS (almost) always wants to control things itself and can do a better job than the OS.
  → Flushing dirty pages to disk in the correct order.
  → Specialized prefetching.
  → Buffer replacement policy.
  → Thread/process scheduling.

- The OS is *not* your friend.

Storage Manager

- The DBMS stores a database as one or more files on disk typically in a proprietary format.

- The storage manager is responsible for maintaining the files of a database.

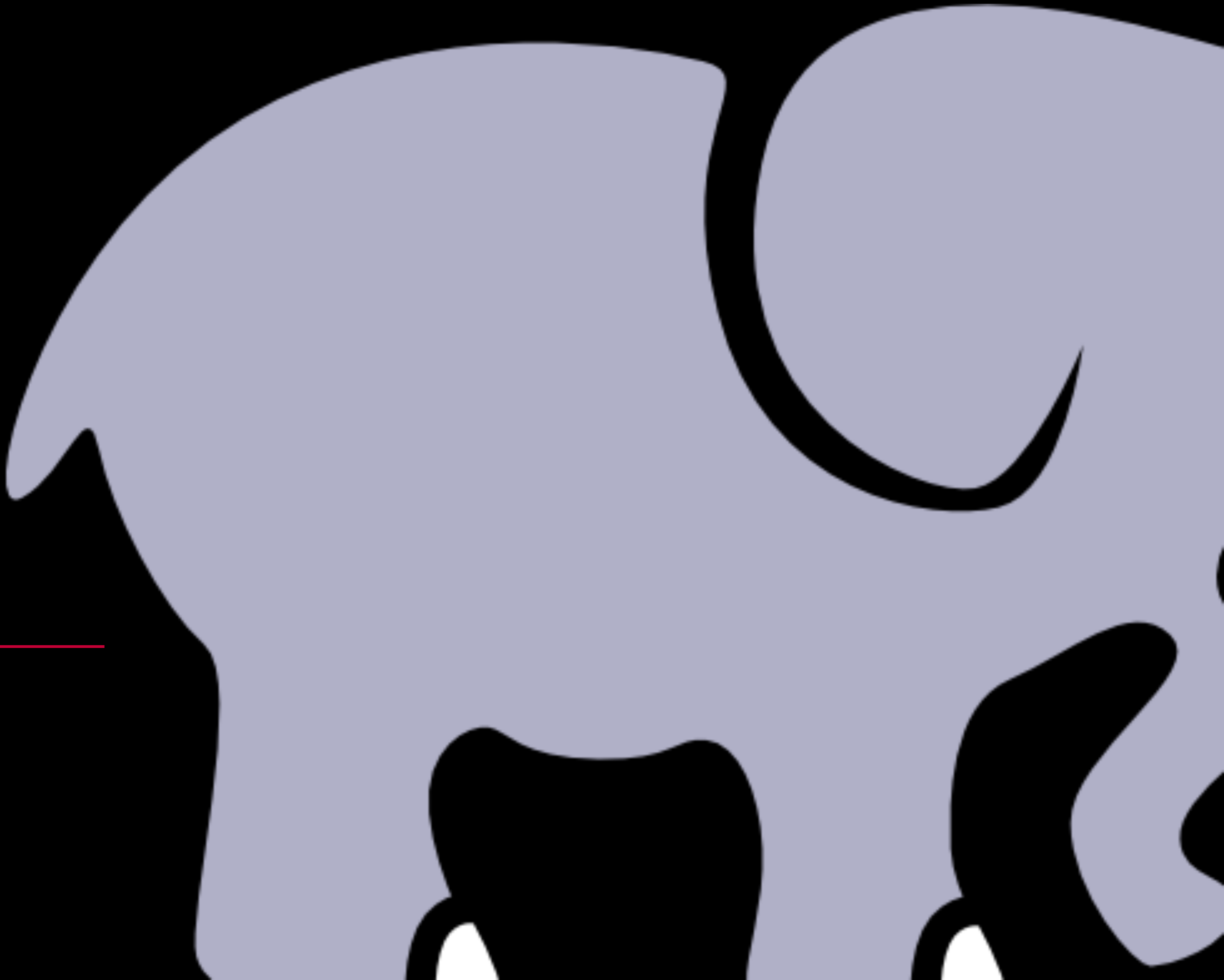
- It organizes the files as a collection of *pages*.
  - Tracks data read/written to pages.
  - Tracks the available space.

Database Pages

- A page is a fixed-size block of data.
  - It can contain tuples, meta-data, indexes, log records…
  - Most systems do not mix page types.
  - Some systems require a page to be self-contained.

