COMP9120

Week 11: Storage & Indexing

Semester 1, 2025

Professor Athman Bouguettaya
School of Computer Science

THE UNIVERSITY OF SYDNEY

Contents co-developed with Dr Lijun Chang

Warming up

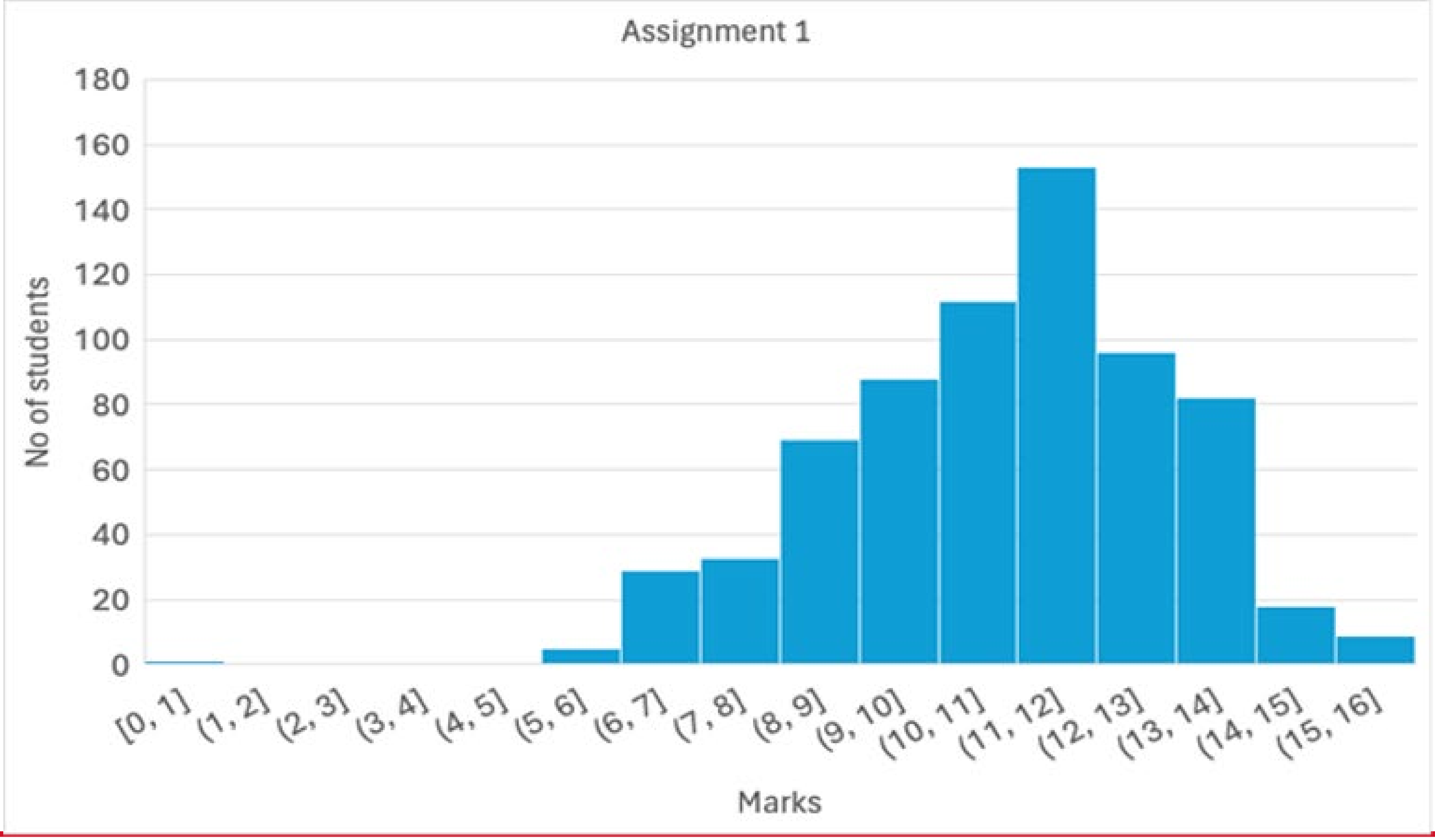
Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

Copyright warning

Assignment 1 statistics

Average: 69%
Median : 71.88%



Outline

- **Physical Data Organization: how data in DBMS is physically stored?**

- **Access Paths: how records are retrieved from DBMS?**
  - Access methods for heap files (linear scan)
  - Access methods for sorted files (binary search)
  - Access methods for indexes (index scan)

- **B+ Tree Index**
  - Primary index
  - Composite search keys

Outline

- **Physical Data Organization: how data in DBMS is *physically stored*?**

- **Access Paths: how records are retrieved from DBMS?**
  - Access methods for heap files (linear scan)
  - Access methods for sorted files (binary search)
  - Access methods for indexes (index scan)

- **B+ Tree Index**
  - Primary index
  - Composite search keys

How is a relation physically stored and accessed?

- Consider the table **Student**

Attributes (also called **columns, fields**)

Tuples (also called **rows, records**)

Student				
sid	name	login	gender	address
5312666	Jones	ajon1121@cs	m	123 Main St
5366668	Smith	smith@mail	m	45 George
5309650	Jin	ojin4536@it	f	19 City Rd

- How are relational tables **physically** stored?
  - Several *organization approaches* for *storing* data
- **How** is data **accessed** and **what** is the **associated cost**?
  - *Access strategies* and *metrics* to execute the query

Data storage

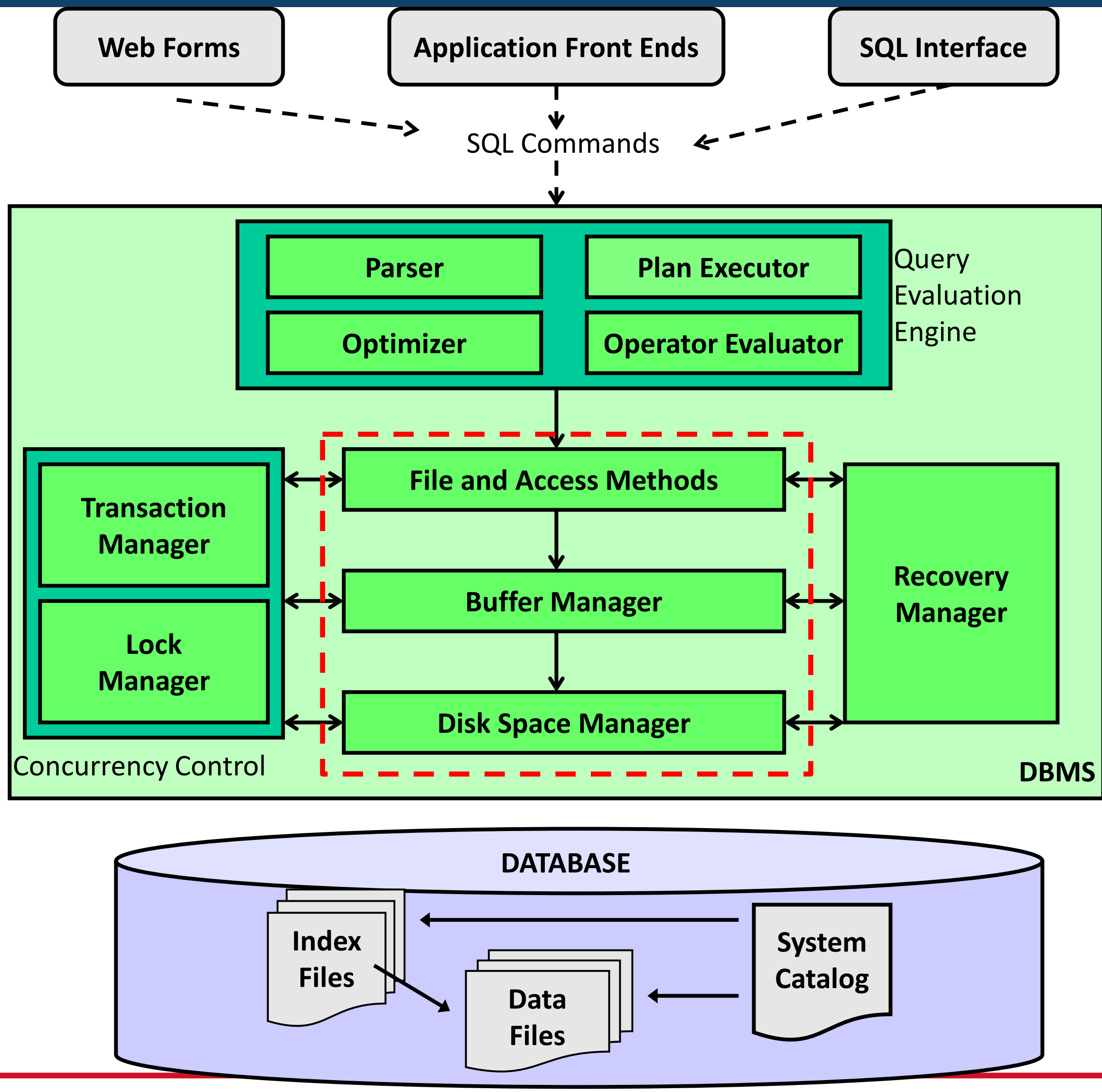
Two **fundamental** DBMS **questions**:

- How do we efficiently **organize** very **large volumes** of data?
- How do we **efficiently access** data, i.e., **minimize I/Os**?

Requirements:

- **Databases** need **persistent storage**
- We **cannot hold all database** content in **main memory**
  - We keep producing **massive** quantities of **data**, i.e., we **always produce more data** than we **can fit** in main **memory**
  - Even if **main memory** is **getting cheaper**, **emerging applications** require **massive storage** capabilities (e.g., IoT, social media, deep space exploration, science, bioinformatics, etc).

Internal Structure of a DBMS



Physical storage

Physical **storage** medium

- **Permanent** storage (external): ***nonvolatile*** or long-term
- **Transient** storage (internal): ***volatile*** or short-term

Permanent storage (**secondary storage**) is usually ***relatively cheap***.

Transient storage (primary memory) is usually ***relatively expensive***.

