COMP9120

Week 10: Storage & Indexing

Semester 2, 2022

Professor Athman Bouguettaya
School of Computer Science

THE UNIVERSITY OF SYDNEY

Contents developed by Dr Lijun Chang

THE UNIVERSITY OF SYDNEY

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 Darug people and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

THE UNIVERSITY OF SYDNEY

Copyright warning

Outline

- **Physical Data Organization: how data in DBMS is 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

Outline

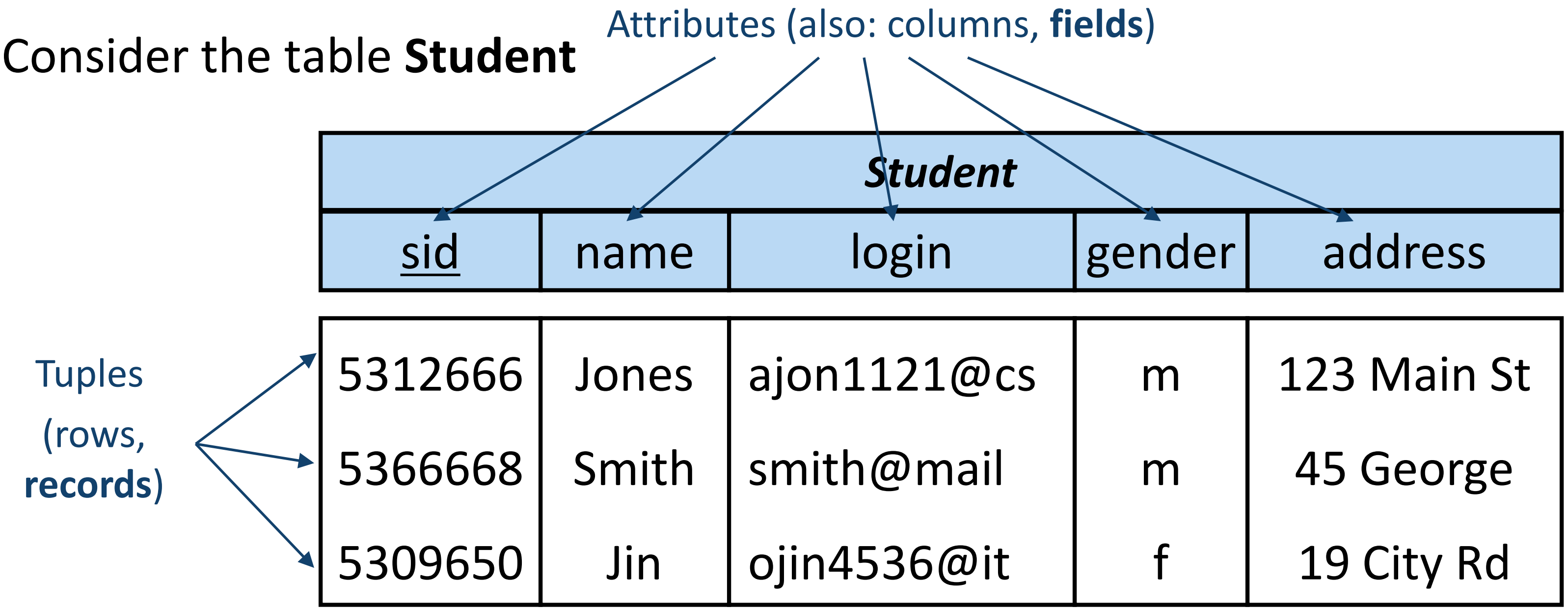
- **Physical Data Organization: how data in DBMS is physically stored?**

- **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

How is a relation physically stored and accessed?

› Consider the table **Student**



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

Tuples (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?

› How are simple queries involving only one table evaluated?

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

› What is the cost?

Data storage

Two fundamental DBMS questions:

- How do we efficiently **organize** very large volumes of data?
- How do we **access** data to minimize I/Os?

Constraints:

- Databases need persistent storage
- Cannot hold all database content in main memory
- Massive quantities of data: always more than we can fit in main memory
- Even if main memory is getting cheaper, emerging applications require massive storage capabilities (IoT, social media, deep space exploration, science, bioinformatics, etc).

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 medium

- 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

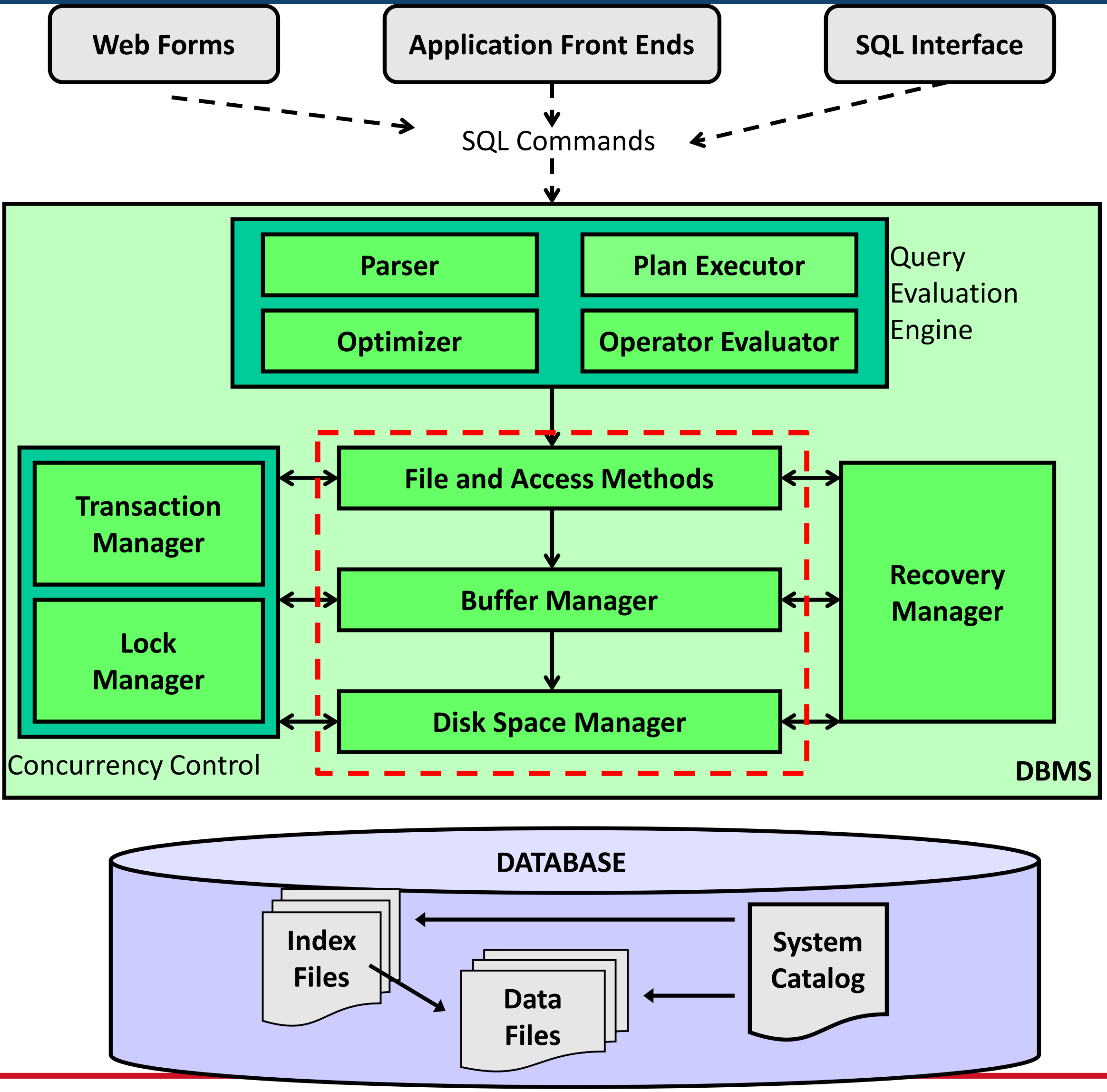
- Magnetic disk (every day use)

Tertiary storage

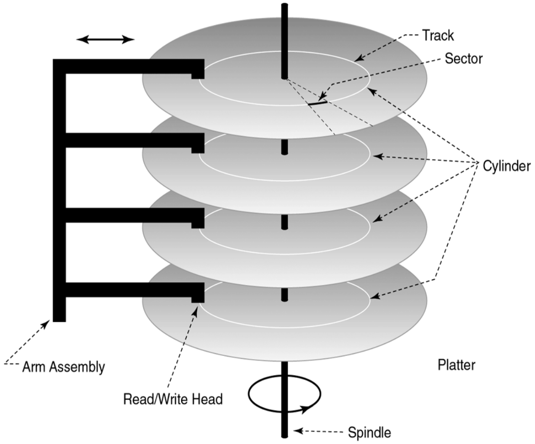
- Magnetic cartridge tapes (tape silos) : sequential access
- Optical disk (juke boxes): random access

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

Internal Structure of a DBMS



Storage comparison



› **Hard Disk (Disk)**:

- ***Cheap:*** 2TB for $100
- ***Permanent***: Once on disk, data will be there until it is explicitly wiped out!
- ***Slow***: Disk read/write are slow!
- More coarsely addressable: block addressable (>= 512 bytes)

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

- ***Expensive***: For $100, get 16GB of RAM
- ***Volatile***: Data can be lost if e.g. crash occurs, power goes out, etc!
- ***Fast***: ~50 x faster for sequential access, ~100,000 x faster for random access, compared to disk access!
- More finely addressable: bit addressable

DBMS stores information on (“hard”) disks.