- Each page is given a unique identifier.
  - The DBMS uses an indirection layer to map page IDs to physical locations.

Indexing



Full Table Scans

When a DBMS sees a query of the form

```
SELECT *
FROM R
WHERE <condition>
```

one obviously possible thing to do is read through **all** the tuples of R and report those tuples that satisfy the condition.

This is called a **full table scan**.

Selective Queries

Consider the query from before:

- If we have to report 80% of the tuples in R, it makes sense to do a full table scan.

- On the other hand, if the query is very **selective**, and returns just a small percentage of the tuples, we might hope to do better.

```
SELECT *
FROM R
WHERE <condition>
```

Point Queries

- Consider a selection query with a single equality condition:

```
SELECT *
FROM person
WHERE birthdate='1975-02-06';
```

- This is a *point query*: We look for a single value of birthdate.
  - We may still return > 1 record!

- Point queries are easy if data is sorted by the attribute used in the condition.
  - How? What algorithms would work?

Range Queries

- Consider a selection query of the form:

```
SELECT *
FROM person
WHERE birthdate BETWEEN
'1975-02-01' and '1975-02-28’;
```

- This is a *range query*: We look for a range of values of `birthdate`.

- Range queries are also easy if data is sorted by the right attribute.
  - But often not as **selective** as point queries.

Indexes

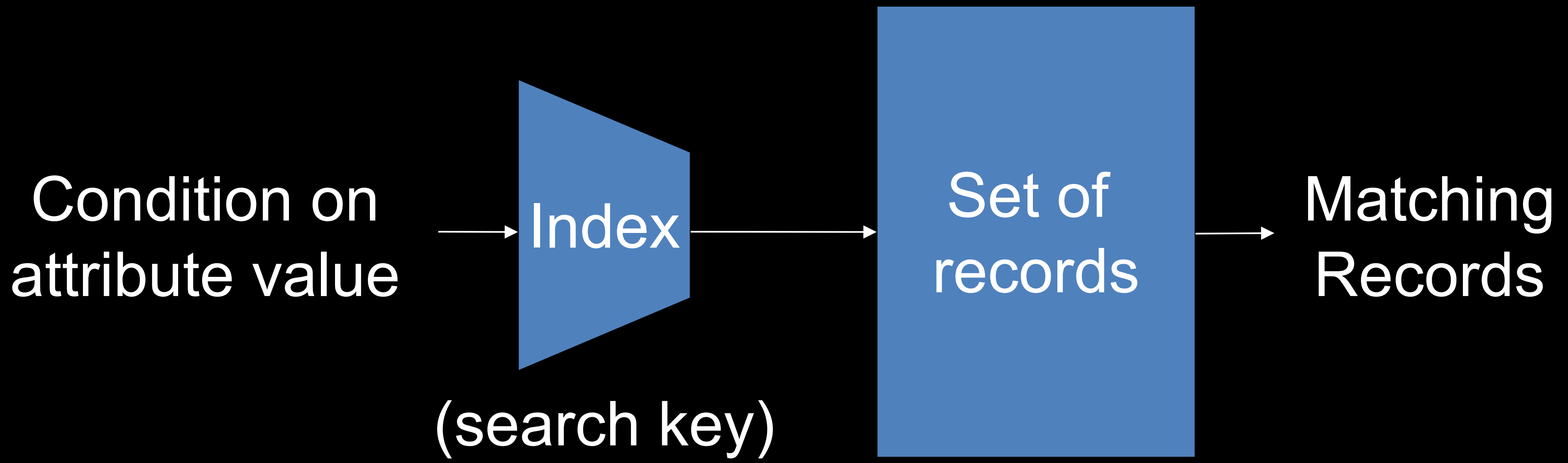
- To speed up queries the DBMS may build an index on the `birthdate` attribute.

- A database index is similar to an index in the back of a book:
  - For every piece of data you might be interested in (e.g., the attribute value `1975-02-06`), the index says where to find the row with the actual data!
  - The index itself is organized such that one can quickly do the lookup.

- Looking for information in a relation with the help of an index is called an **index scan** (range) or **index lookup** (point).