Physical storage

Transient storage:

- **Main memory**
- **Cache**

Main memory is used to store data that is being used for *on-going computations.*

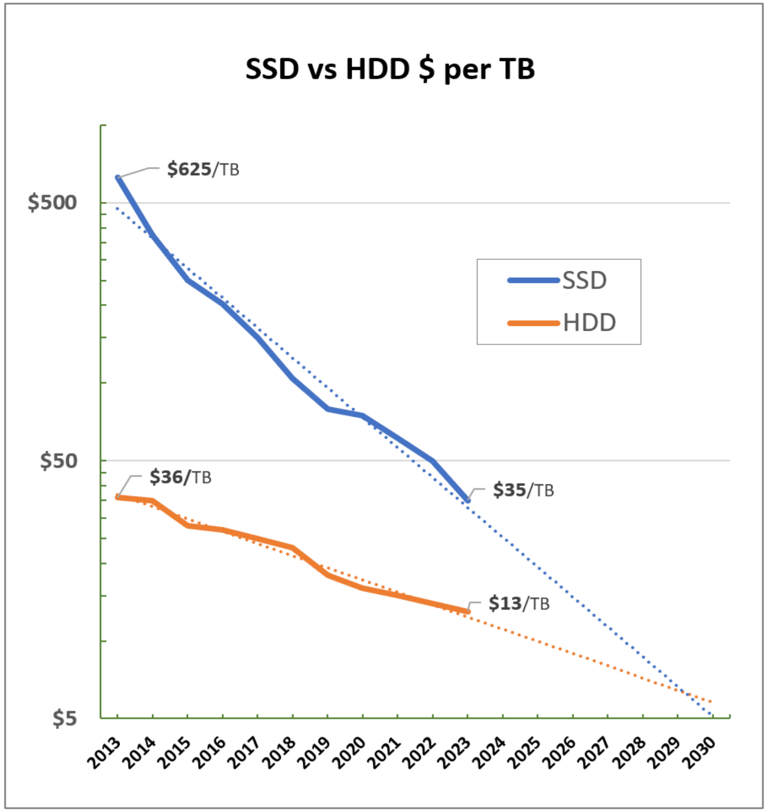
Cache is used to store data in very fast computer memory chips to speedup computations.

Cache sizes are **usually a lot smaller** than main memory sizes.

Physical storage

Secondary storage

- **Magnetic disk** (Hard Disk Drive - HDD): HDD devices are **slower**, but they have a **large storage** capacity.
- **Solid-State Drive** (SSD). SSD devices are **faster**, but they also **cost much more**.

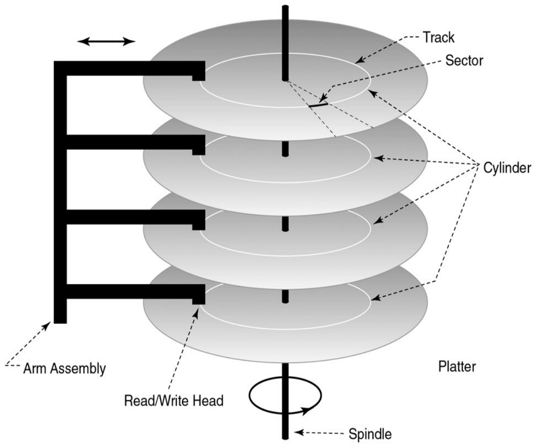


Tertiary storage

- **Magnetic cartridge tapes (tape silos)**: **sequential** access
- **Optical disk (juke boxes)**: **random** access

"live" databases are mostly **disk-based**. Therefore, we will focus on the ***magnetic disk*** medium.

Storage comparison



› **Hard Disk (Disk)**:

- ***Cheap:*** 1TB for $23 (2024)
- ***Permanent***: Once on disk, data will be there until it is explicitly wiped out!
- ***Slow***: Disk read/write *are slow*: 150MB/s to 250MB/s (2024)
- More coarsely addressable: **block addressable** (>= 512 bytes)

› **Random Access Memory (RAM) or Main Memory:**

- ***Expensive***: For $100 (2024), we get 16GB
- ***Volatile***: Data is lost when a crash occurs, power goes out, etc!
- ***Fast***: up to 8GB/s (2024), ~50 x faster for sequential access, ~100,000 x faster for random access, compared to disk access!
- More finely addressable: **bit addressable**

Secondary storage

Focus in database research has mainly been on secondary storage:

Why?

Databases are ***computationally I/O bound* – most** operations are **read** and **write** operations **between disk** and **memory**.

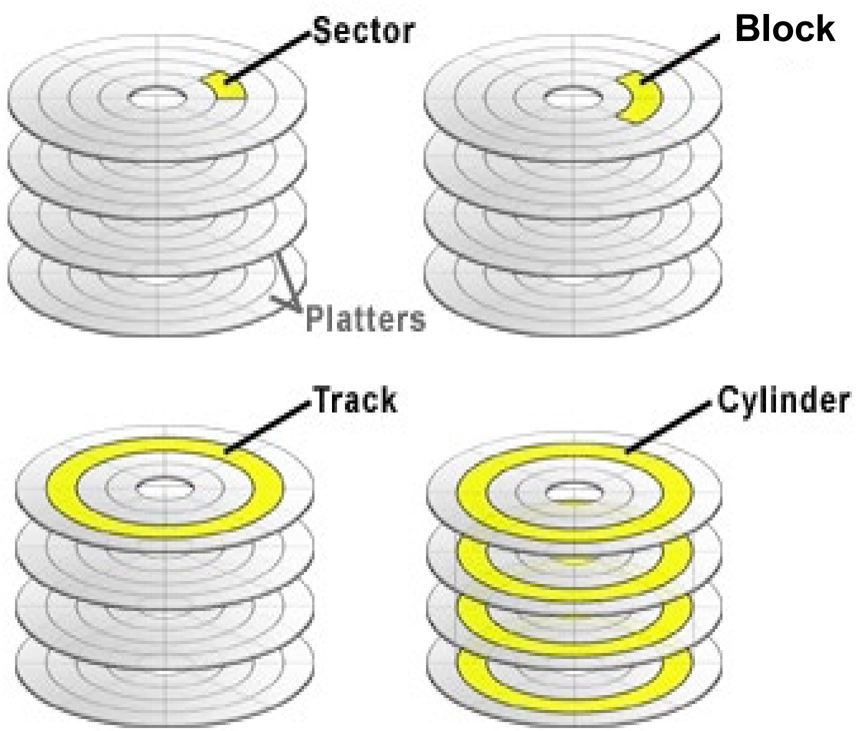
CPU bound computations are ***far less significant*** compared with **I/O bound computations**

Block on Disks

- Let us now turn to the representation of blocks on disks.
- **Block**:
  - typically defined by the **Operating System**
  - **unit of transfer** between **disk** and **main memory**
  - usually **spans more** than **one sector**. E.g. of block size: 4096 bytes (4K), 8K,….
  - **no concept** of **logical records** on disk.
  - once in **buffer**, a **block** is **divided** in **logical records** (i.e., tuples) by software.
- **Blocking factor**: Block size *vs*. record size.
  - ***b*** = *block size / record size*
  - ***Record spanning*** occurs when **b < 1**. This means that the *record size is **bigger** than the block size, i.e., we need **more than 1 block** to store **a record**!*
    - Important to **relational design** to **prevent record spanning**

Data is Organized as Blocks on Disk

- Due to the *high access latency*, data is organized in the form of ***data blocks*** on disk, such that the unit of **transfer** of disk read/write is a **block**
  - ***Block size*** is set by the operating system (OS) when the disk is initialized, usually as a multiple of sector size (typically 4KB or 8KB)

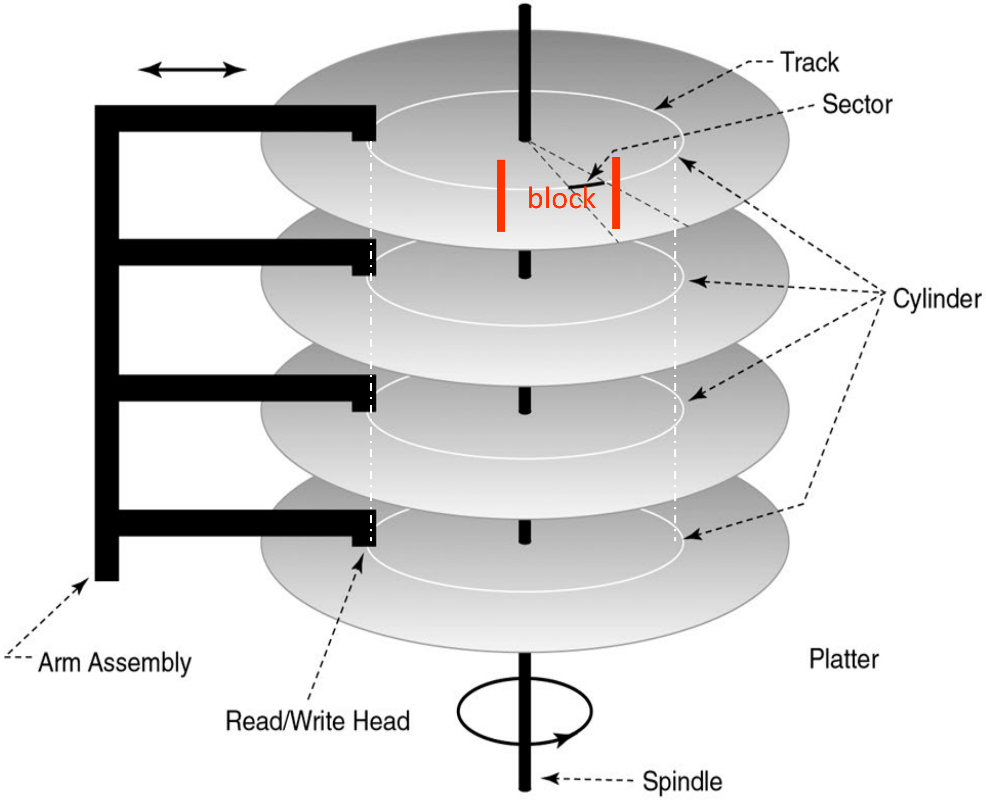


1KB = 1024 Bytes, 1MB = 1024*1024 Bytes

Data is Organized as Blocks on Disk

› **Time** for disk ***read/write***

1) ***Seek time***: the arm assembly is moved in or out to position a read/write head on a desired track
2) ***Rotational delay***: The platters spin
3) ***Transfer time:*** Reading/writing the data



Physical organization of a disk storage unit.

Buffer Management

Buffer management:

- Note that **databases** are usually **too large** to be held **totally** in **main memory**:
- The database **buffer** is **a main-memory area** used to **store database blocks**.
  - **Buffers** provide **virtual memory** for operations to be performed in main memory.
  - **Transfers** are done in **blocks** through a **buffer**.
  - Because **I/Os are expensive**, we try to **keep as many blocks** as possible **in memory** for later use.
    - Devise **strategies** that will “guess” which block the application will ask for, in such a way that **most likely it** will be in the **buffer** when **requested**.
    - **Buffer manager** is **responsible** for **managing** such **strategies**.

Main Functions of Buffer Manager

Buffer management:

- Define a ***buffer replacement strategy***:
  - If the buffer is **full** and a program needs a block, determine what block in buffer should be **replaced**, i.e., what is the ***best replacement algorithm***?

- Two main algorithms:
  - **LRU** (Least Recently Used)
  - **MRU** (Most Recently Used).

- There are other variants as well.

Example

Consider this natural join: **Borrower ⋈ Customer**

Assume **each relation** is in a **separate file** and **neither** one **fits** into main memory. Assume that the *only common attribute* in the two relations is *customer-name*. Let the following **program implement** the **join** above:

```
for each tuple b of Borrower do
    for each tuple c of Customer do
        if b[customer-name] = c[customer-name] then
            let x be a tuple defined as follows:
                x[branch-name] = b[branchname]
                ....
                x[customer-city]= c[customer-city]
                add x to result
            endif
    end
end
```

Focusing on only the buffering of the *Customer* relation, which buffer replacement strategy (LRU or MRU) will be better suited for this operation?