Secondary storage

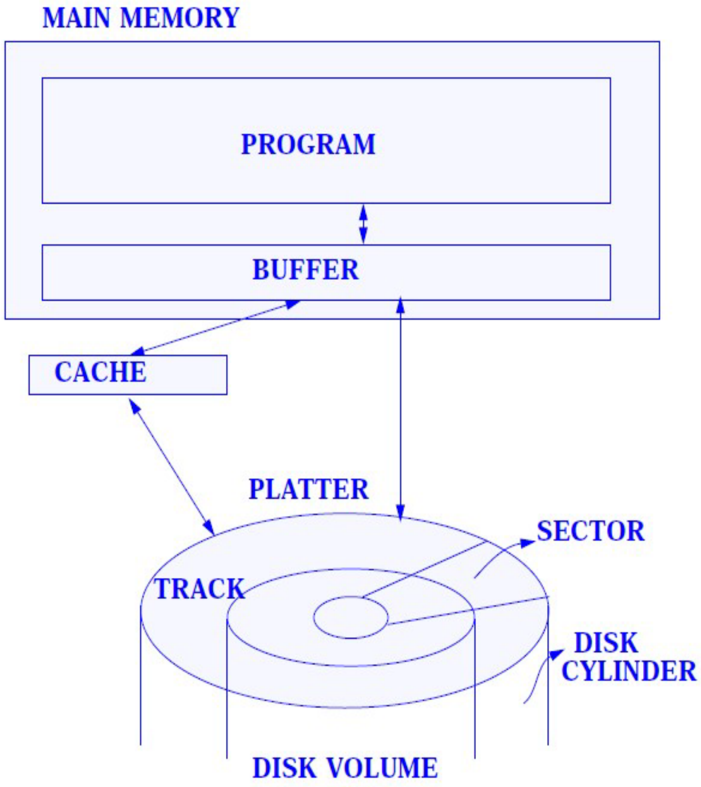
Focus in database research has mainly been on secondary storage:

Why?

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

Memory Hierarchy

Memory hierarchy: operations when reading and writing records:



Physical address of item x:<Volume#,Platter#,Track#,Sector#>

Data is Organized as Blocks on Disk

- Time for disk ***read/write***

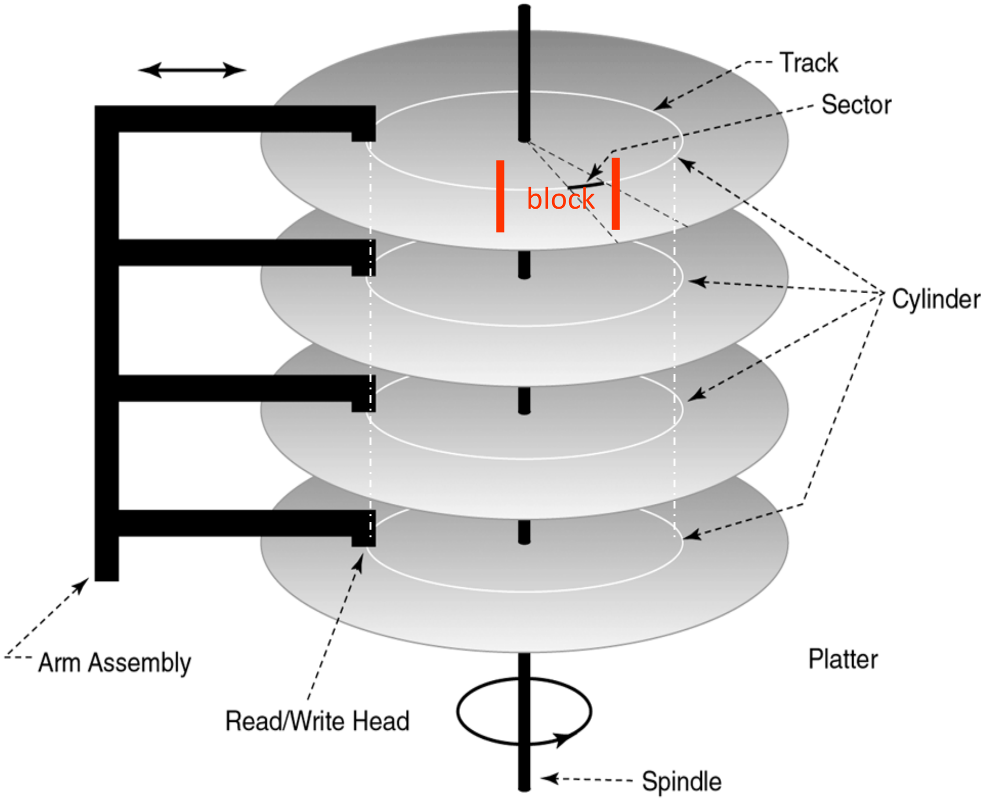
  1) ***Seek time***: the arm assembly is moved in or out to position a read/write head on a desired track

  2) ***Rotational delay***: The platters spin

  3) ***Transfer time:*** Reading/writing the data

- 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

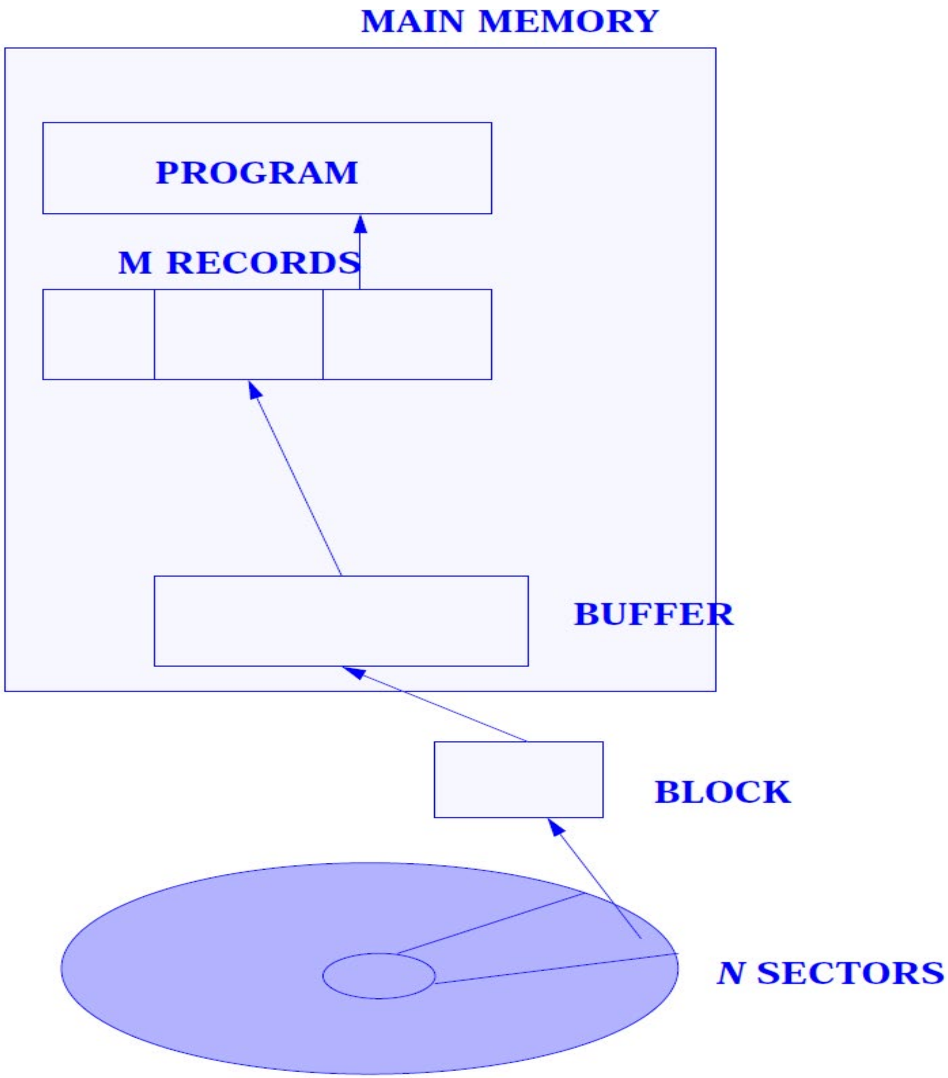
  - ***Block size*** is set by the operating system (OS) when the disk is initialized, usually as a multiple of sector size (typically 4KB or 8KB)



Physical organization of a disk storage unit.

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

Disk Transfer



Block on Disks

Block transfer:

- 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, it is divided in logical records by software.
- Record blocking definition: The number of records allocated to a block is called the *blocking factor*.
  - *b = block size / record size*
  - *Record spanning* occurs when b < 1.

Buffer Management

Buffer management:

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

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) and **MRU** (Most Recently Used). There are other variants as well.

› Note: for recovery purposes and irrespective of the approach used, we need to take into account the following parameters:

  - *Pinned blocks* (may not be written back to disk as they may be needed by recovery manager in case of a crash).
  - *Forced output* of blocks to ensure consistency: blocks dependent on each other (e.g., in case of crash).

› *LRU* works better in operating systems. Research indicates it is not always optimal for databases. *MRU* is good for many applications.

