COMP9120 Database Management Systems

Week 10: Storage & Indexing

bryn.jeffries@sydney.edu.au

Dr. Bryn Jeffries
School of Information Technologies

Topics

- How is data stored in a database?
  - File organisations
    - Heap files
- How does a database retrieve it?
  - Access Paths
    - File scans
    - B+ tree index
    - Hash index
- Focus on dealing with simple queries on an individual relation.
- More sophisticated queries involving multiple tables are considered next week (Query Evaluation)

Based on slides from Ramakrishnan/Gehrke "Database Management Systems"

How to Store a Database?

- Logical Database Level:

Relation

```
CREATE TABLE MovieStar (
   name      CHAR(30) PRIMARY KEY,
   address   VARCHAR(255),
   gender    CHAR(1),
   birthdate DATE
);
```

Fields (attributes)

```
INSERT INTO MovieStar VALUES (‘Brad Pitt’,
‘Sunset Boulevarde’, ‘m’, ‘18/12/1963’);
```

Record (tuple)

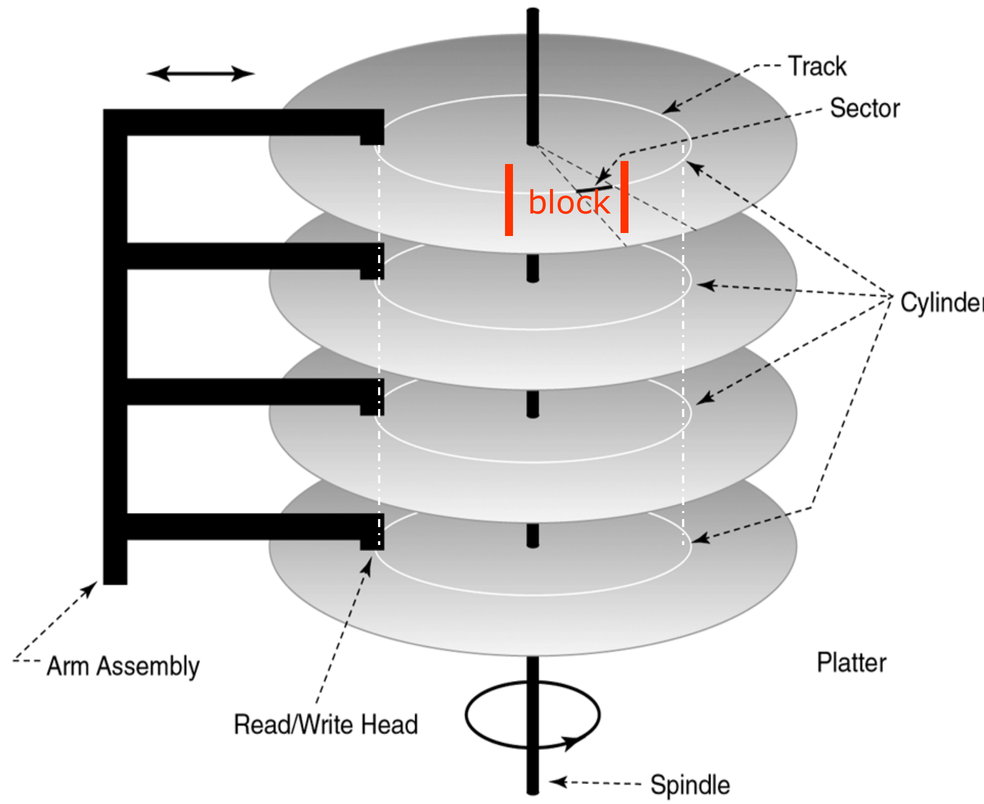
- Physical Database Level:
  - How do we represent SQL data types?
  - How to represent tuples with several attributes (*fields*)?
  - How to represent collection of tuples?

Motivation: Disk Storage

- DBMS stores information on ("hard") disks.
  - *Main memory is much more expensive than HDDs.*
  - *Main memory is volatile.*
    We want data to be saved between runs. (Obviously!)
- Implications for DBMS design:
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!
- Typical storage hierarchy:
  - Main memory (RAM) for data in use.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).