Example

Focusing on **only** the **buffering** of the ***Customer*** relation:

1st strategy: LRU (Least Recently Used)

- In this scheme, the *least recently used block* (assuming **n tuples fit** into **1 single block**) is the ***victim*** (i.e., ***selected** to be **replaced**)*.

```
for each tuple b of Borrow do
    for each tuple c of Customer do
        if b[customer-name] = c[customer-name] then
            let x be a tuple defined as follows:
                x[branch-name] = b[branchname]
                ....
                x[customer-city]= c[customer-city]
                add x to result
            endif
    end
end
```

However, according to the **join** program, a **more efficient strategy** would have been to **select** the **victim** the ***most recently*** used block! **Why?**

- *if a Customer block is used, it will **not be used again until all others have been used**.*

Example

2nd strategy: MRU (Most Recently Used)

- The **most recently used record is tossed out**. This makes sense in our case:
  - This is obviously an optimal strategy for our join

```
for each tuple b of Borrow do
    for each tuple c of Customer do
        if b[customer-name] = c[customer-name] then
            let x be a tuple defined as follows:
                x[branch-name] = b[branchname]
                ....
                x[customer-city]= c[customer-city]
                add x to result
            endif
    end
end
```

Example

Assume the buffer consist of 3 blocks, number of **Customer** records is **4** (c1, c2, c3, c4) and *one Customer record fits in exactly one block.*

After 3 access, the buffer looks like (a): **c1** is the *least recently used* and **c3** is the *most recently used.* Next operation is **Read c4**. So, **we must swap one Customer record out**.

LRU: **c1** is the **least recently** used and therefore it is **selected** as the **victim**. After this read operation, c1 is replaced by c4 as shown in (b).

According to the join algorithm: next, **Read c1**. The **victim** is **c2** as shown in (c). This implies we have a total of **2** ***page faults*** (i.e., 2 I/Os).

c1	c2	c3

(a)

c4	c2	c3

(b)

c4	**c1**	c3

(c)

Contrast with **MRU**: **Read c4**: the **next victim** is **c3** to be swapped out as shown in (d). Next, **Read c1** : **no I/O** required in this case. Therefore, the total ***page faults*** is 1. *Better than LRU in this case*.

c1	c2	c3

(a)

c1	c2	**c4**

(d)

Disk performance measurements

Measuring the **performance** of **hard disks**

Disk Access = **Seek time** + **Rotation latency**

(5ms - 10ms) = (3ms - 4ms) + (2ms - 7ms)

Data Transfer Rate = 40-200MB/s

- Typically between **5ms** and **25ms** per **MB** transferred
- **multiple disks** with shared interface can support **higher rates**

Mean Time to Failure (MTTF)

- Fact of life: Disks **do fail** every now and then.
- **MTTF** is a statistical value that describes the average operating time until the disk fails.
  - Vendors' claim: Depending on the disk model, MTTF is between one million hours (114 years!) and 2.5 million hours (285 years!) - For a single disk, MTTF is of *limited significance*.
    - A disk could fail at any time, so **regular backups** and **RAID configurations** are required to protect against data loss.
- In practice, MTTF is helpful with a large number of disks where MTTF helps to estimate how regularly failures could occur.
  - Example: Assuming a *uniform statistical distribution* failure rate, with an MTTF of **one million hours** and **one million disks**, a **disk failure** would be expected to occur **each hour**. With the **same** MTTF but with **1,000 disks**, a **disk failure** is expected to occur every **1,000 hours**!

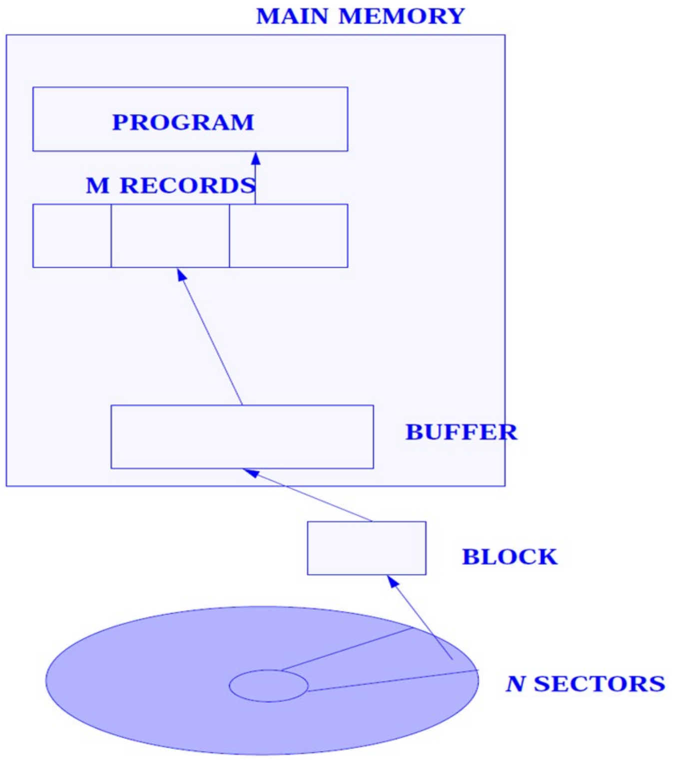
Strategies for physical disk transfer

Key approaches for ***physical disk-based transfer***:

- Block transfer
- Cylinder-based
- Multiple disks
- Disk scheduling
- Prefetching/double buffering

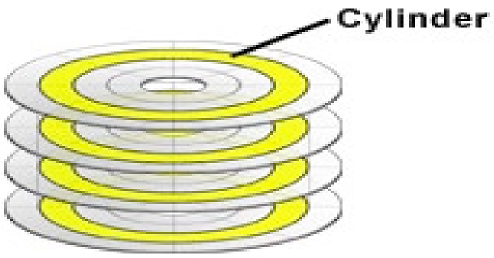
Block transfer:

- Data is always **transferred** by **blocks** to *minimize transfer time*
- A disk block is a set of *contiguous sectors from a single track of one platter*
- Block sizes typically range from 4k bytes to 32k



Cylinder-based Transfer

- Important *observation*: ***data*** *in* ***relations*** *is likely to be* ***accessed together***.
  - Principle: store relation data on the *same cylinder*: *blocks in the same cylinder* effectively involve only *one seek* and *one rotation latency*.
  - *Excellent* strategy if access can be *predicted*.



Multiple disks

- Observation: disk drives continue to become **smaller** and **cheaper**.
- Idea: use **multiple disks** to support **parallel access** - also enhances **reliability**.

RAID: Redundant **A**rray of **I**nexpensive (Independent) **D**isks

- *Load balance* multiple small accesses to *increase throughput*
- *Parallelize* large accesses so the *response time* is *reduced*.

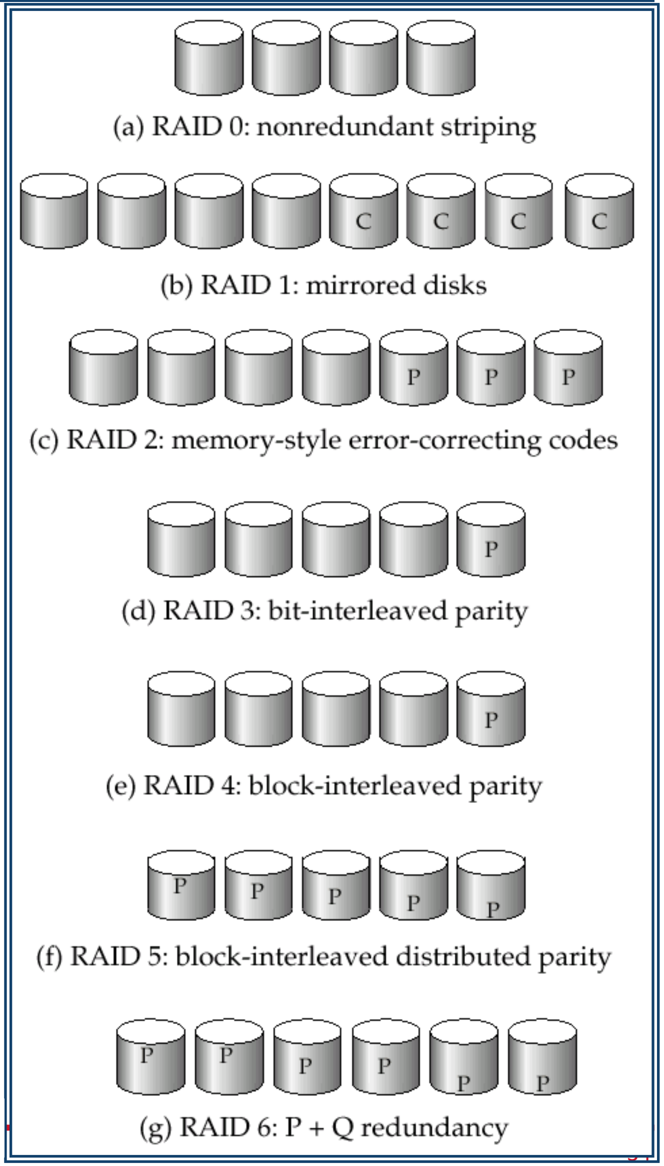
Multiple disks

RAID: Redundant **A**rray of **I**nexpensive (Independent) **D**isks

- disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
  - high capacity and high speed by using multiple disks in parallel, and
  - high reliability by storing data redundantly, so that data can be recovered even if a disk fails

RAID levels:

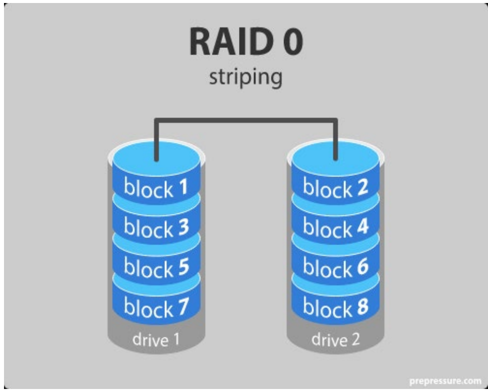
- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
- Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics



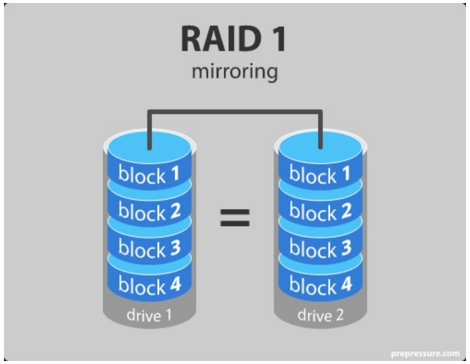
Multiple disks: Data Striping

Multiple disks

- Several organization choices:
  - Data is **partitioned** over **several disks** - **Block-leve**l striping (**RAID-0**)
    - Advantage:
      - *Increases rate of both predictable and random disk accesses*
    - Disadvantages:
      - Disk request *collisions* – e.g., if the requested data is on *several consecutive blocks in one disk*, this causes delays in the processing in memory.