Example

Example:

Borrow ∞ Customer

Assume each relation is in a separate file. Assume that the only common attribute in the two relations is *customer-name*. Let the following program do the join:

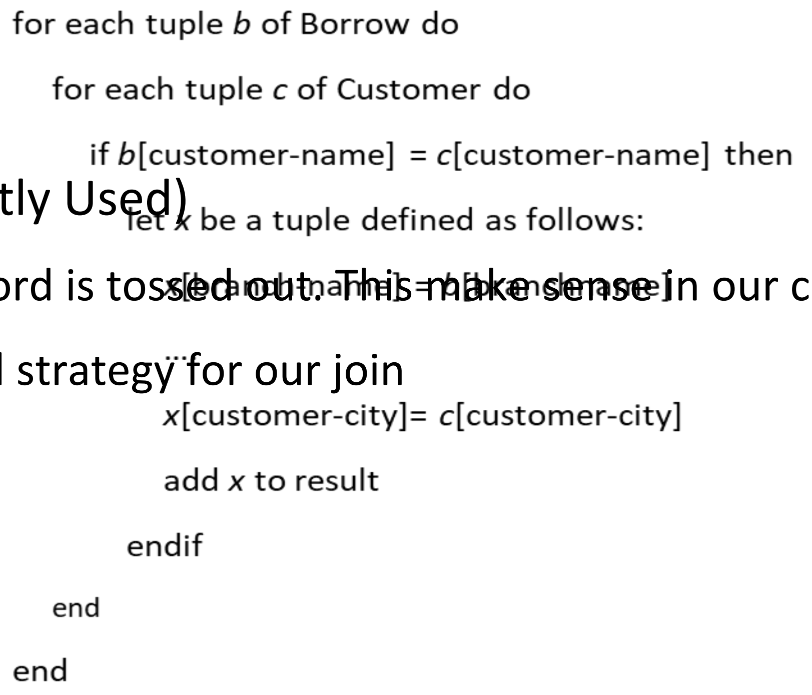
```
for each tuple b of Borrow do
    for each tuple c of Customer do
        if b[customer-name] = c[customer-name] then
            let x be a tuple defined as follows:
                x[branch-name] = b[branchname]
                ….
                x[customer-city]= c[customer-city]
            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

Example

Considering only the *Customer* relation:

1st case: LRU (Least Recently Used)

- In this scheme, *least recently* used block is the *victim* (i.e., *to be selected to be replaced)*. According to the join program, the victim should have been the *most recently* used block.
- Why? if a Customer block is used, it will not be used again until all others have been used.