Indexing

- An index is a data structure that supports efficient access to data
  - In databases, indexes are also stored on disk



Two Techniques

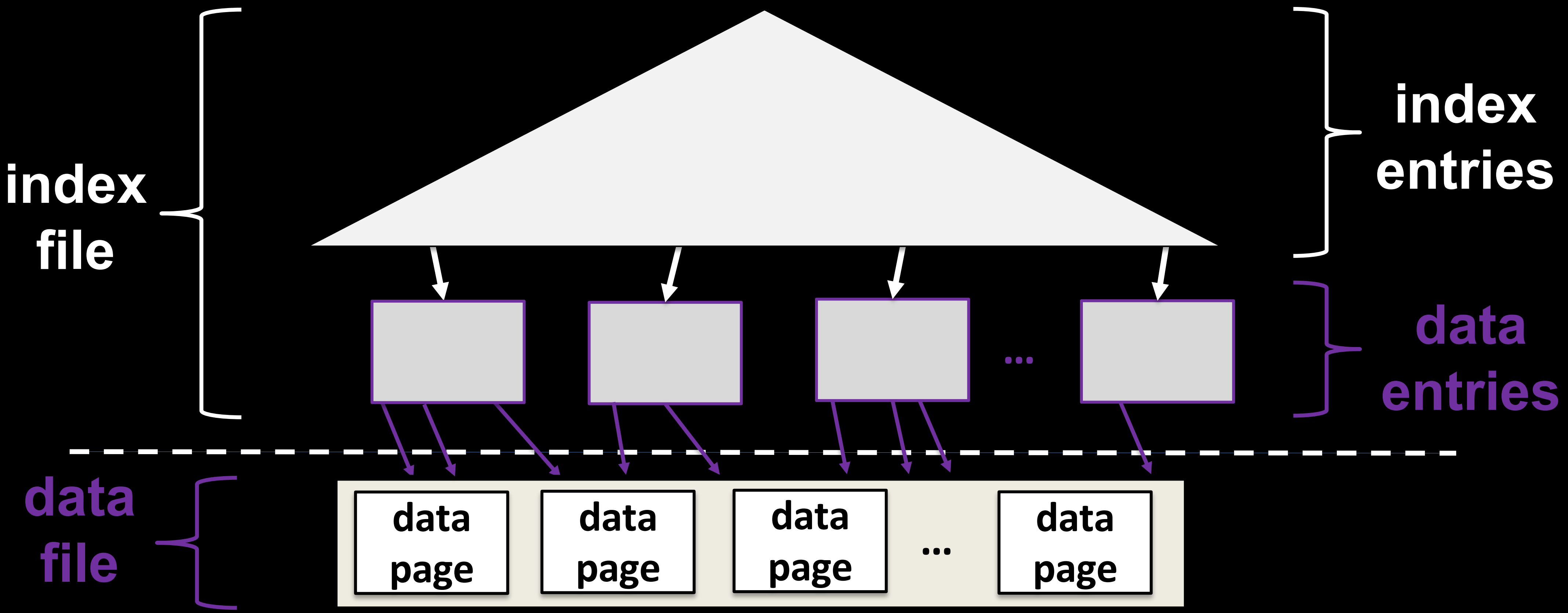
- **Hashing:** Use a fixed transformation algorithm to convert the attribute value into a database address.

- **Tree search**: A dynamic search structure is built that guides the search for a given attribute value to the proper database location.

- Hashing supports equality queries only
  - Typically used dynamically for large-scale joins
  - Rarely available to developers