  - **Mirror disks**: disks hold **identical** copies (**RAID-1**)
    - Advantage: ***increases read rate*** *without request collisions*.
    - Disadvantage: Writes are slower because data must be written to both disks.



Multiple disks: Data Striping

Other **data striping**

- **Bit-level** striping: split **groups of bits** over **different disks** (**RAID-2**)
  - For example we could split a byte into 8 bits where each bit goes to a different disk (total of 8 disks).
  - *ith* bit is written to the *ith* disk.
  - each disk participates in every access; thus, the number of accesses is the same, but every access reads 8 times as many data!
- **Byte-leve**l striping for sectors of a block (**RAID-3**)

Disk scheduling: **Elevator** scheduling

- Principle: *schedule* readings of requested blocks *in the order* in which *they appear* under the *disk head*.

Elevator approach:

- The arm moves towards one direction *serving all requests* on the *visited tracks*.
- *Changes direction* when *no* more pending *requests*.

- Advantage: *reduces* average access time for *unpredictable* requests
- Disadvantage: *benefit* is *not uniform* among requests and does not work well when there are only a few requests

Prefetching

- With **Buffering**:
  - Goal is to **keep** as **many blocks** in **memory** as possible to **reduce** disk **accesses.**
- **Prefetching** or **double buffering**:
  - Situation: *needed data is known in advance;* however, *the timing is data-dependent*.
    - Principle: speed-up access by *pre-loading* needed *data*.
    - Problem: requires *extra memory*

Short break

please stand up and stretch

Let us also menti….

How is Data Organized

- A table in a DBMS is **stored** as a **collection of records**, referred to as a ***file***
  - Each **file** consists of **one** or **more pages**
  - Each **page** stores **records** of **one type** (i.e., from **one table**)
  - A **page** is the **smallest data unit** in **main memory** that the **buffer manager is in charge of**.

Disk space manager

  - **Translates page** requests into **disk block** requests.
  - Each **page** typically consists of **one** or **more disk blocks**

In our **subsequent discussion**, we will assume that ***a page consist of a single block*** *unless* otherwise indicated.

Example of Physical Data Organisation

Relation(<u>tuplekey</u>, attribute1, …)

Table size: assume there are **2,000,000 records** in the table **Relation** and each **record** is **200 bytes** long, including a *primary key* **tuplekey** of **4 bytes**, an *attribute* **attribute1** of **4 bytes**, along with **other attributes**.

General features: assume that each **page size** is **4K bytes**, of which **250 bytes** are reserved for *header* and *array of record pointers*.

Questions:

- *How many bytes are required for a record?*
- *How many records can fill a page?*
  - We assume that **no record is ever split** across pages, i.e., ***no record spanning***.
- *How many pages are required to store the table?*

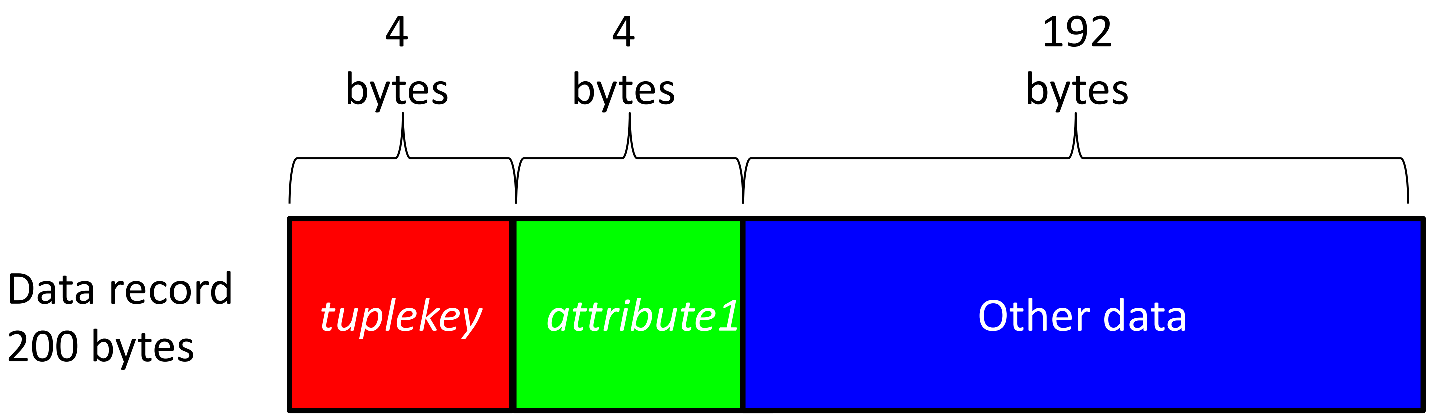
How Many Bytes per Record?

› Databases typically **store** each **tuple** (row) in a **record**

- *All fields* of a record are *stored together* ***consecutively***

From earlier
“each **record** is **200 bytes** long, including a primary key called **tuplekey** of 4 bytes, an attribute called **attribute1** of 4 bytes, and also other attributes”

Relation(<u>tuplekey</u>, attribute1, …)



How Many Records per Page?

Some space consumed by header/pointer data

250 bytes

All pages in DB have the same size, commonly 4KB (4096 bytes)

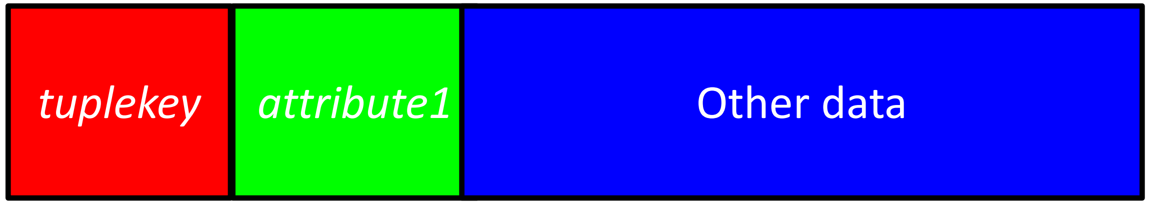
4096 – 250 = 3846 bytes

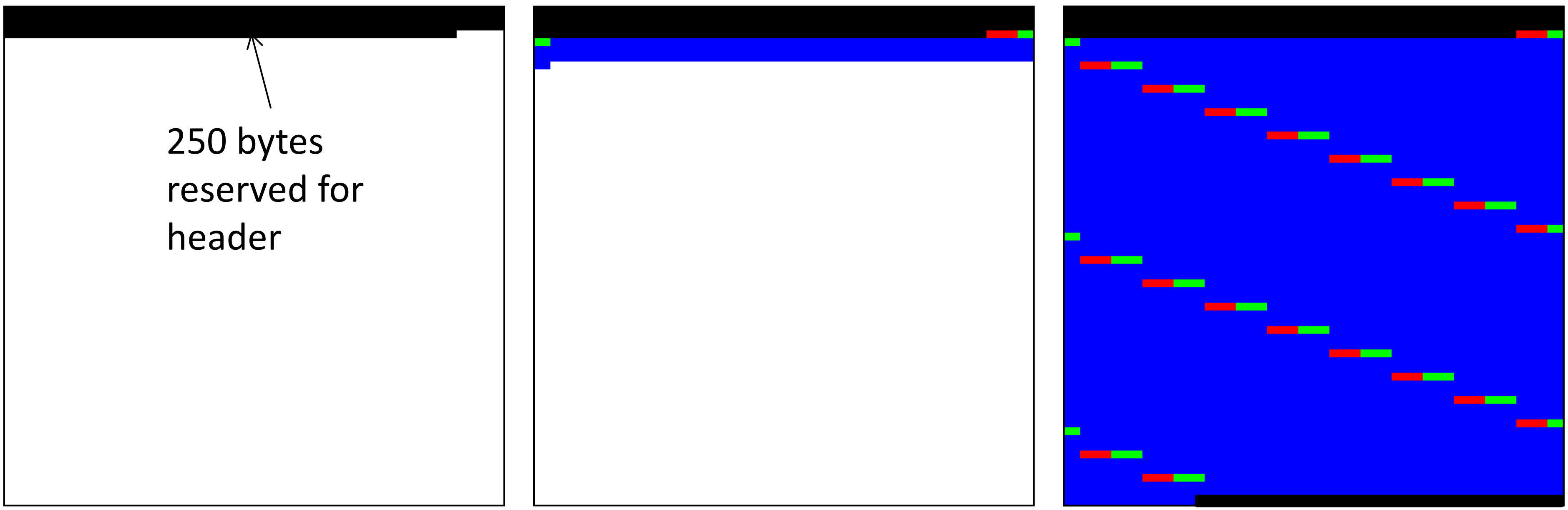
Remaining space available for **storing data**, e.g., record data, index entries

From earlier
"each page is 4K bytes, of which 250 bytes are reserved for header and array of record pointers"

How Many Records per Page?

Each example data record 200 bytes





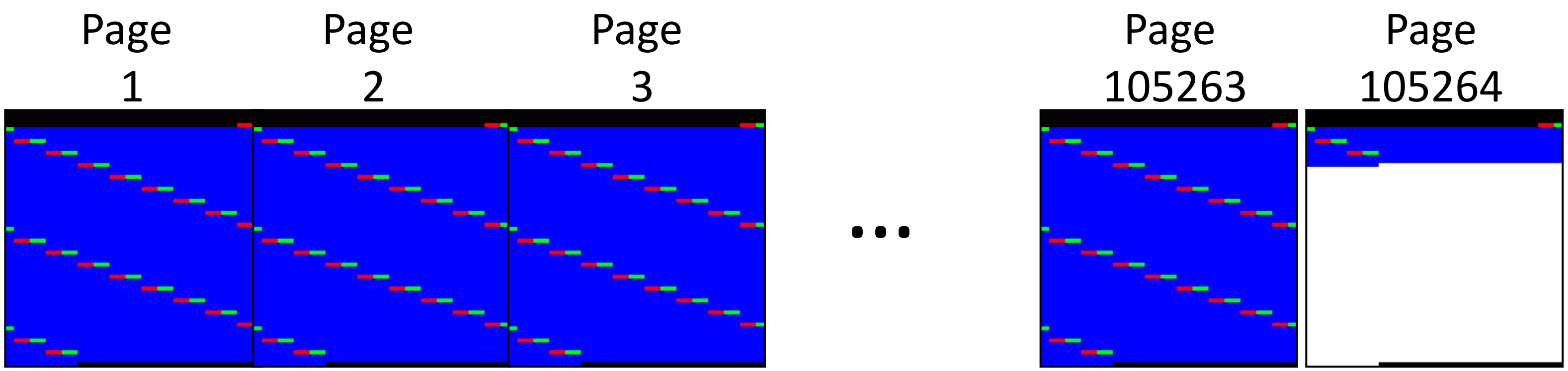
records: 0
empty space: 3846 bytes

records: 1
empty space: 3646 bytes

records: 19
Remaining space: **46 bytes**

⌊3846 bytes/page ÷ 200 bytes/record⌋ = ⌊19.23⌋ = 19 records/page **plus 46** *remaining* **bytes**

How Many Pages for the Table?



$$\left\lceil \frac{\text{2,000,000 records}}{\text{19 records/page}} \right\rceil$$ = **105,263** full pages **plus 1 page** containing **3** *remaining* **records = 105,264**