```
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[branch-name]
        ...
        x[customer-city]= c[customer-city]
        add x to result
    endif
  end
end
```

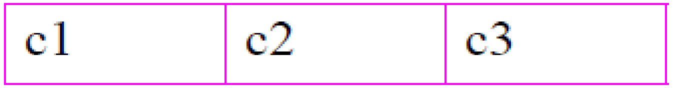
2nd case: MRU (Most Recently Used)

- The most recently used record is tossed out. This make sense in our case
- Obviously, this is an optimal strategy for our join

Example

Example:

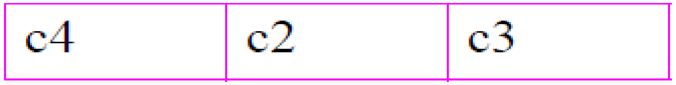
Assume 3 blocks in the buffer, i.e., n=3. Assume the number of Customer records is 4 (c1, c2, c3, c4) and one Customer record fits in exactly one block. Assume that currently the buffer looks as follows (c1 is the least recently used and c3 is the most recently used):



c1	c2	c3

Next, read c4. Therefore, we need to swap one Customer record out.

LRU: c1 is the least recently used Customer record and therefore it is picked as the victim. After this read operation, the buffer looks like:



c4	c2	c3

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

Contrast with **MRU**: read c4: the victim is c3

Next, read c1 : no I/O required in this case. Therefore the total *page faults* is 1. Better than LRU in this case.

Disk performance measurements

Performance measurement of hard disks

Disk Access = Seek time + Rotation Latency

8 - 20 ms = 3 -1 0 ms + 5 - 10 ms

Data Transfer Rate = 40-200MB/s

between 5ms and 25ms per MB transferred

multiple disks with shared interface can support higher rates

Mean Time To Failure (MTTF)

Vendors' claim: ~30k hours (3.6 years) - 1.2M hours (136 years)

In practice: 1000 disks with 1.2M hours MTTF. This implies that on average 1 disk will fail every 1200 hours!

Strategies for disk access

Strategies for disk access time:

- Block transfer
- Cylinder-based organization
- Multiple disks
- Disk scheduling and elevator algorithm
- Buffering and prefetching

Block transfer

Block transfer:

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

Cylinder-based organization

Cylinder-based organization

- Observation: data in relations is likely to be accessed together.
- Idea: store data on the same cylinder: blocks in the same cylinder effectively involve only one seek and one rotation latency.
- Advantage: excellent if access can be predicted in advance.

Multiple disks

Multiple disks

- Observation: disk drives continue to become smaller and cheaper.
- Idea: use multiple disks to support parallel access - also enhances reliability.
- Organization alternatives:
  - Data are partitioned over several disks
    - Advantage:
      - Increases rate of predictable and random disk accesses
    - Disadvantages:
      - Disk request collisions – e.g., requested data is on several consecutive blocks in one disk, thus delaying the processing in memory.
      - Cost of several small disks is greater than a single one with the same capacity.
  - Mirror disks: disks hold identical copies
    - Advantage: increases read rate without request collisions.

Multiple disks: Data Striping

Multiple disks: Data striping

- RAID: Redundant Array of Inexpensive (Independent) Disks
  - Load balance multiple small accesses to increase throughput
  - Parallelize large accesses so the response time is reduced.
- Data striping: splitting data units across multiple disks
  - Bit-level striping: split groups of bits over different disks
    - For example 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/s is the same but every access reads 8 times as many data!