- Tree search is more versatile and accessible

Clustered Index



order of data entries matches order of data records

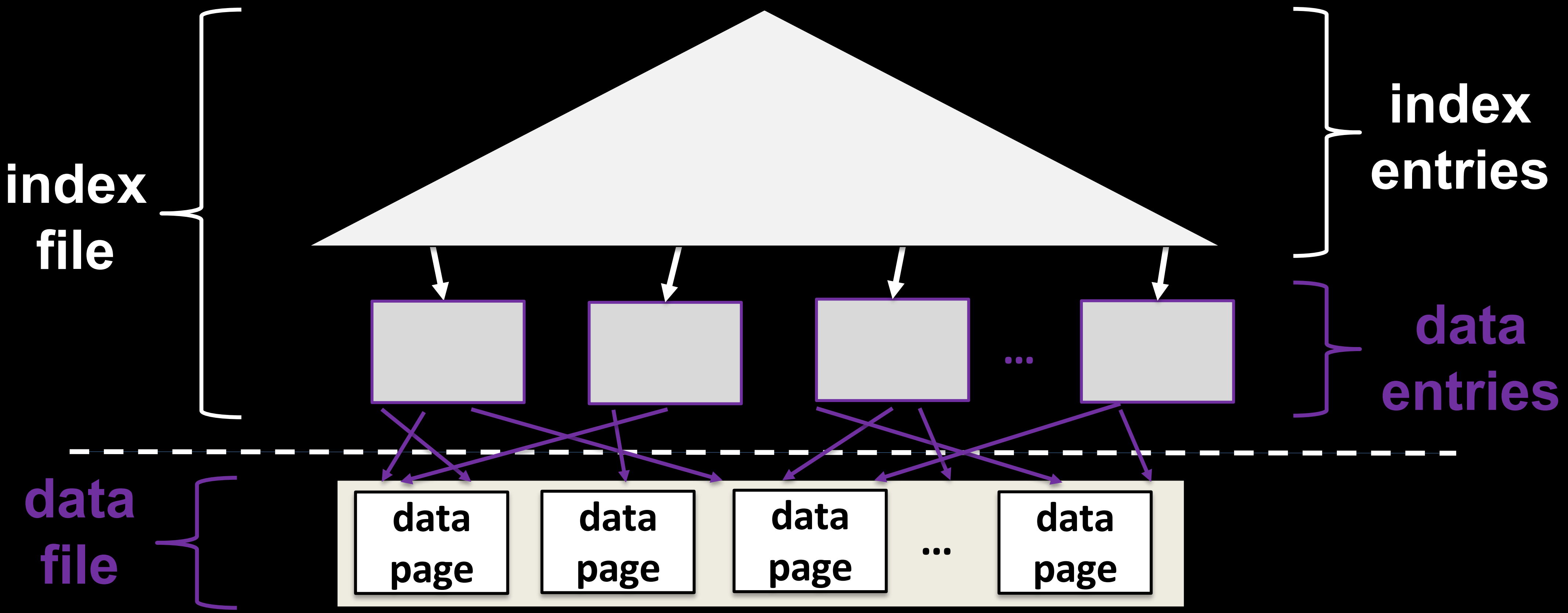
Clustered Index

- If the tuples of a relation are stored sorted according to some attribute, an index on this attribute is called clustered.
  - Clustered indexes make point and range queries on the key *very efficient*
  - Why? Sequential reads + As few reads as possible!