Total size of the required pages = **105,264** pages × **4096** bytes/page = **431,161,344** bytes

Actual **Table** size = 2,000,000 x 200 = **400,000,00**0 bytes

$$\textbf{Overhead} = \frac{431{,}161{,}344 - 400{,}000{,}000}{400{,}000{,}000}$$ = **7.79%** (~8% overhead)

Fill-Factor (Occupancy) for Pages

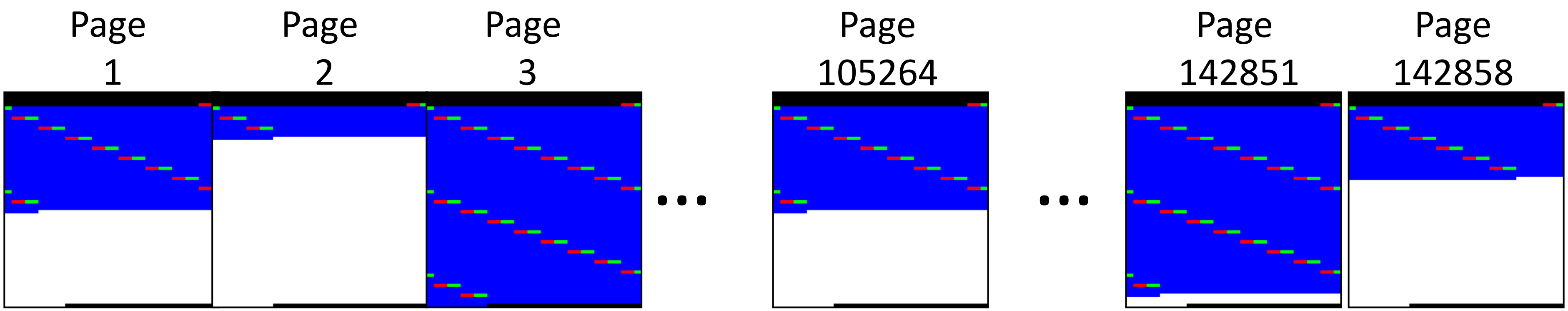
- **What is a fill-factor (occupancy)**?
  - The fill-factor value determines the **percentage** of space on each page to be **filled** with data, reserving the **remainder on each page** as **free space** to help in the **index expansion** when **new data is inserted**, **without** the **need** to **split** the **index page**.
  - The fill-factor value is a **percentage** from **1** to **100** with the value **100** meaning that the pages **are filled to capacity**.

- The **Fill-factor** is provided for **performance reasons**

- For example, specifying a **fill-factor** value of **80** *means* that **20 percent** of each page will **be left empty**.

- Further discussion can be found @ https://www.geeksforgeeks.org/sql-fill-factor-and-performance/

How Many Pages With a Fill Factor of Less Than 100?



Assume pages are 75% full on average: use **records/page** as the **initial size** of the page

$$\lfloor 19 \text{ records/page} \times 75\% \text{ average occupancy} \rfloor = \lfloor 14.25 \rfloor = 14 \text{ records/page (rounded down)}$$

$$\text{Total pages} = \left\lceil \frac{2{,}000{,}000 \text{ records}}{14 \text{ records/page}} \right\rceil = \textbf{142,858 pages}$$

Total size = 142,858 pages × 4096 bytes/page = **585,146,368** bytes

Using the previous **overhead formula**, this equates to **~47% overhead**

$$\frac{585{,}146{,}368 - 400{,}000{,}000}{400{,}000{,}000} = 46.28\% \text{ (}\mathbf{\sim 47\%\ overhead}\text{)}$$

Typical File Organizations

File Organisation*:* a **method** of **arranging** the **records** stored on **disk**.

- ***Unordered (Heap) Files*** – a record can be placed **anywhere** in the file **wherever** there is space (**random order**)
  - suitable when the typical access is a **file scan** retrieving **all records**.
- ***Sorted Files*** – store records **in sorted** order, **based** on the **value** of the **search key** of each record
  - may **speed up** searches using **binary search**, however **updates** can be **very expensive**
- ***Indexes*** – a **specialized data structure** to **organize** records via **trees** or **hashing**
  - like sorted files, they **speed up** *searches* based on values in certain fields ("search key").
  - **updates** are much **more efficient** than in **sorted** files.

Outline

- **Physical Data Organization: how data in DBMS are physically stored?**

- **Access Paths: how records are retrieved from DBMS?**
  - Access methods for heap files (linear scan)
  - Access methods for sorted files (binary search)
  - Access methods for indexes (index scan)

- **B+ Tree Index**
  - Primary index
  - Composite search keys

Access Paths

- An **Access Path** is a **method** for **retrieving** records, and **refers** to the **data structure** + **algorithm** used for **retrieving** and **storing** records in a table
  - ***Linear scan:* *Unordered (Heap) files*** *can be accessed with a* ***linear scan***
  - ***Binary search:* *Sorted files*** *can be accessed with a* ***binary search***
  - ***Index scan:* *Indexed data*** can be accessed with an ***index scan***

- **Physical Data Independence:**

SELECT * **FROM** Student **WHERE** sid = 5309650;

- **Definition**: The **choice** of an **access path** to use in the execution of the above SQL statement has ***no effect*** on the ***semantics*** of the statement
- **However**, this choice may have **a *major effect*** on the ***execution time*** of the SQL statement

Access Method for Unordered (Heap) Files

- Unordered (**heap**) file is the simplest file structure, which contains records in **no particular order**.

- Access method is ***linear scan*** *(file scan, table scan),* as records are unsorted in heap files.
  - On *average* ***half* of *all pages*** in a file must be read, but in the ***worst case*** the ***whole file*** must be read.

Example: Linear Scan

› **Linear Scan** (sequential search): For each page,

1) Load the page into main memory (cost = 1 I/O);

2) Check each record in the page for match;

Assume we need 140,351 pages to store our Table: How many page I/Os are needed to find records for:

- **SELECT** * **FROM** Relation **WHERE** tuplekey=715;

› For **equality search**, if tuplekey is **unique**, so can terminate on first match.

  › If a matching record is present, it will on average look through half of all pages, so we require 70,176 I/Os

  › For **zero matching records** or **non-unique** attribute, it needs to check every record, so we require 140,351 I/Os

How many page I/Os (out of 140,351 pages) are needed to find records for:

- **SELECT** * **FROM** Relation **WHERE** attribute1 **BETWEEN** 100 **AND** 119;

› For **range search** need to check each record, so 140,351 I/Os

Access Method for Sorted Files

Speeding up query processing

- Store records in a **sorted file** based on some attribute
  - Suppose we want to **search** for people of a specific **age**, and the records are sorted by **age**

11,12	13,14	15,16	17,18	19,20

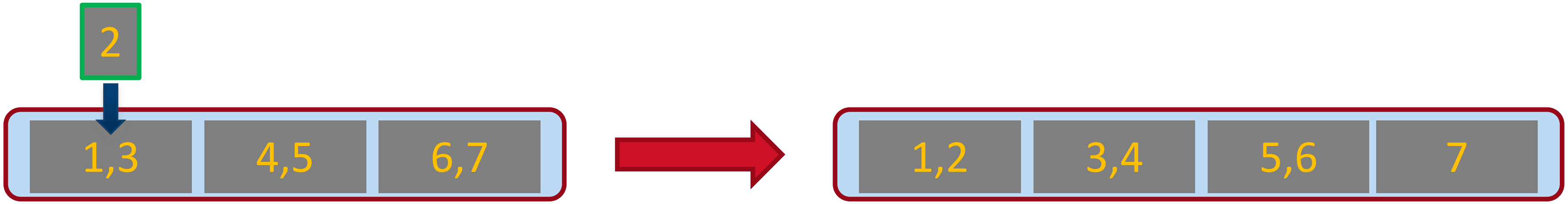
- Access method is ***binary search***
  - The I/O cost (worst case scenario) is ***log_2B + any additional pages containing retrieved records***

    (**B** is the **total number of pages** in the file)
  - *If B = 140,351 pages, the I/O cost f*or retrieving the page containing *first record* of *Table is* ***$log_2B = log_2 140{,}351 = 18$***

Drawback of Sorted Files

› Isssue: It is **expensive** to **maintain** a **sorted order** when **records** are **inserted**

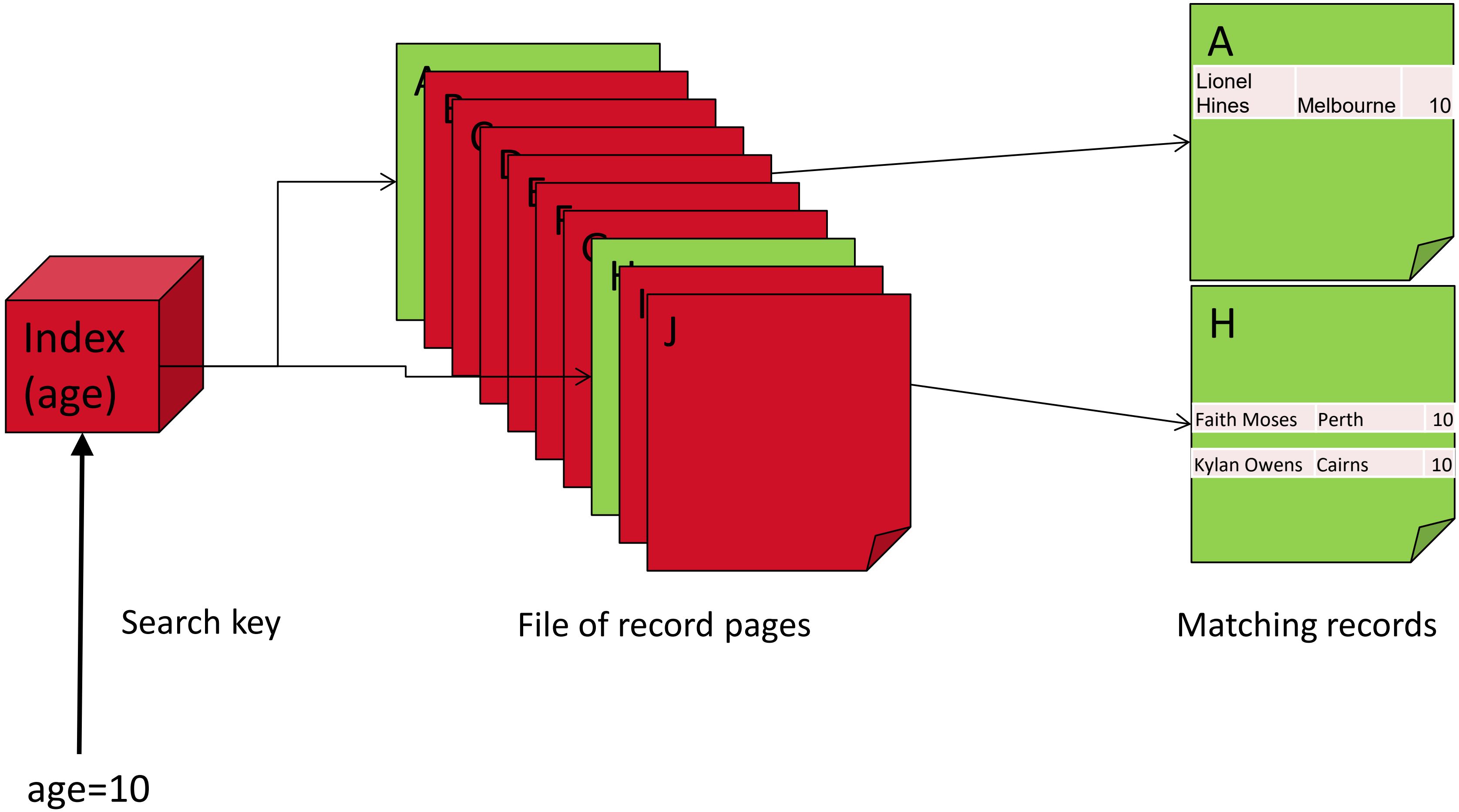
- After the **correct position** for an i**nsert** has been determined, it **needs to shift** all **subsequent records** to **make space** for the **new record**