  - Block-level striping for blocks of files
  - Sector-level striping for sectors of a block

Disk scheduling

Disk scheduling: elevator scheduling

- Idea: schedule readings of requested blocks in the order in which they appear under the disk head.
- Elevator algorithm: works like an elevator
  - 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 a few requests

Buffering and prefetching

Buffering and prefetching

- Buffering:
  - Idea: keep as many blocks in memory as possible to reduce disk accesses.
  - Devise page replacement policies (e.g., LRU or MRU)
- Prefetching or double buffering:
  - Situation: needed data is known, however the timing is data-dependent.
  - Idea: speed-up access by pre-loading needed data.
  - Problem: requires extra memory

Short 5 mn break:

please stand up, stretch, and move around

Physical Data Organisation

- A table in a DBMS is stored as a collection of records, or a ***file***
  - Each file consists of one or more pages, and each page stores records of one type (i.e., from one table)
  - ***Disk space manager*** supports the concept of a ***page*** as a unit of data, and each page is stored as one or more disk blocks
    - We assume that *a page means a block* in this subsequent discussion

Example of Physical Data Organisation

Relation(tuplekey, attr, …)

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 attr of 4 bytes, and 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.

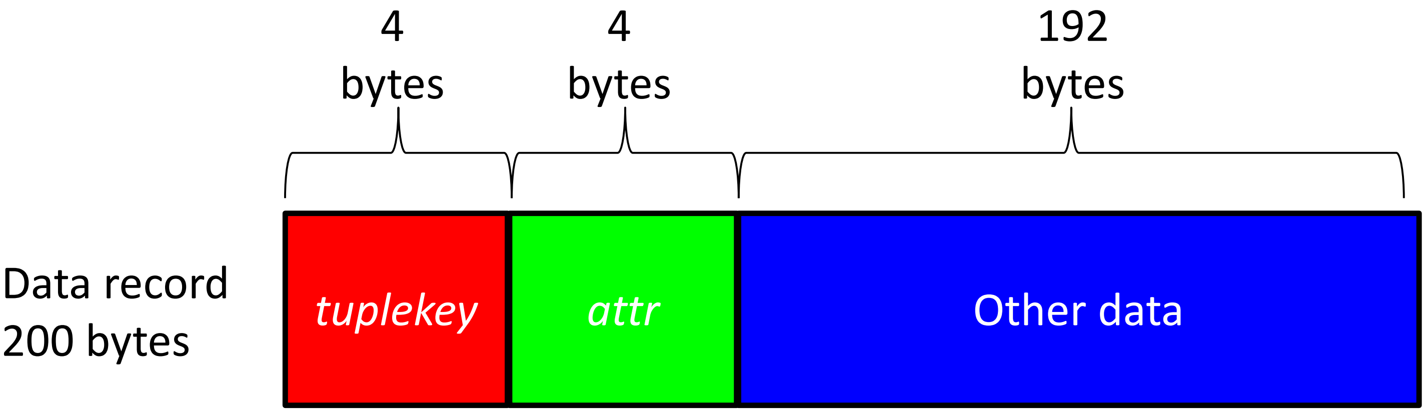
Questions:

- *How many bytes per record?*
- *How many records per page?*
  - Assume that no record is ever split across several pages.
- *How many pages for storing the table?*

How Many Bytes per Record?

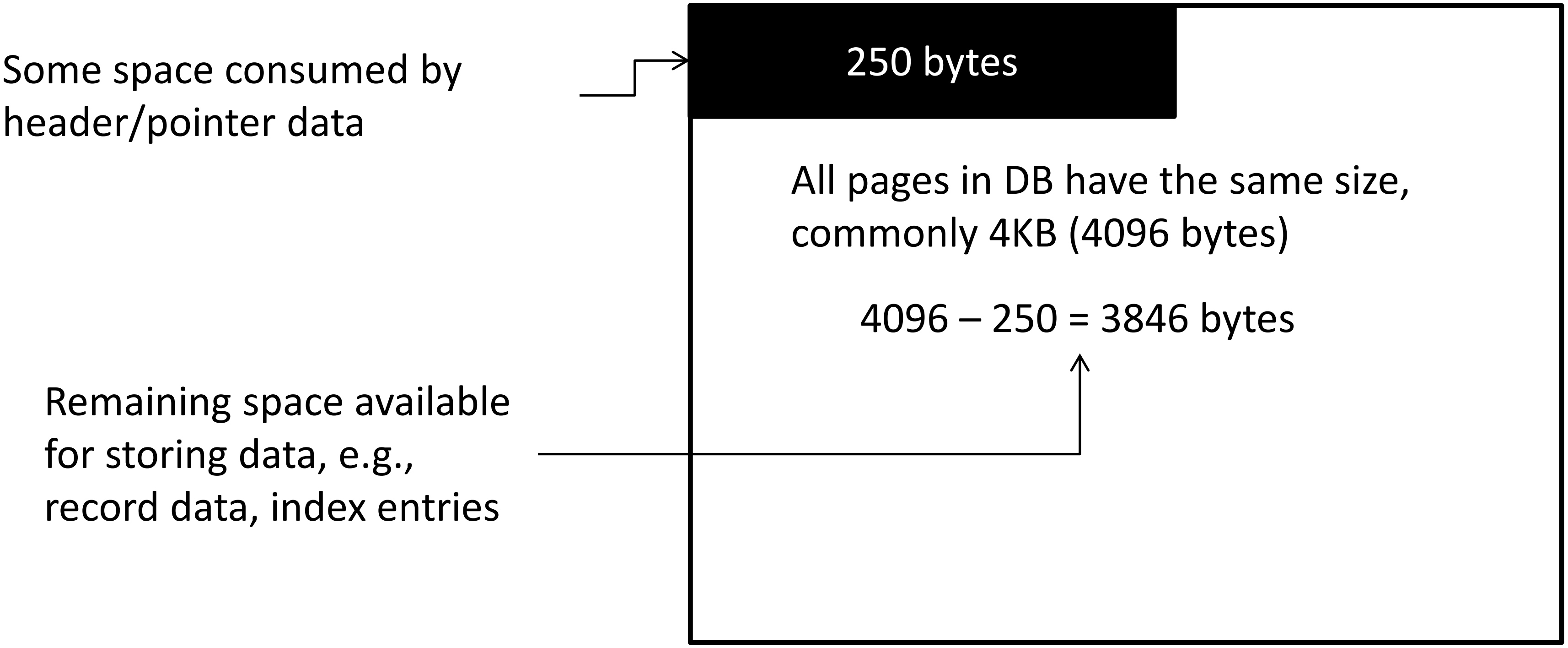
- Most databases store each tuple (row) in a record
  - All fields of a record are stored together consecutively

Relation(tuplekey, attr, …)



From earlier
"each record is 200 bytes long, including a primary key tuplekey of 4 bytes, an attribute attr of 4 bytes, and also other attributes"

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

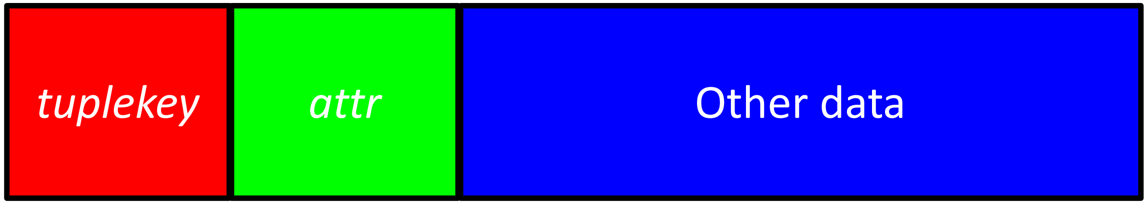
Remaining space available for storing data, e.g., record data, index entries

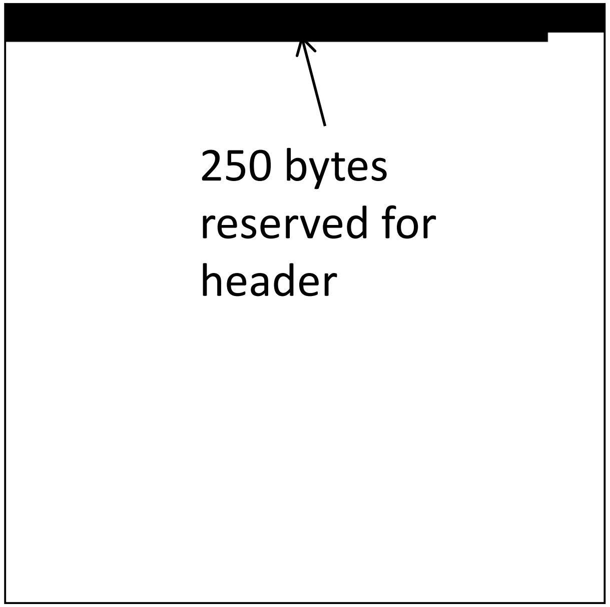
From earlier

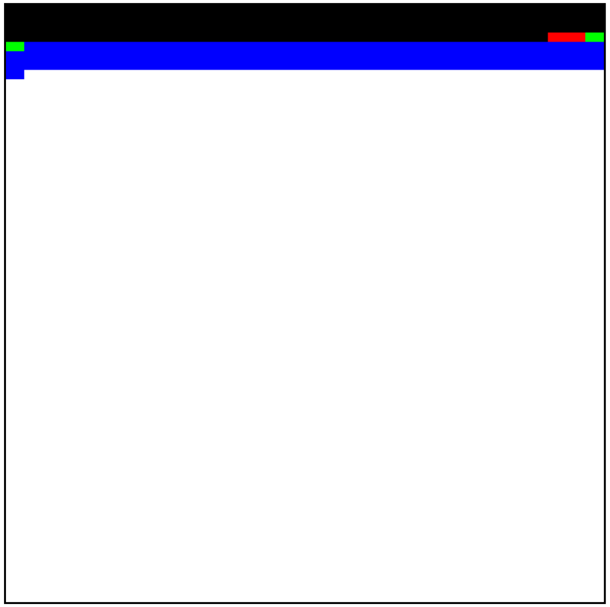
“each page is 4K bytes, of which 250 bytes are reserved for header and array of record pointers”

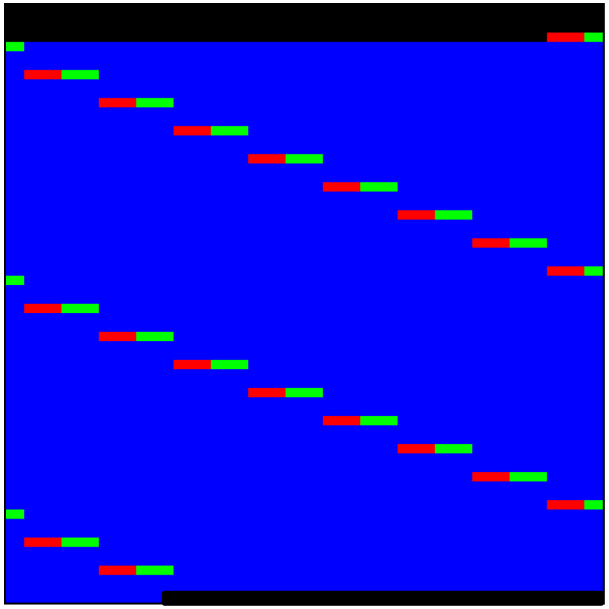
How Many Records per Page?

Each example data record 200 bytes




250 bytes reserved for header





# records: 0
empty space: 3846 bytes

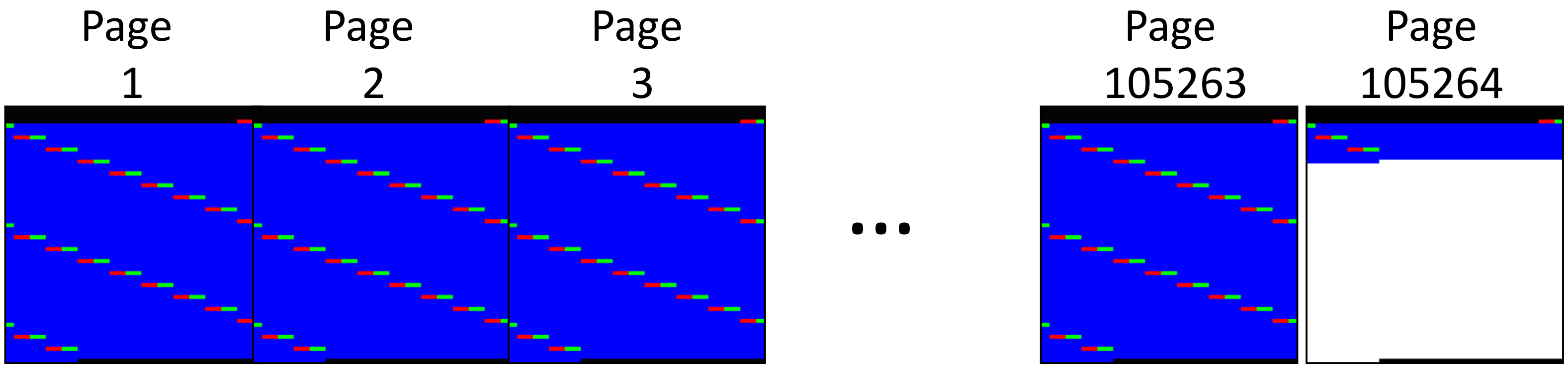
# records: 1
empty space: 3646 bytes

# records: 19
empty space: 46 bytes

$\lfloor 3846 \text{ bytes/page} \div 200 \text{ bytes/record} \rfloor = 19$ records/page

plus 46 remaining bytes

How Many Pages for the Table?



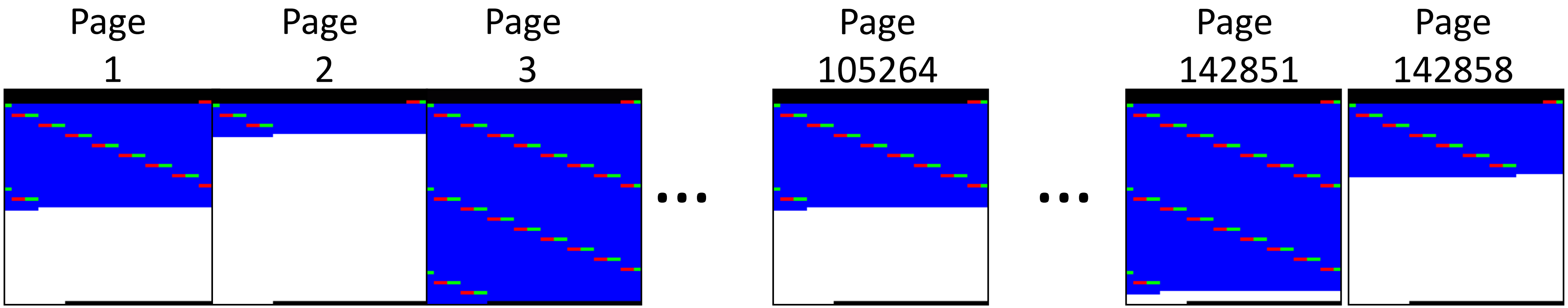
$$\frac{\text{2,000,000 records}}{\text{19 records/page}} = \text{105263 full pages, plus 3 remaining records}$$

Remaining 3 records must go into a further page, so 105264 pages in total

$$\text{105,264 pages} \times \text{4096 bytes/page} = \text{431,161,344 bytes}$$

$$\text{Overhead} = \frac{431{,}161{,}344 - 400{,}000{,}000}{400000000} = 7.79\% \text{ (~8\% overhead)}$$

How Many Pages for the Table When Pages Have Spare Capacity?



Assume pages are 75% full on average

19 records/page × 75% average occupancy = 14 records/page

$$\frac{2{,}000{,}000 \text{ records}}{14 \text{ records/page}} = 142{,}858 \text{ pages (rounded)}$$