- Many DBMSs automatically build a clustered index on the primary key of each relation.

- A clustered index is sometimes referred to as a primary or sparse index.
  - There can only be one clustered index! Why?

Unclustered Index



order of data entries != order of data records

Unclustered Index

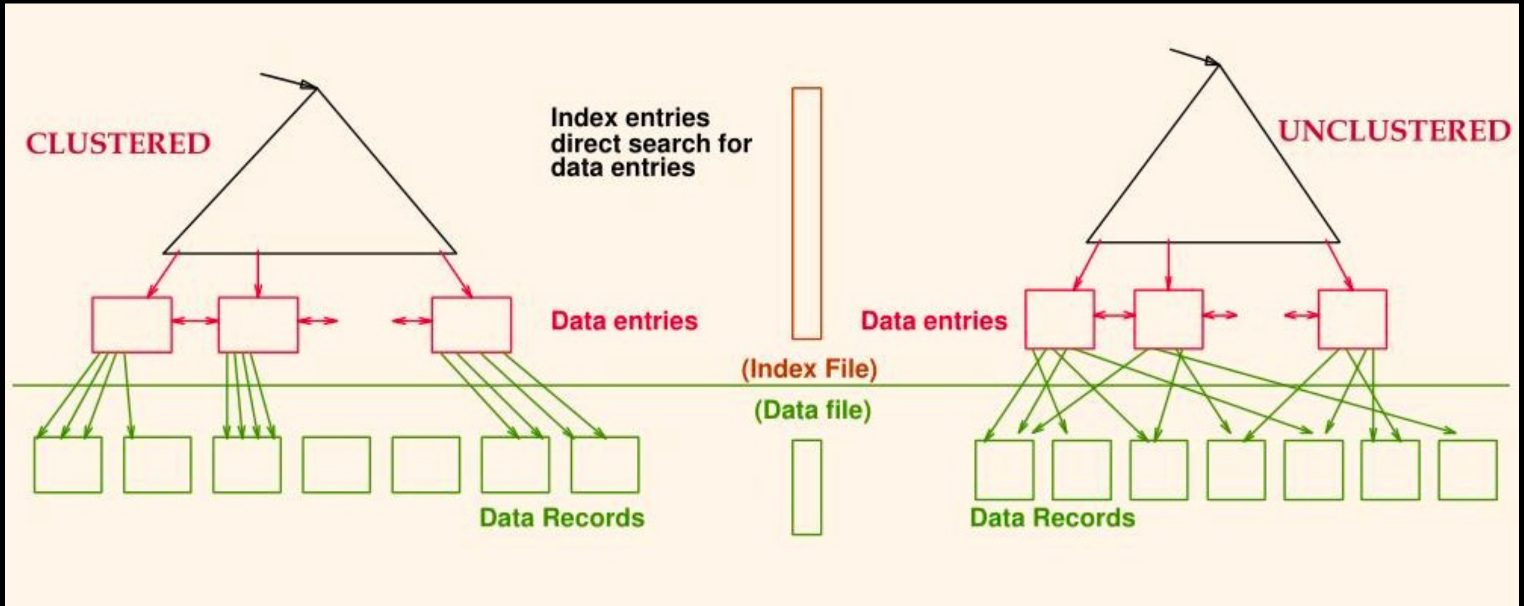
- It is possible to create further indexes on a relation. Typical syntax:

```
CREATE INDEX myIndex ON involved(actorId);
```