- Hence, sorted files are typically hardly used by commercial DBMS, but rather they use *index-organised (clustered) files*

Speeding Up Access via Indexing

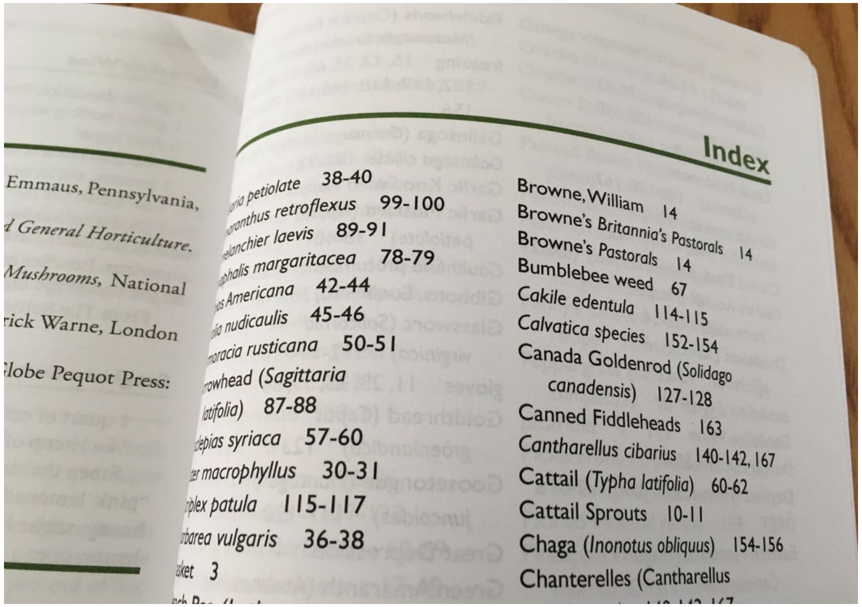
› To speed up access (i.e., reduce time), we add additional information to facilitate query answering



Indexing – Formal Definition

- An ***index*** is a <u>data structure</u> mapping <u>search key</u> values to <u>sets of records in a database table</u>
  - **Search key** properties
    - Any subset of fields
    - is **<u>not</u> necessarily the same** as the ***primary key*** *of a relation*
      - **Primary key** is *name* and **search key** may be *price*

Product(<u>name</u>, maker, price)



Indexing – Formal Definition

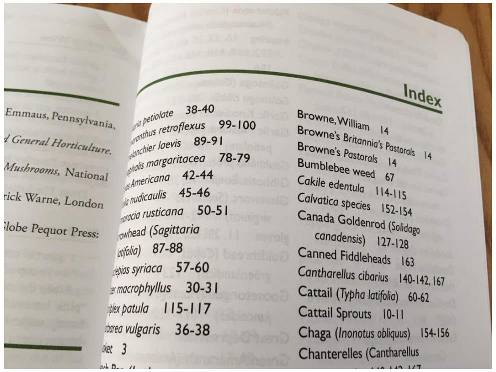
- An ***index*** is a data structure mapping search key values to sets of records in a database table
  - **Search key** properties
    - Any subset of fields
    - is **not necessarily the same** as the ***primary key*** *of a relation*
      - **Primary key** is *name* and **search key** may be *price*

Product(name, maker, price)

- An index provides efficient lookup & retrieval by search key value (***index scan***)
  - usually **much *faster*** than linear scan

- Typical index types
  - **Hash index** (can only be used for **equality** search)
  - **B+ tree index** (good for **range search**, but can also be used for **equality search**)

Indexing - The Downside



- **Downside:**
  - *Additional* storage space to store index pages
  - *Additional* I/O to access index pages
    (except if index is small enough to fit in main memory)
  - Index must be *updated* when table is modified.
    - depending on index structure, this could be quite costly

- Nevertheless, the **benefits** of using indexing far **outweigh** the **drawbacks.** indexing is used in all DBMSs to improve **performance.**
- **Indexing** is one of the most important **features** provided by a DBMS for **performance** purposes.

Outline

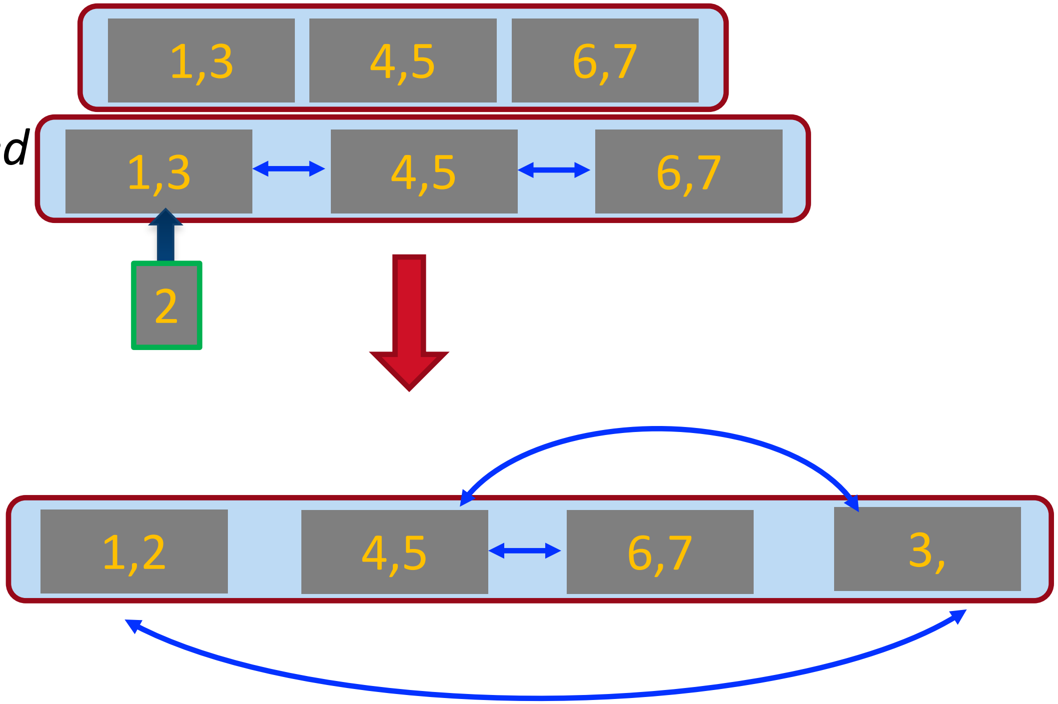
- **Physical Data Organization: how data in DBMS are physically stored?**

- **Access Paths: how records are retrieved from DBMS?**
  - Access methods for heap files (linear scan)
  - Access methods for sorted files (binary search)
  - Access methods for indexes (index scan)

- **B+ Tree Index**
  - Primary index
  - Composite search keys

General Idea of B+ Tree

- Maintain a ***logically*** **sorted file** to cope with **insertions/deletions**
- Example: Given the following set of pages
- Not *physically contiguous but sorted*

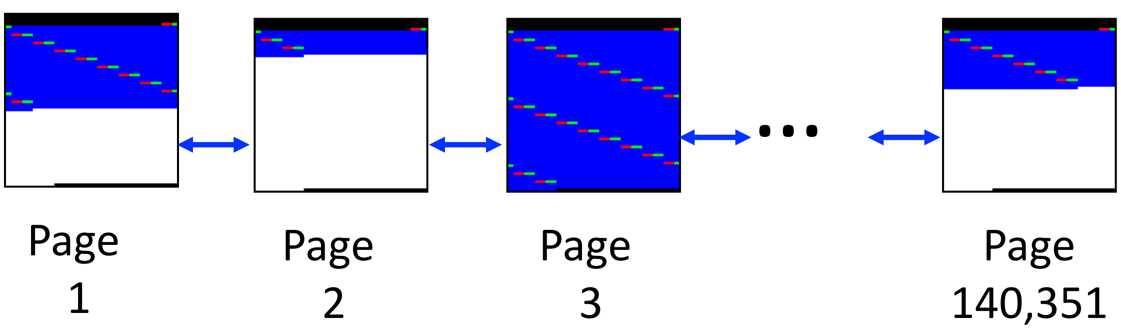


- **Bad news**: since this file is ***not physically*** sorted, we cannot perform a binary search (i.e., we can perform a binary search on an array, but not on a linked list)
- **Good news**: we can build an **index** for the data file

Step 1: Create a Logically Sorted Data File

- Assume a table **Relation** is stored in a ***data file*** consisting of **140,351 pages** with *search key* **tuplekey**
  - For each page, all **records** *within a page* are *physically sorted* on the *search key* (let us assume in increasing order)
  - **Pages** are ***logically sorted***: this means that all records in page *i* have smaller search key values than all records in page *i+1*
    - Note: these pages **may *not* be stored consecutively** on disk
  - **We call this file a logically sorted file based on search key *tuplekey***

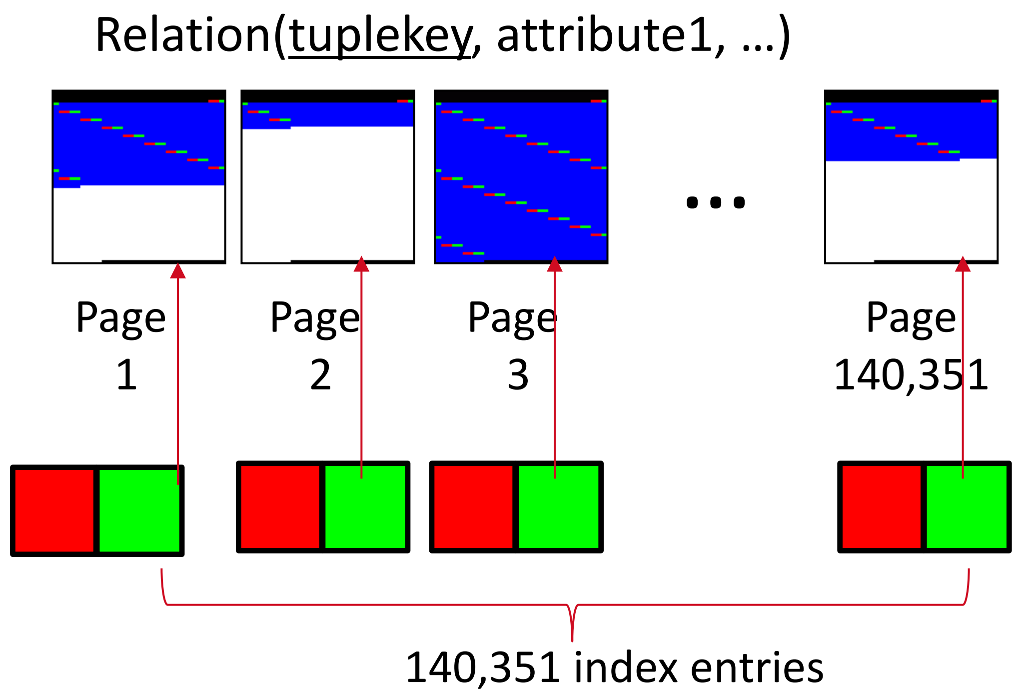
Relation(<u>tuplekey</u>, attribute1, …)