142,858 pages × 4096 bytes/page = 585,146,368 bytes ➔ ~47% overhead

Typical File Organizations

File Organisation*:* a method of arranging the records in a file when the file is stored on disk.

› ***Unordered (Heap) Files*** – a record can be placed anywhere in the file where there is space (random order)

- suitable when 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

- can speed up searches by binary search, but updates can be very expensive

› ***Indexes*** – data structures to organize records via trees or hashing

- like sorted files, they speed up *searches for a subset of records*, based on values in certain ("search key") fields.
- updates are much faster than in sorted files.

Outline

- **Physical Data Organization: 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

Access Paths

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

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

- The choice of an access path to use in the execution of an SQL statement has no effect on the semantics of the statement (**Physical Data Independence)**
  - However, this choice can have a major effect on the execution time of the SQL statement

I/O Cost of Access Paths

- Data is permanently stored on disk, but must be read into main memory for the DBMS to operate on it

- The unit for data transfer between disk and main memory is a page
  - Even if a single record in a page is needed, the entire page is transferred
  - Reading or writing a page is called an **I/O** (input/output) operation

- The cost of page I/O dominates the cost of typical database computation operations
  - The cost of SQL execution usually is measured by the number of I/Os

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 the pages in a file must be read, in the worst case even the whole file

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;

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

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

- For **equality search**, 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 requires 70,176 I/Os
  - For **zero matching records** or **non-unique** attribute, it needs to check every record, so requires 140,351 I/Os

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

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

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

Drawback of Linear Scan

- Linear scan is **extremely** slow for **large** tables
  - Consider a table of 1 PetaByte (PB, 10^{15} bytes). Assuming the fastest Solid-State Drives (SSD) with disk scanning speed of 6GB/s, how much time do we need to scan the entire table?

$$\frac{10^{15}}{6 \times 10^{9} \times 3600 \times 24} = 1.9 \text{ days}$$

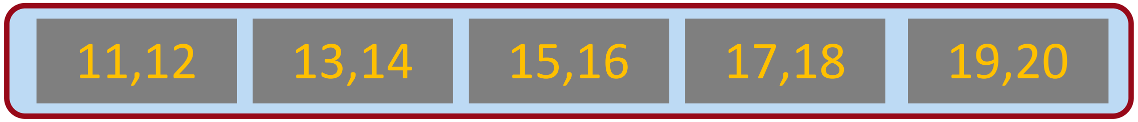
  - a linear scan of this table takes 1.9 days!

Access Method for Sorted Files

Speeding up query processing (first attempt)

› Store records in a sorted file based on some attribute(s)

- 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***

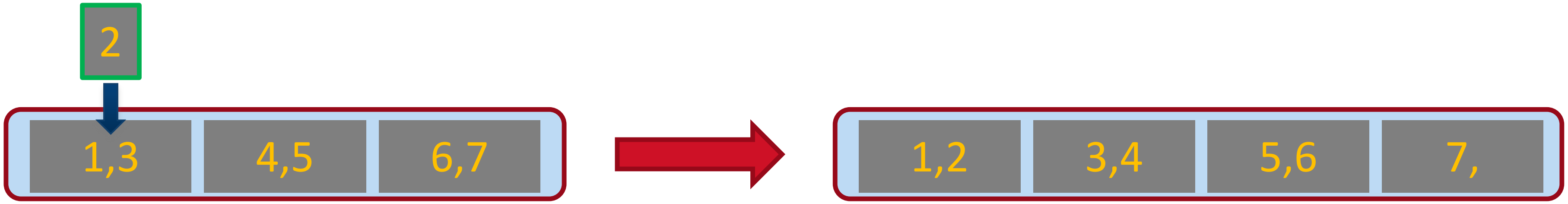
- The I/O cost (worst case scenario) is *log_2B + the number of pages containing retrieved records*

  (B is the total number of pages in the file)

  - *log_2B* for retrieving the page containing first record (*$log_2 140,351 = 18$*)
  - Successive records are in same (or successive) page(s)

Drawback of Sorted Files

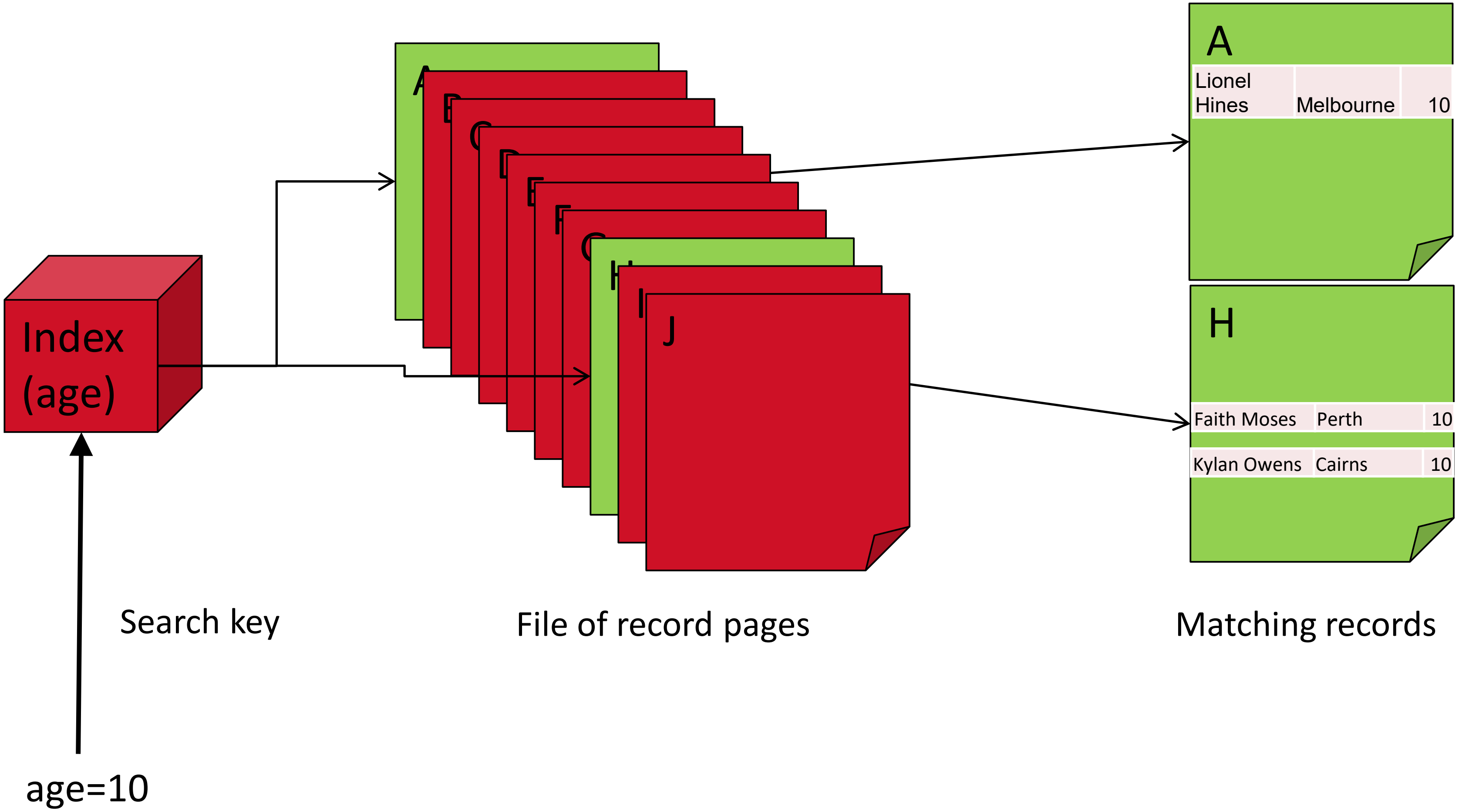
- It is expensive to maintain the sorted order when records are inserted