- The unclustered indexes are sometimes called non-clustered, secondary or dense indexes.

- Unclustered indexes:
  - Make most point queries more efficient.
  - Make some (narrow) range queries more efficient.

Clustered vs. Unclustered Index



- To retrieve *M* records, where *M* is small:
  - Clustered: Probably one disk read
  - Unclustered: Probably *M* random disk reads
- To retrieve *M* records, where *M* is large:
  - Clustered: Probably *M*/*records_per_page* sequential disk reads
  - Unclustered: Up to *M* random disk reads
- We still need to read the index itself – same for both!

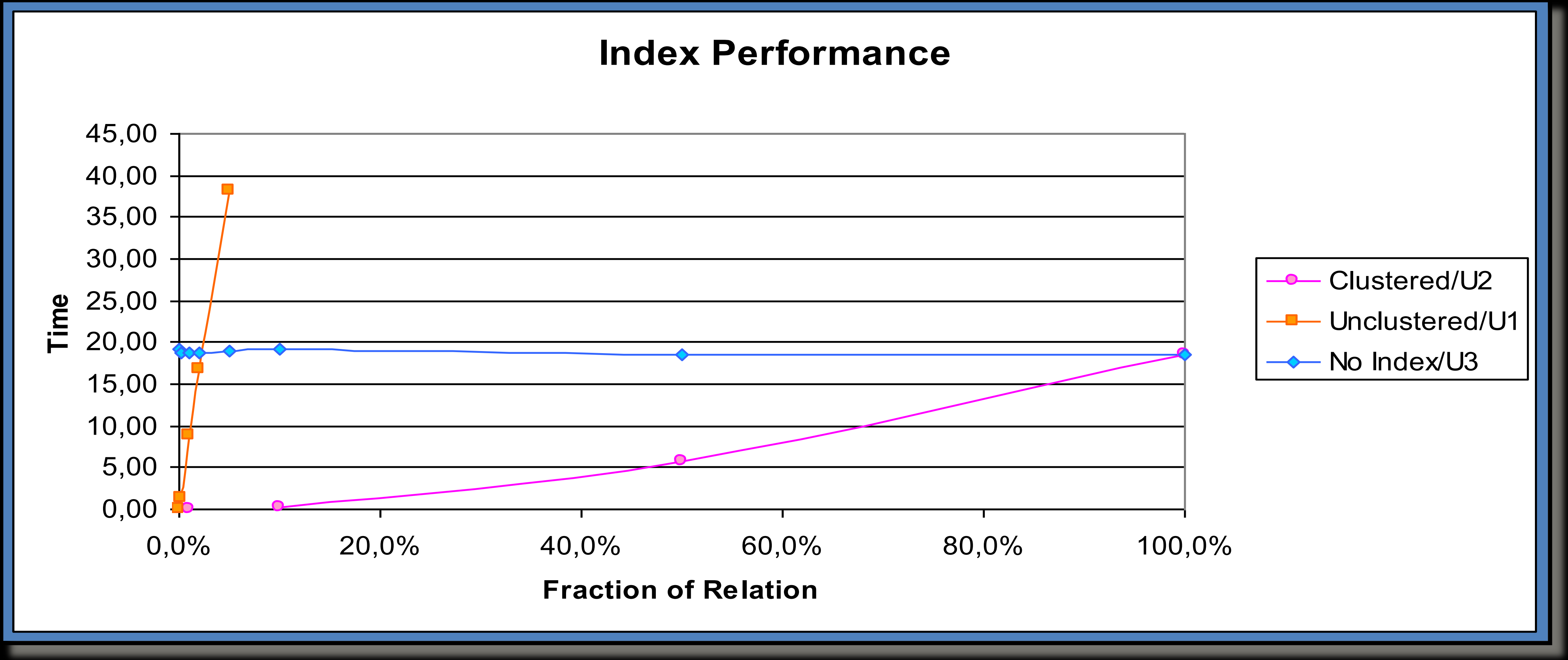
Index Scan vs. Full Table Scan

- Point and range queries on the attribute(s) of the clustered index are almost always best performed using an index scan.

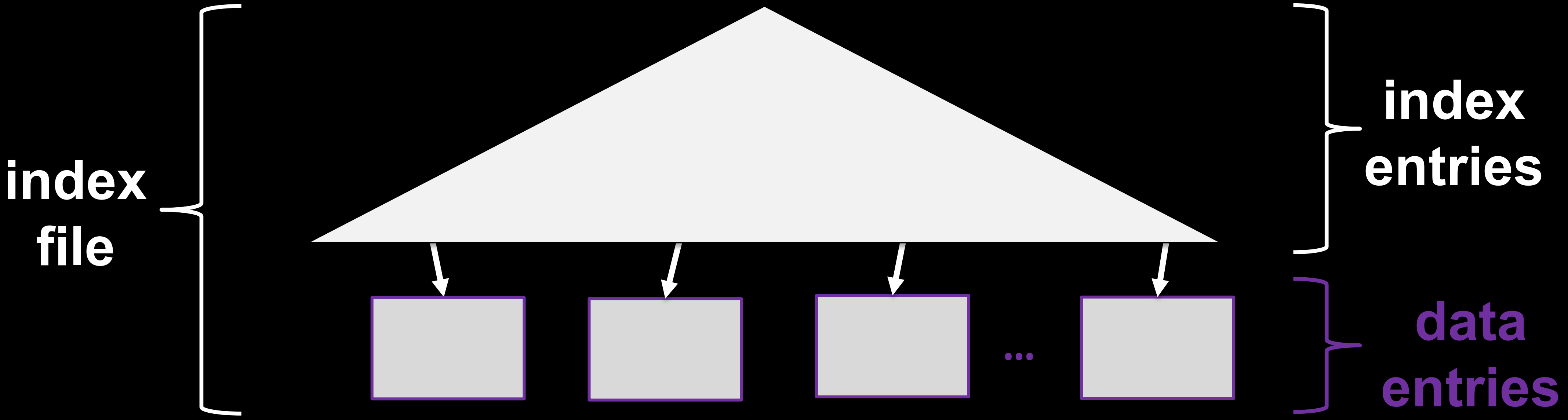
- Non-clustered indexes should only be used with high selectivity queries.

- Rule of thumb?