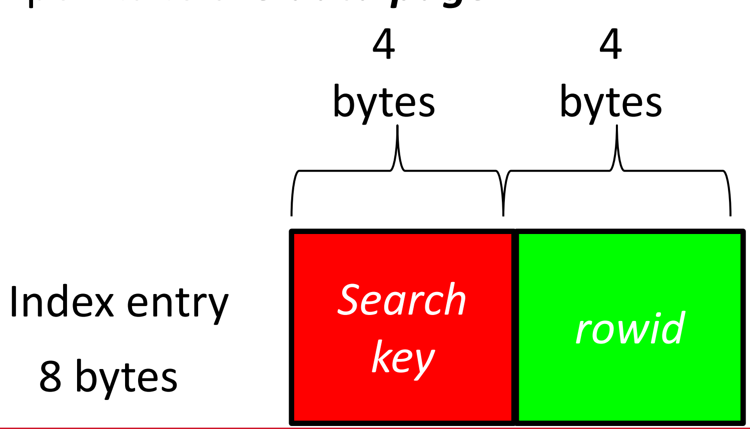


Step 2: Create One Index Entry for Each Record Page

- The table Relation is stored in a ***logically sorted data file*** consisting of 140,351 pages with search key represented by ***tuplekey***

- Create one **index entry** for **each page of the data file**, where the index entry stores a ***pointer rowid*** which points to ***the data page***



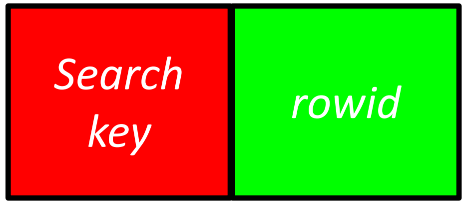


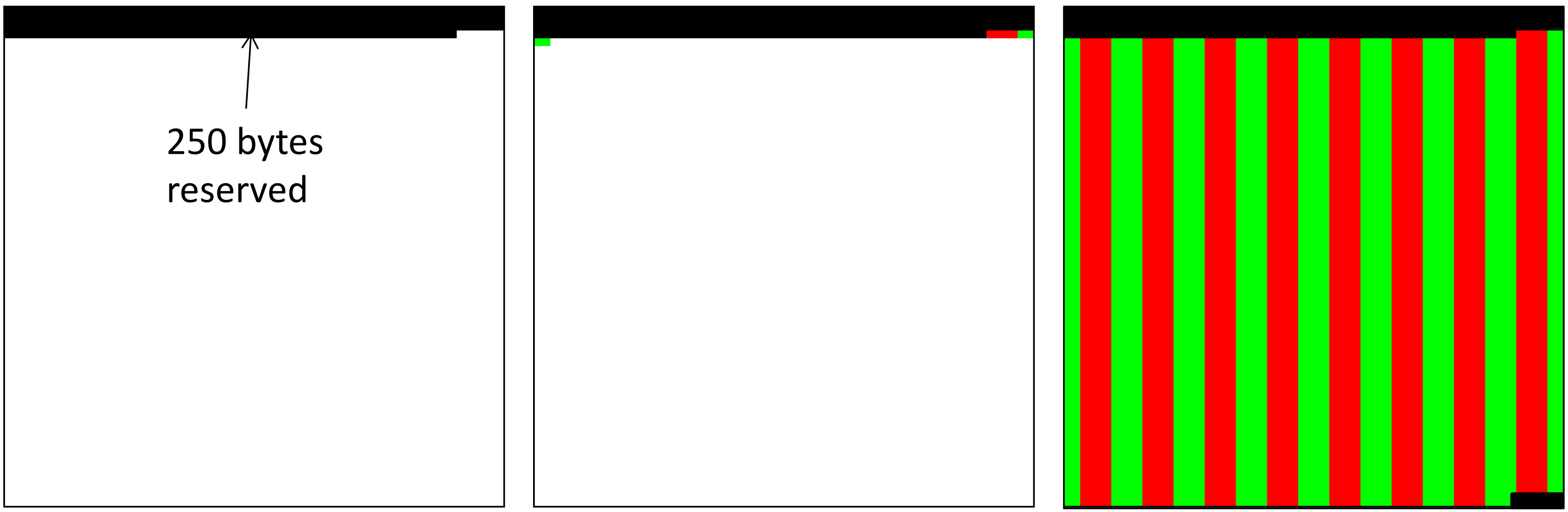
Index on **tuplekey:** search key is the size of **tuplekey**.

rowid is a 32-bit pointer (so 4 bytes).

Step 3: Fit Index Entries into Pages

Each example index
Entry is 8 bytes





entries: 0
empty space: 3846 bytes

entries: 1
empty space: 3838 bytes

entries: 480
empty space: 6 bytes

⌊3846 bytes/page ÷ 8 bytes/entry⌋= 480 entries/page

plus 6 remaining bytes

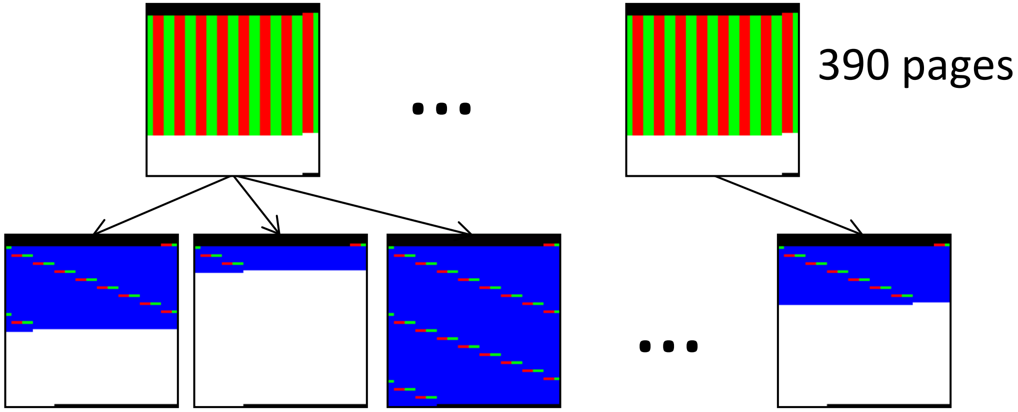
Step 3: Fit Index Entries into Pages (cont')

› Let us call a page *storing index entries* an ***index page***

- Recall each index page holds up to 480 index entries
- Assume average occupancy (i.e., fill factor) of 75%
- The average number of index entries in an index page is 480 × 75% = 360

› The **number** of **index pages** is $\lceil 140{,}351/360 \rceil$ = 390 (rounded up)

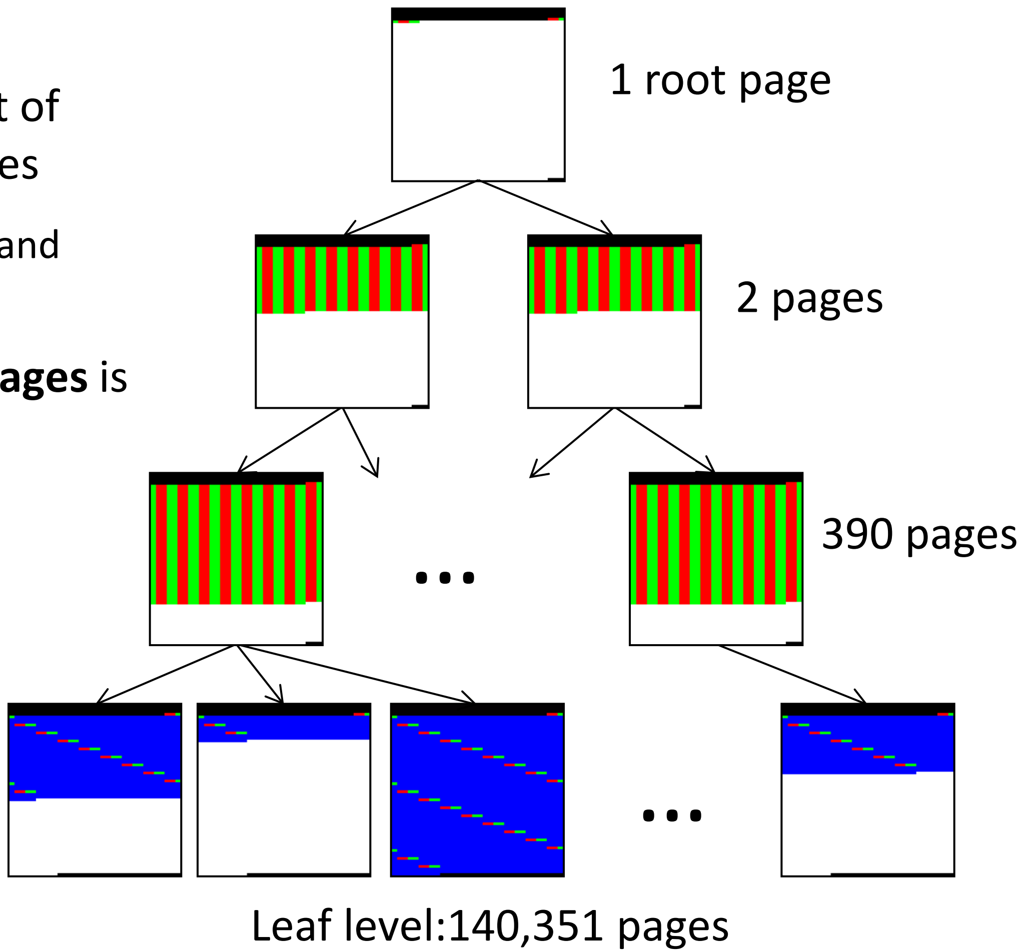
› We could store these 390 index pages as a <u>logically</u> sorted file and **then** *recursively* build indexes on it