  - After the correct position for an insert has been determined, it needs to shift all subsequent records to make space for the new record
    - In a sorted file, all pages are required to be consecutive



  - Hence sorted files typically are not used per-se by DBMS, but rather in the form of index-organised (clustered) files

Speeding Up Query Processing via Indexing

- In order to speed up a query (i.e., reduce time), we add additional information to facilitate query answering



Lionel Hines	Melbourne	10
Faith Moses	Perth	10
Kylan Owens	Cairns	10

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** the same as *key of a relation*

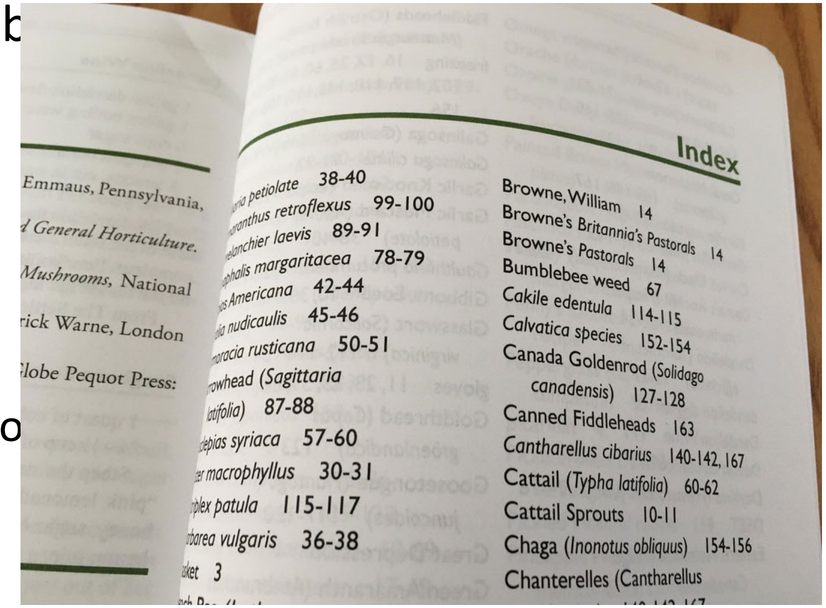
Product(name, maker, price)

› An index provides efficient lookup & retrieval b

- 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, and can also



Emmaus, Pennsylvania,

d General Horticulture.

Mushrooms, National

rick Warne, London

lobe Pequot Press:

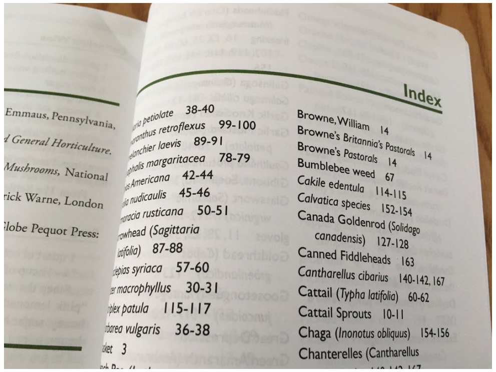
Index

ria petiolate 38-40
aranthus retroflexus 99-100
elanchier laevis 89-91
aphalis margaritacea 78-79
as Americana 42-44
alia nudicaulis 45-46
moracia rusticana 50-51
rowhead (Sagittaria
latifolia) 87-88
clepias syriaca 57-60
ter macrophyllus 30-31
riplex patula 115-117
rbarea vulgaris 36-38
sket 3

Browne, William 14
Browne's Britannia's Pastorals 14
Browne's Pastorals 14
Bumblebee weed 67
Cakile edentula 114-115
Calvatica species 152-154
Canada Goldenrod (Solidago
canadensis) 127-128
Canned Fiddleheads 163
Cantharellus cibarius 140-142, 167
Cattail (Typha latifolia) 60-62
Cattail Sprouts 10-11
Chaga (Inonotus obliquus) 154-156
Chanterelles (Cantharellus

Indexing - The 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 can become quite costly



Emmaus, Pennsylvania,
d General Horticulture.
Mushrooms, National
rick Warne, London
lobe Pequot Press:

Index

ria petiolate 38-40
aranthus retroflexus 99-100
elanchier laevis 89-91
aphalis margaritacea 78-79
os Americana 42-44
lia nudicaulis 45-46
moracia rusticana 50-51
rowhead (Sagittaria
latifolia) 87-88
clepias syriaca 57-60
ter macrophyllus 30-31
riplex patula 115-117
rbarea vulgaris 36-38
sket 3

Browne, William 14
Browne's Britannia's Pastorals 14
Browne's Pastorals 14
Bumblebee weed 67
Cakile edentula 114-115
Calvatica species 152-154
Canada Goldenrod (Solidago
canadensis) 127-128
Canned Fiddleheads 163
Cantharellus cibarius 140-142, 167
Cattail (Typha latifolia) 60-62
Cattail Sprouts 10-11
Chaga (Inonotus obliquus) 154-156
Chanterelles (Cantharellus

Nevertheless, indexing is used in all DBMSs to improve query performance.

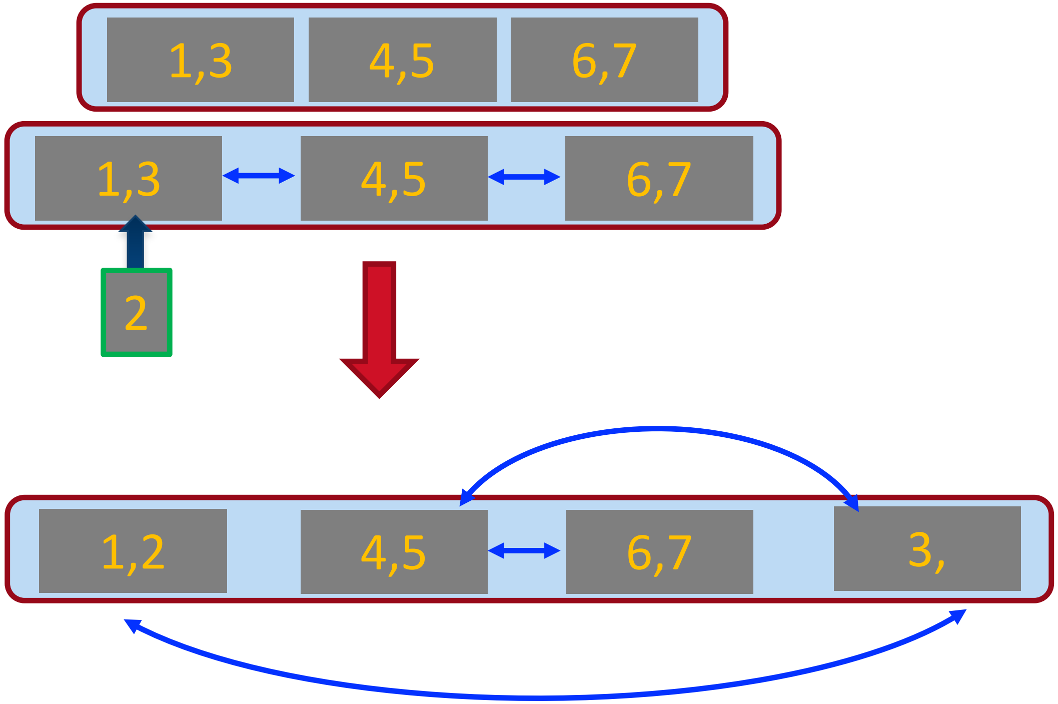
Indexing is one of the most important features provided by a DBMS for performance.

Outline

- **Physical Data Organization: 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

General Idea of B+ Tree

- Maintain a **logically** sorted file to cope with insertions/deletions



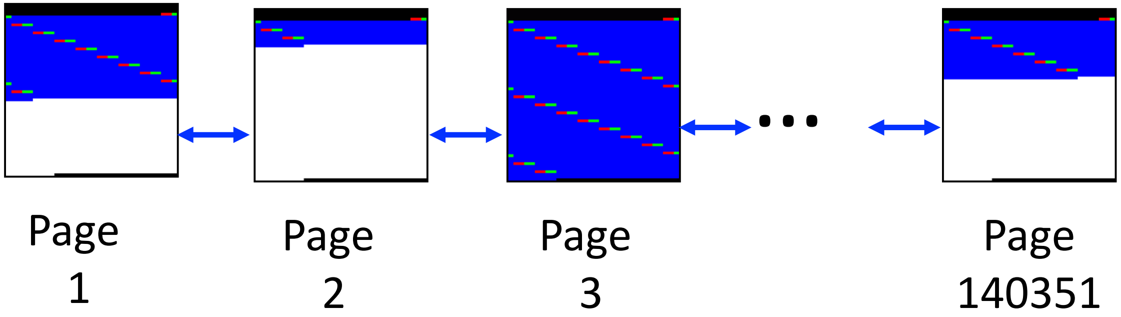
- **Bad News**: now this file is not physically sorted, we cannot do binary search (e.g., we can do binary search on an array, but not on a linked list)
  - Remedy: we build an index for the data file

Step 1: Create a Logically Sorted Data File

› The table Relation is stored in a ***data file*** consisting of 140351 pages with search key tuplekey

- For each page, all *records in a page* are *physically sorted* on the search key (let us assume in increasing order)