Impact of Clustering on Performance



Covering Index



data entries contain ALL queried attributes

Covering Index

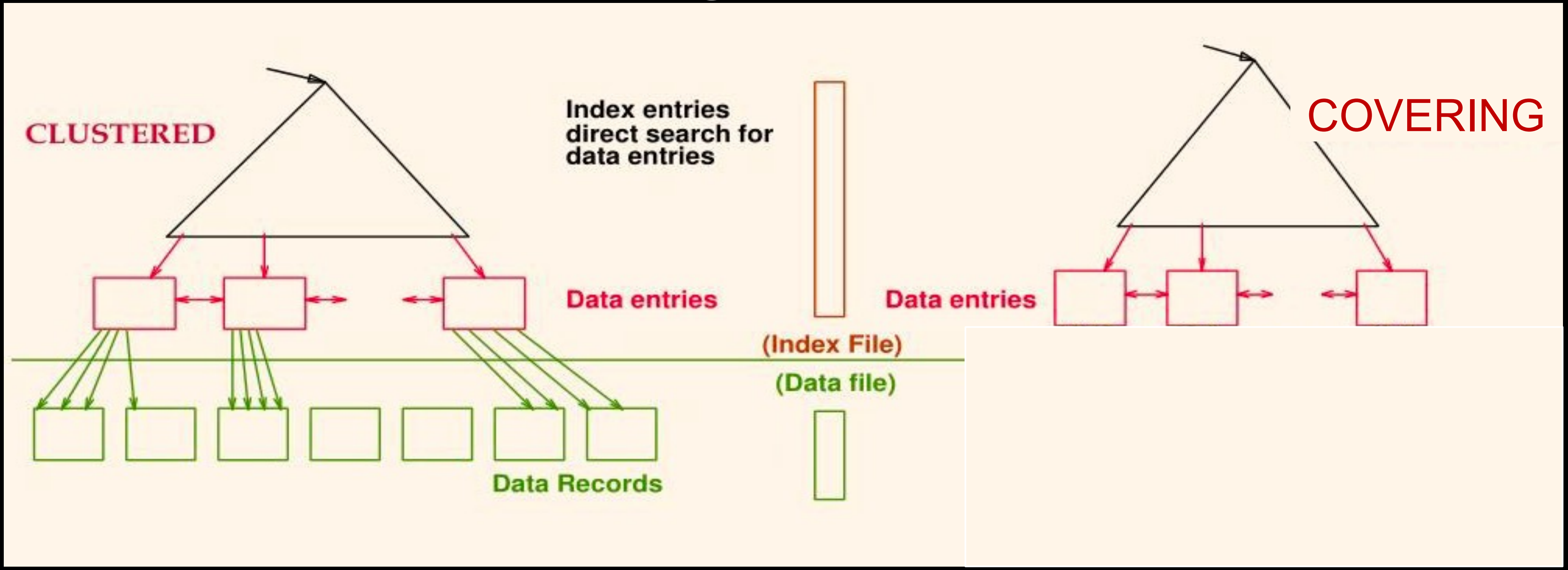
- An index that contains ALL attributes used in a query is called covering
  - The resulting query plans are index-only

```
SELECT COUNT(*) FROM movie WHERE year=1948;
CREATE INDEX movieyear ON movie(year);

SELECT name FROM person WHERE height=170;
CREATE INDEX phn ON person(height, name);
```

- The data from the relation is not needed = no disk reads required to retrieve tuples
  - Should a covering index be clustered or non-clustered?

Clustered vs. Covering Index



- To retrieve *M* records, where *M* is small:
  - Clustered: Probably one disk read
  - Covering: Definitely need 0 disk reads
- To retrieve *M* records, where *M* is large:
  - Clustered: Probably *M*/*records_per_page* sequential disk reads
  - Covering: Definitely need 0 disk reads
- We still need to read the index itself – same for both!

Problem Session

- For each of the following queries:
  - Which index would give the best plan?
  - Would the index be covering?
  - Would you prefer clustered or unclustered index?
  - Based on these queries, if you could choose, which index should be clustered?

```
1. SELECT * … WHERE birthdate = '20-02-2002'
2. SELECT * … WHERE height < 170
3. SELECT * … WHERE ID = 4564
4. SELECT AVG(birthdate) …
```

```
CREATE INDEX bIndex
ON person(birthdate);
CREATE INDEX hIndex
ON person(height);
CREATE INDEX iIndex
ON person(ID);
```

Multi-Attribute Indexes

- Defining an index on several attributes:

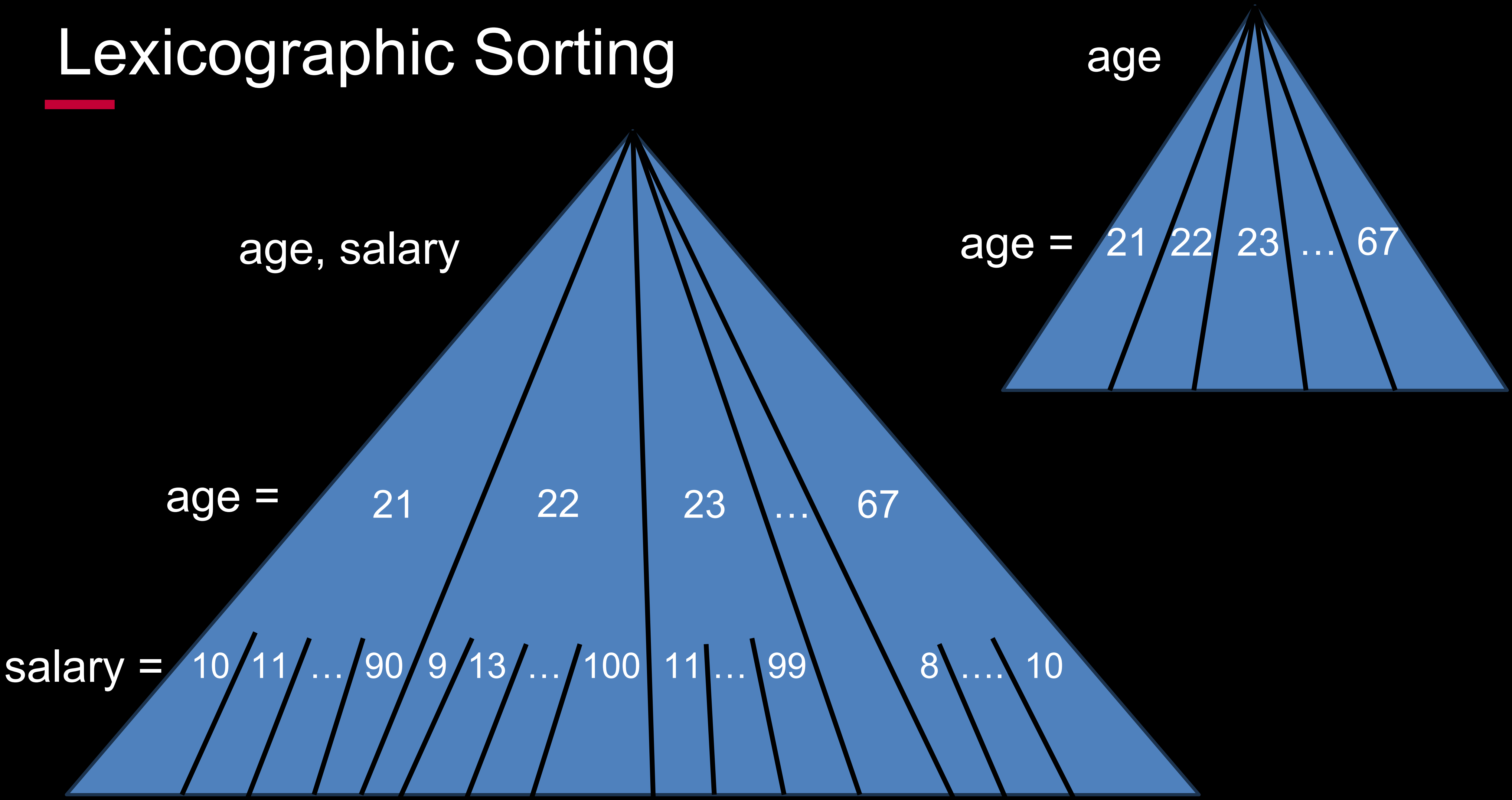
```
CREATE INDEX myIndex
ON person (height,birthdate);
```

- Speeds up point/range queries such as:

```
SELECT *
FROM person
WHERE height=170 and birthdate<'1945-08-08'
```

- An index on several attributes usually gives index for any **prefix** of these attributes, due to lexicographic sorting.

Lexicographic Sorting



Problem Session

- Which point and range queries are ”easy” (equality on a prefix, range on one) when the relation is indexed with this two-attribute index?

```
CREATE INDEX myIndex
ON person (height, birthdate);
```

1. A range query on height?
2. A range query on birthdate?
3. A point query on birthdate?
4. A point query on birthdate combined with a range query on height?
5. A point query on height combined with a range query on birthdate?

Choosing to Use an Index

- The choice of whether to use an index is made by the DBMS *for every instance of a query*
  - May depend on query parameters
  - You do not need to take indexes into account when writing queries
- Estimating selectivity is done using statistics
  - In PostgreSQL, statistics are collected by executing statements such as `ANALYZE table_name;`

Choosing Columns

- Candidates for index search keys
  - Columns in WHERE clauses
  - Columns in GROUP BY clauses
  - Columns in ORDER BY clauses

- Columns that are rarely candidates
  - Large columns (too much space)
  - Frequently updated columns (too much maintenance)
  - Columns in SELECT clauses (not used to find tuples)
    - ... but see **covering indices**!

What Speaks Against Indexing

- Space usage:
  - Similar to size of indexed columns (plus pointer)
  - Most space for leaves, less for tree nodes
  - Not really important!

- Time usage for keeping indexes updated under data insertion/change:
  - Depends on the index architecture
  - This IS important!

Take Away

- Storage hierarchy:
  - Different layers have different characteristics.
  - Sequential access is faster than random access for all layers, but especially for hard disks.
  - DBMSs have various optimizations to minimize data access latency.

- Databases are organized in pages.

- Large databases need to be equipped with suitable indexes
  - Need understanding of what indexes might help a given set of queries.
  - *Key distinction*: Clustered vs. Unclustered.
  - A detailed understanding of various index types is beyond the scope of this course.