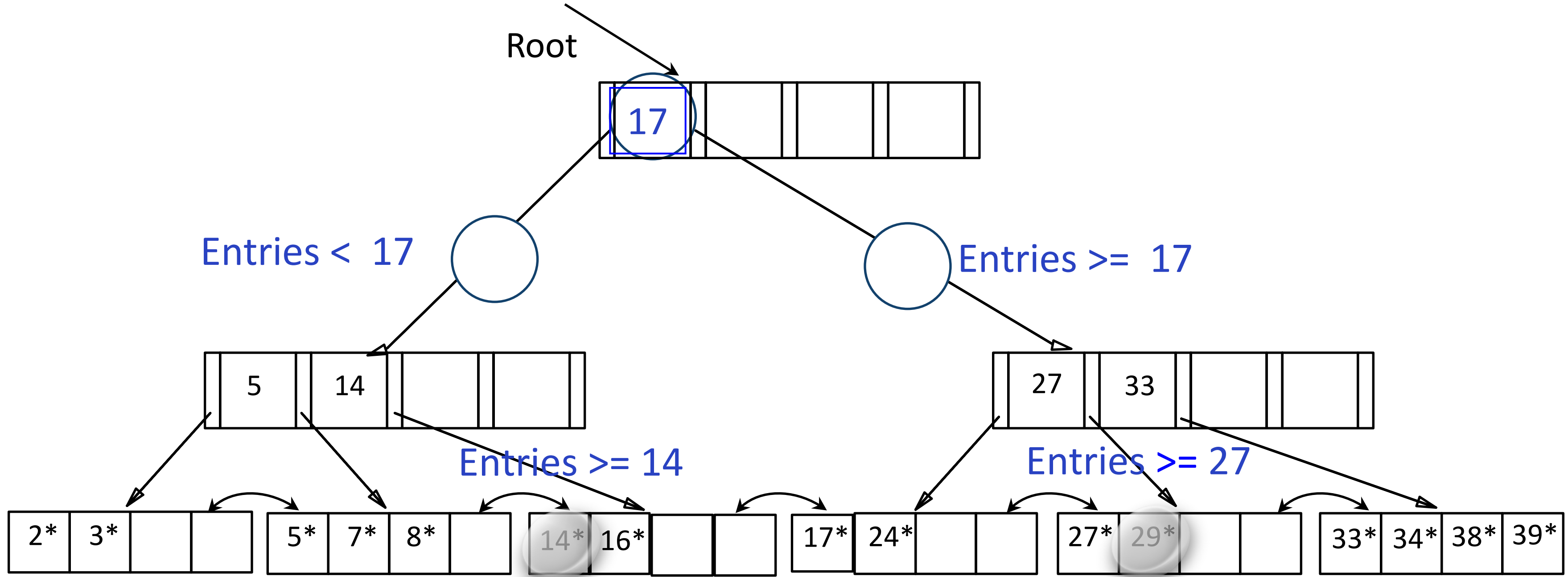
Step 4: Build B+ Tree Index

Noting that the average number of index entries in an index page is 480 × 75% = 360

- The next-level index would then consist of 390 index entries, and 390/360 = 2 pages
  - The **root level** consists of 2 index entries, and 2/360 = 1 page
- Therefore, the **total number** of **index pages** is 390 + 2 + 1 = **393 index pages**
- 393/140,351 = **0.2% increase**



Searching in a B+ Tree Index

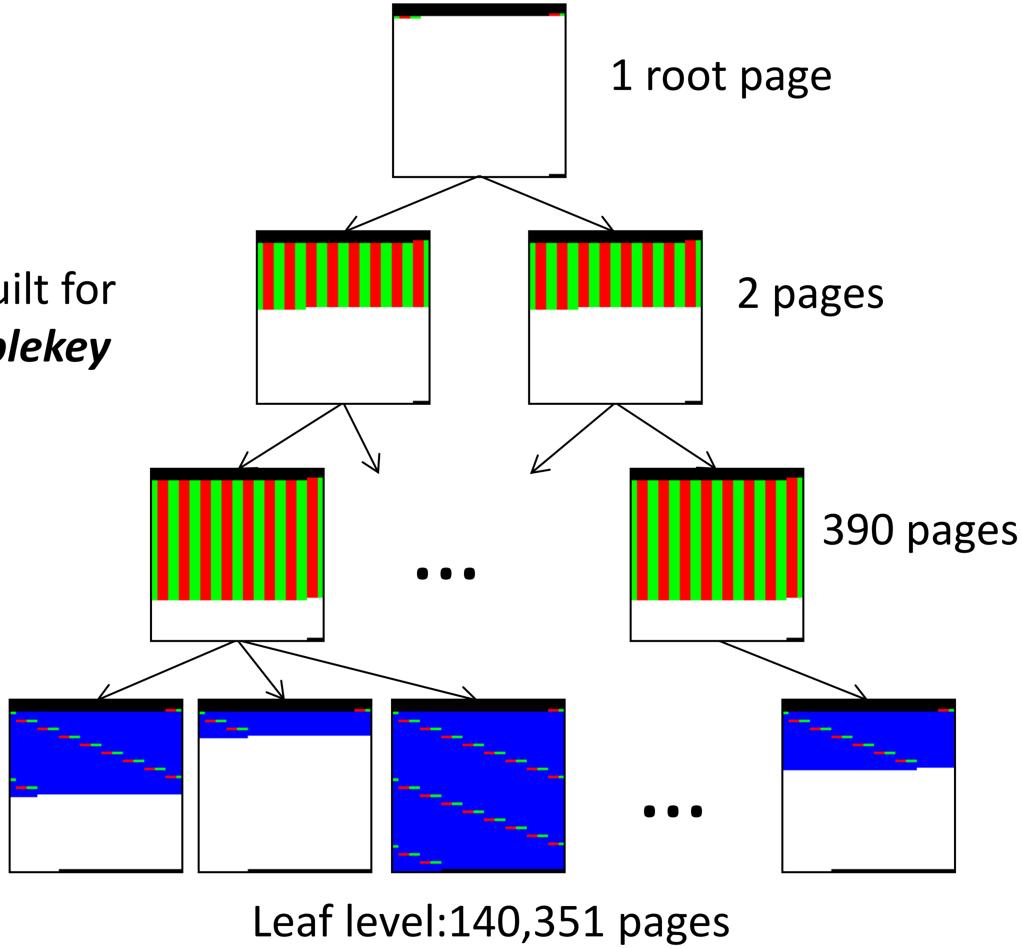


- Note how **record/data pages in the leafs** are sorted and linked
- **Find 14**
  Find 29
- The cost of searching in a B+ tree index: *the lookup cost* + *number of pages containing retrieved records*
  - **Lookup cost** is equal to ***the number of levels of the B+ tree + data pages***: In B+ tree, **all paths** from the root to a leaf are of **the same length**

Example: Cost of Equality Search in a B+ Tree Index

Example: **SELECT** * **FROM** Relation **WHERE** tuplekey=715;

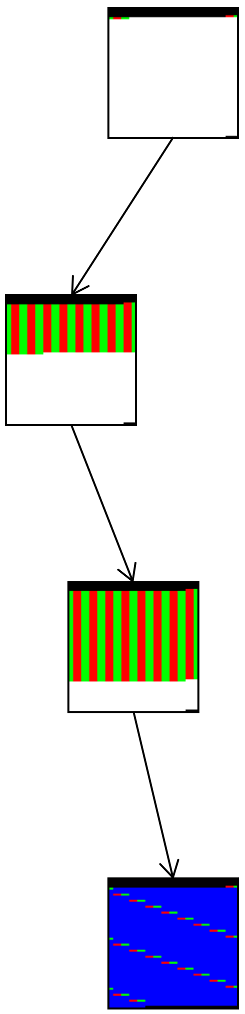
The B+ tree built for search key ***tuplekey***



Example: Cost of Equality Search in a B+ Tree Index

SELECT * **FROM** Relation **WHERE** tuplekey=715;

- For **equality search** on tuplekey, we can use the related index:
  1) Load index root into main memory (cost 1 I/O);
  2) Find location of matching page in next level;
  3) Load matching next level page (cost 1 I/O);
  4) Find location of matching page in following level;
  5) Load matching page on following level (cost 1 I/O);
  6) Find location of matching record page in leaf level;
  7) Load matching record page (cost 1 I/O);
  8) Check each record in the record page for match.
- Total of **4 I/Os** vs **70,176 I/Os** for heap file



SELECT * **FROM** Relation **WHERE** ***attribute1*** **BETWEEN** 100 **AND** 119;

- For **range search** on **attribute1**, the index above is of **no use. Why? We still need to check each record page sequentially**, so **140,351 I/Os**: Alternative: *build another index on attribute1*.

Outline

- **Physical Data Organisation: how data in DBMS are physically stored?**
- **Access Paths: how (specific) records are retrieved from DBMS?**
  - Access methods for heap files (linear scan)
  - Access methods for sorted files (binary search)
  - Access methods for indexes (index scan)
- **B+ Tree Index**
  - Primary index
  - Composite search keys

Composite Search Key: Field Order Matters

- **Composite search key**: case where the **search key** contains **several fields**
  - (age, salary): first (logically) sort on age, then among records with the same age, (logically) sort on salary
  - (salary, age): first (logically) sort on salary, then among records with the same salary, (logically) sort on age
  - “*Index* with *search key (age, salary)*” ≠ “*Index* with *search key (salary, age)* ”

	age	salary
r1	22	2000
r2	21	2000
r3	23	2000
r4	22	3000
r5	22	1000

Heap file order

	age	salary
r2	21	2000
r5	22	1000
r1	22	2000
r4	22	3000
r3	23	2000

Sorted order on search key *(age, salary)*

	age	salary
r5	22	1000
r2	21	2000
r1	22	2000
r3	23	2000
r4	22	3000

Sorted order on search key *(salary, age)*

Composite Search Key

- Given an **index** with search key **(age, salary)**
  - Can we use the index to lookup search condition “age = 22”? YES!
  - Can we use the index to lookup search condition “age >= 22”? YES!
  - Can we use the index to lookup search condition “salary = 2000”? NO!
  - Can we use the index to lookup search condition “salary >= 2000”? NO!
  - Can we use the index to lookup search condition “age = 22 AND salary = 2000”? YES!
  - Can we use the index to lookup search condition “age = 22 AND salary >= 2000”? YES!

	age	salary
r2	21	2000
r5	22	1000
r1	22	2000
r4	22	3000
r3	23	2000

Sorted order on search key (age, salary)

Important indexing rule: An index with a search key can be used to lookup a search condition if all records satisfying the search condition are consecutive in the sorted order of the table’s records based on that search key

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 9 (9.1-9.4)
  - *Kifer/Bernstein/Lewis gives a good overview of indexing*
- Ramakrishnan/Gehrke (3rd edition)
  - Chapter 8
  - *The Ramakrishnan/Gehrke is very technical on this topic, providing a lot of insight into how disk-based indexes are implemented.*
- Ullman/Widom (3rd edition - ‘1st Course in Databases’)
  - Chapter 8 (8.3 onwards)
  - *Mostly overview, with simple cost model of indexing*
- Silberschatz/Korth/Sudarshan (5th ed)
  - Chapter 11 and 12

Next Week: Query Processing and Evaluation

- Ramakrishnan/Gehrke – Chapters 13 and 14
- Kifer/Bernstein/Lewis – Chapter 10
- Garcia-Molina/Ullman/Widom – Chapter 15

See you next week!