- Pages are *logically sorted*: all records in page *i* have smaller search key values than all records in page *i+1*
  - These pages *may not be consecutive* on disk
- **We call this file as logically sorted for search key**

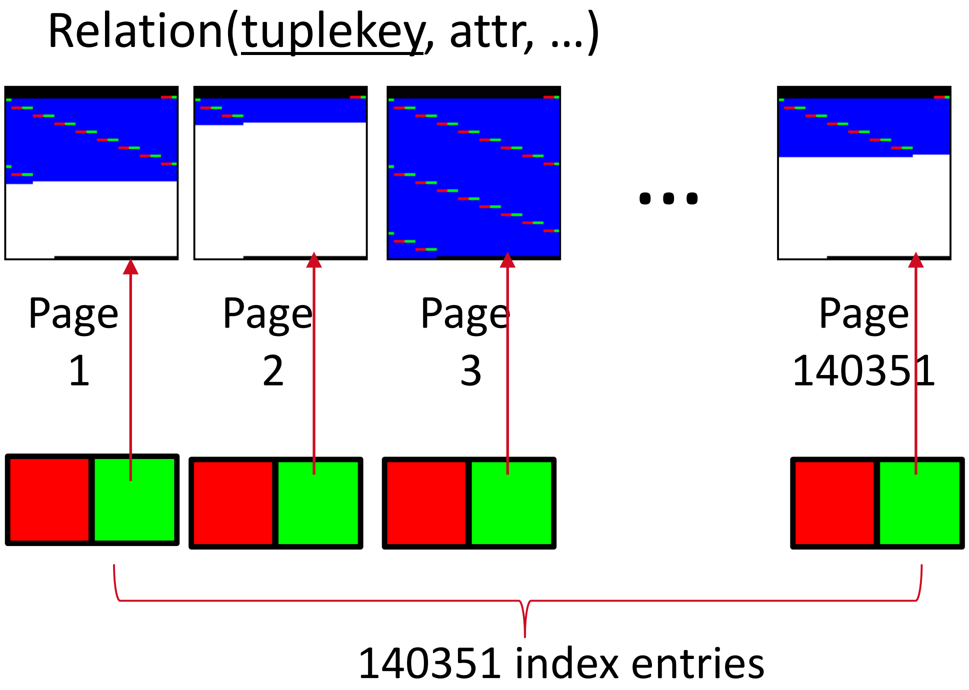
Relation(tuplekey, attr, …)



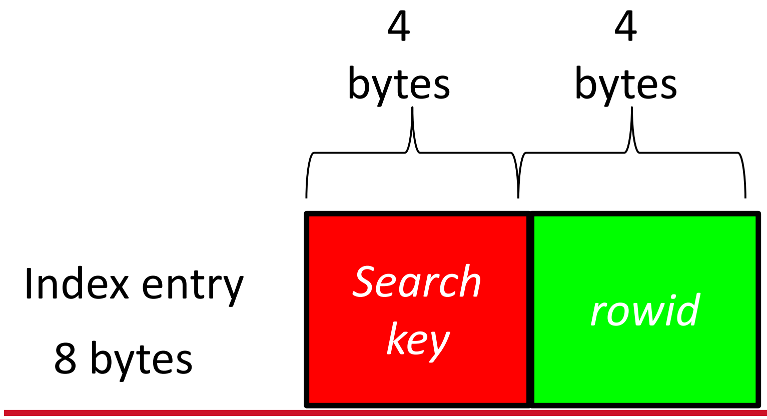
Step 2: Create One Index Entry for Each Record Page

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

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



Relation(<u>tuplekey</u>, attr, …)

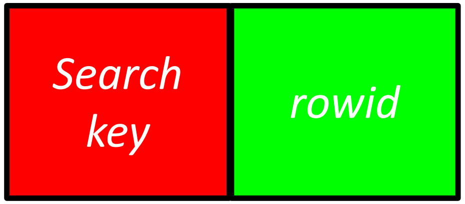


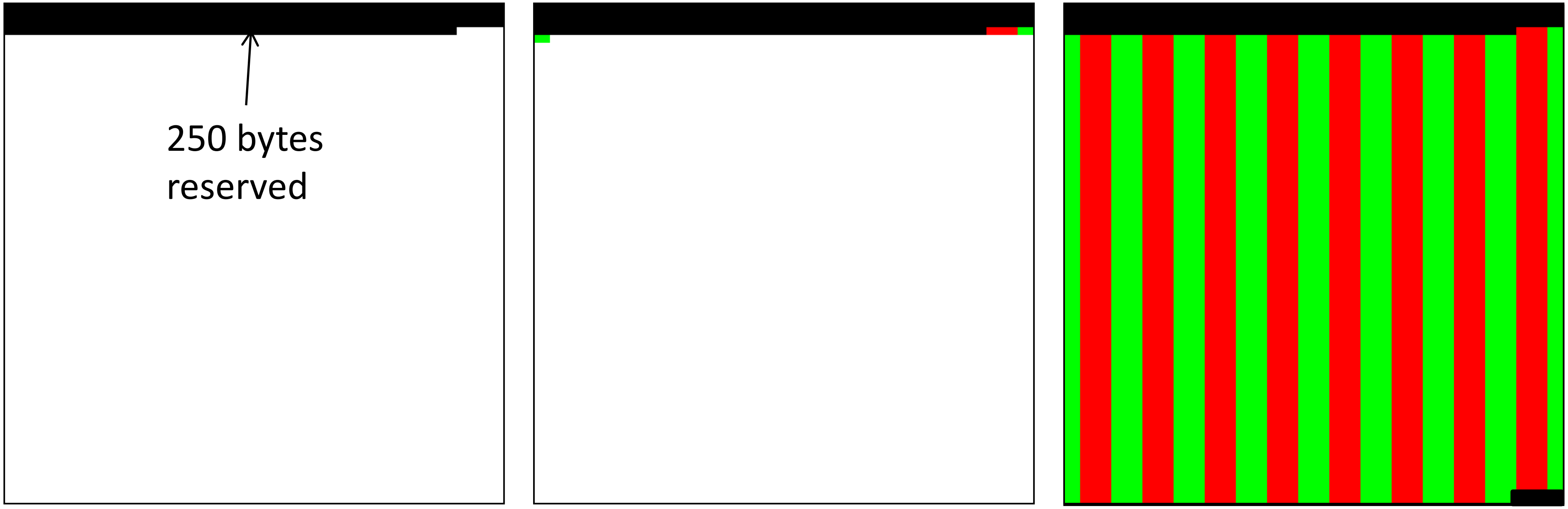
Index on tuplekey, so search key same size as tuplekey.

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

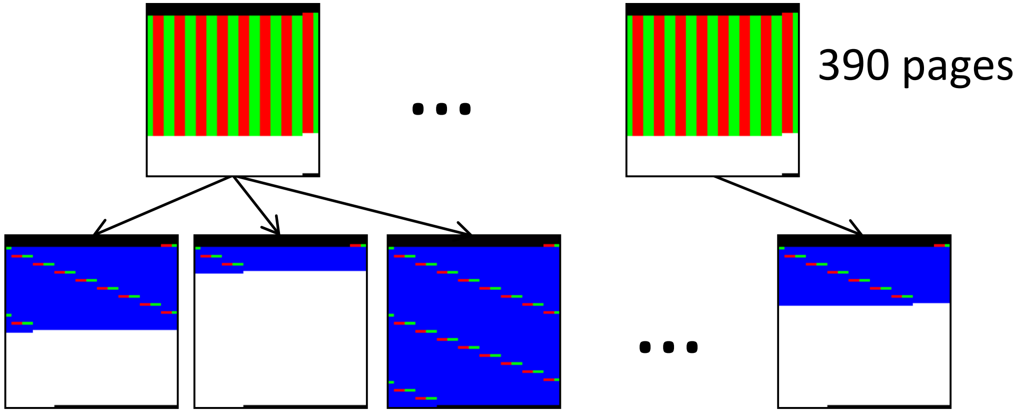
# entries: 480
empty space: 6 bytes

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

plus 6 remaining bytes

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

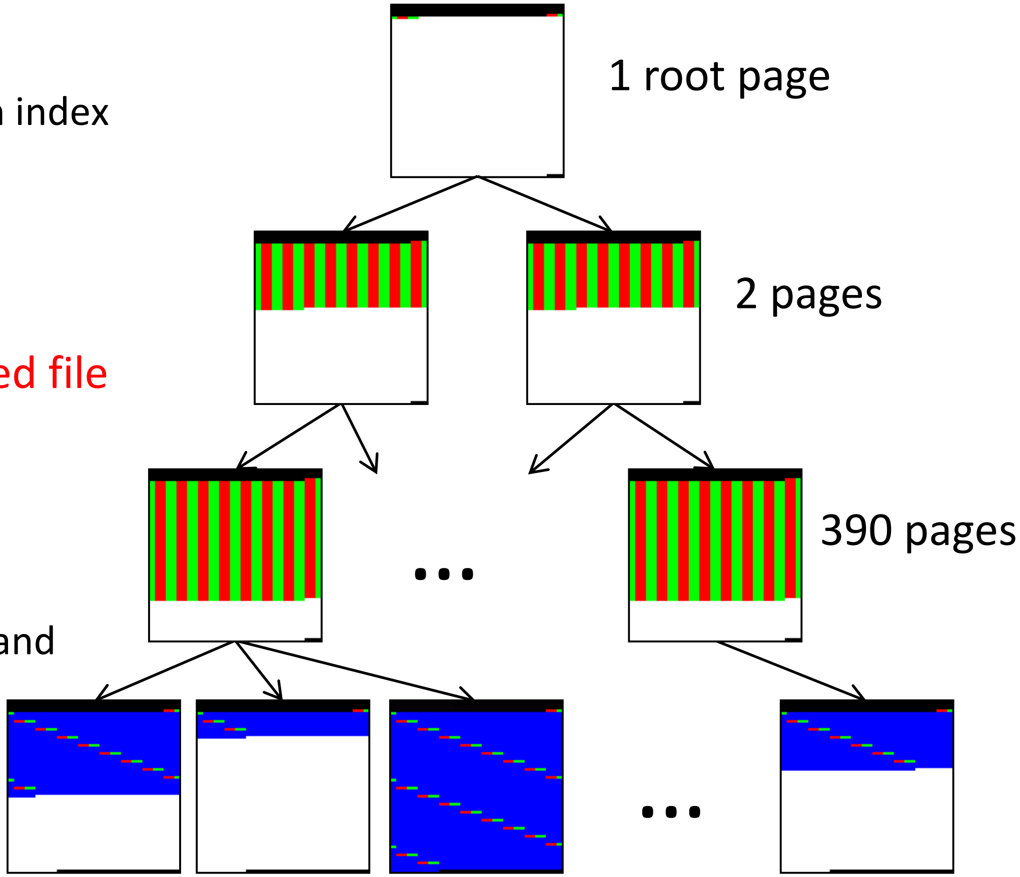
- Let us call a page storing index entries an ***index page***
  - Recall each index page holds up to 480 index entries
  - Assume average occupancy of 75%
  - The average number of index entries in an index page is 480 × 75% = 360
- The number of index pages is 140351/360 = 390 (rounded up)
- We could store these 390 index pages as a sorted file
  - Or we could store them as a logically sorted file and recursively build indexes on it



Leaf level:140,351 pages

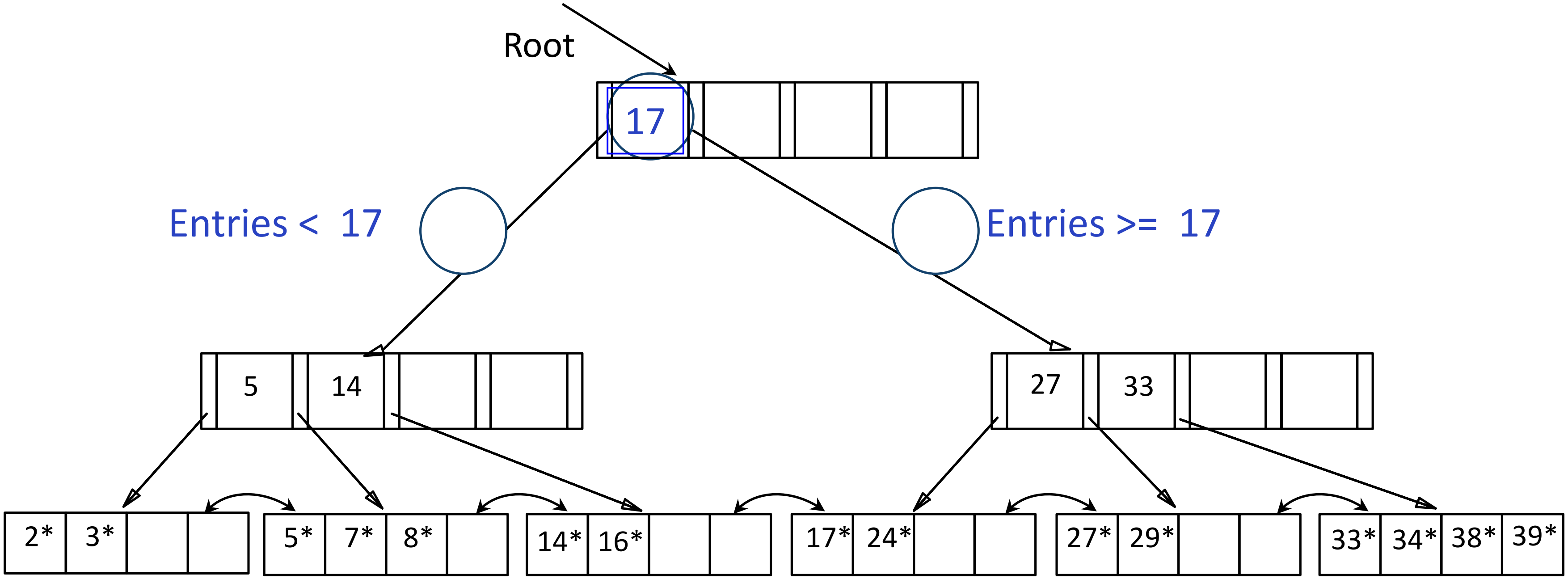
Step 4: Build B+ Tree Index

- Let us call a page storing index entries an ***index page***
  - The average number of index entries in an index page is 480 × 75% = 360
  - The number of first-level index pages is 140351/360 = 390 (rounded up)
- We could store them as a <u>logically</u> sorted file and recursively build indexes on it
  - The next-level index consists of 390 index entries, and 390/360 = 2 pages
  - The root level consists of 2 index entries, and 2/360 = 1 page
- 390 + 2 + 1 = 393 index pages
- 393/140351 = 0.2% increase



Leaf level:140,351 pages

Searching in a B+ Tree Index

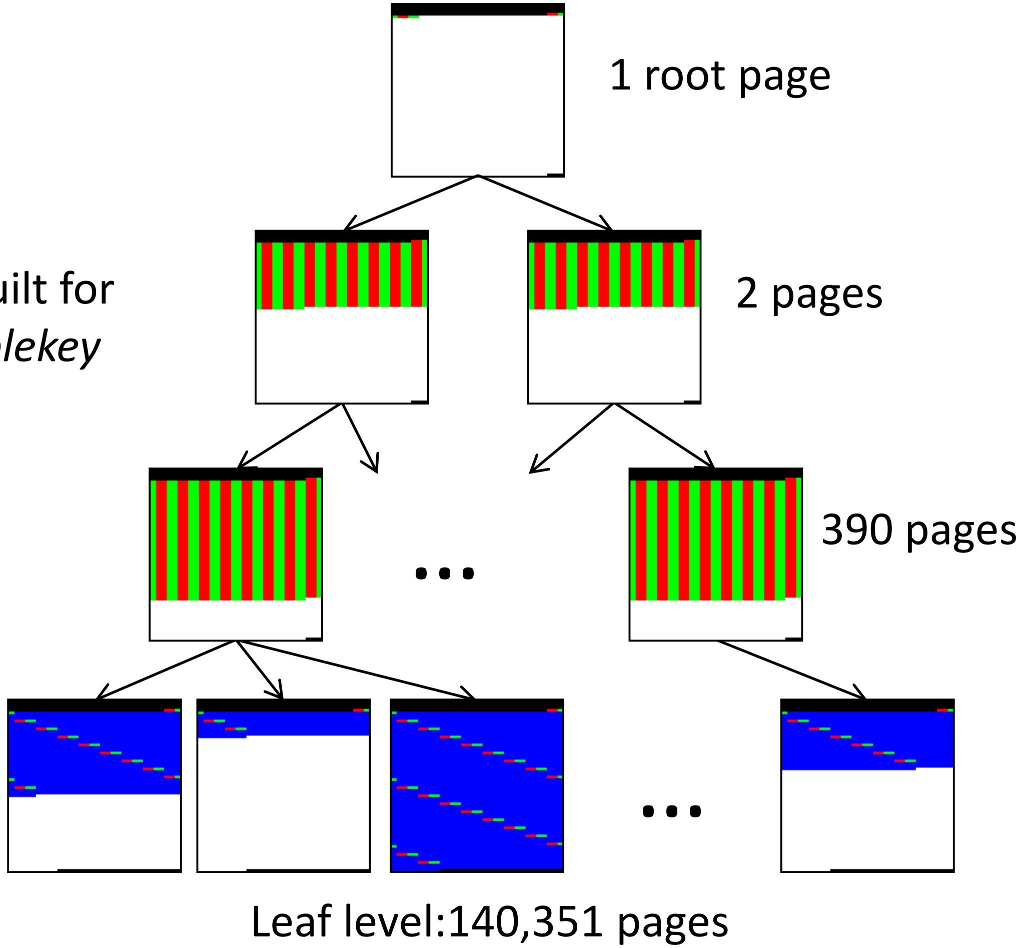


- Note how **record/data pages in the leafs** are sorted and linked
- Find 14? 29? All values >20 and <30?
- The cost of searching in a B+ tree index: *the lookup cost + number of pages containing retrieved records*
  - Lookup cost is the number of levels of the B+ tree excluding the level of record/data 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

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

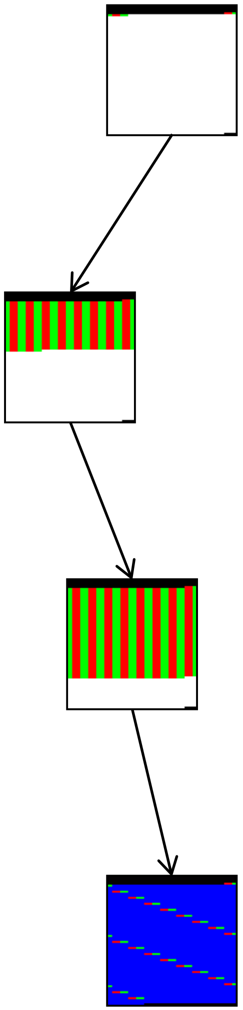
The B+ tree built for *search key tuplekey*



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

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

- For **equality search** on tuplekey, we can use 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** ***attr*** **BETWEEN** 100 **AND** 119;

- For **range search** on attr, this index is no use, so still need to check each record page sequentially, so 140,351 I/Os. Need to *build another index*.

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: the search key for an index 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)

An index with a search key can be used to lookup a search condition, if for all legal table instances, all records satisfying the search condition are consecutive in the sorted order of the table’s records based on the 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!

THE UNIVERSITY OF
SYDNEY