Physical Hard Disk Structure

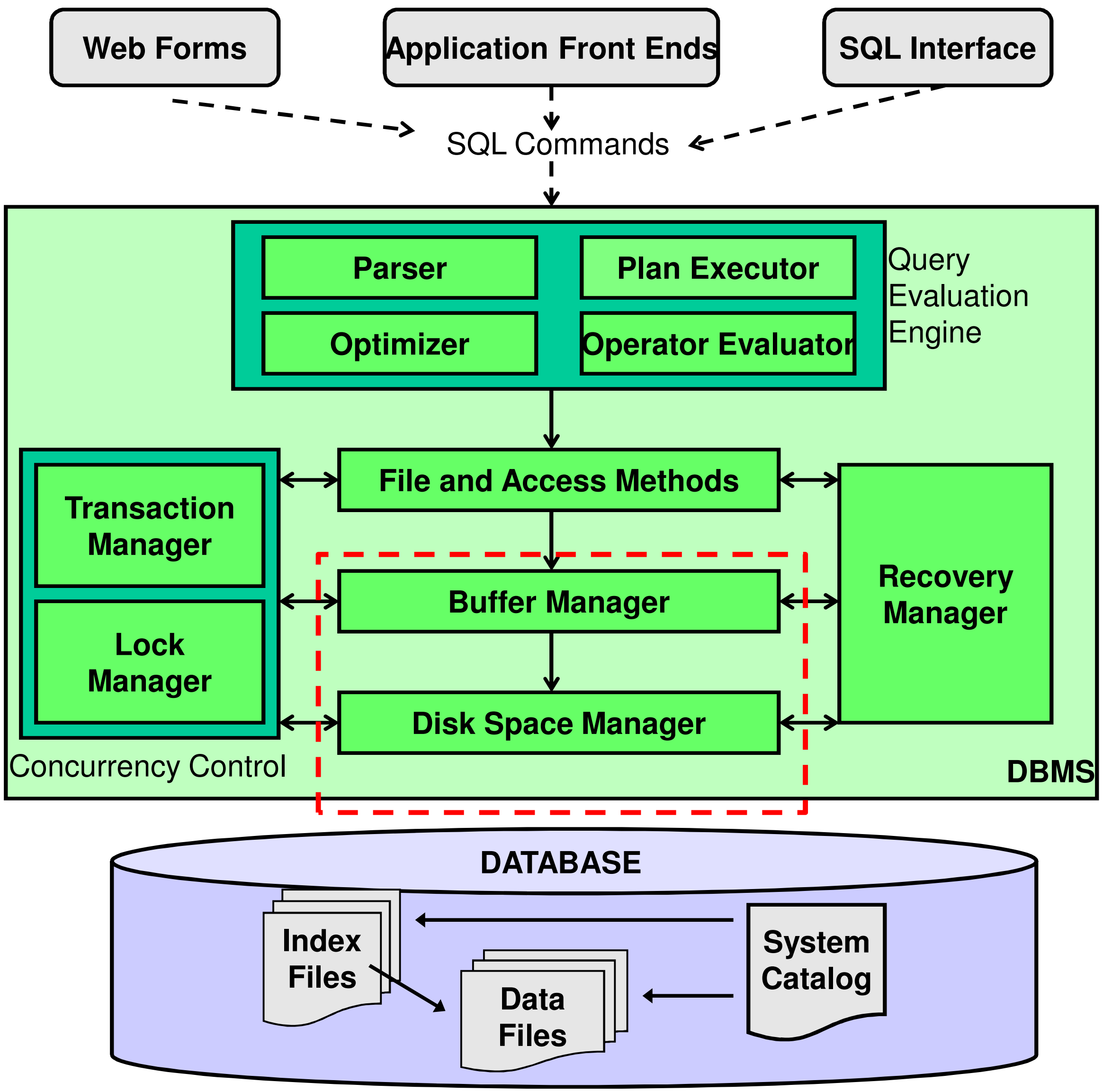
- The platters spin (e.g., 15000 rpm) – **rotational delay**.
- The arm assembly is moved in or out to position a head on a desired track. (**Seek time** 1 – 20 ms)
- Only one head reads/writes at any one time.
- **Block size** is a multiple of sector size (which is fixed).
  - typically 4kB or 8kB
- **Transfer time** approx. 0.03 ms per block



Physical organization of a disk storage unit.

[Kifer/Bernstein/Lewis, 2006]

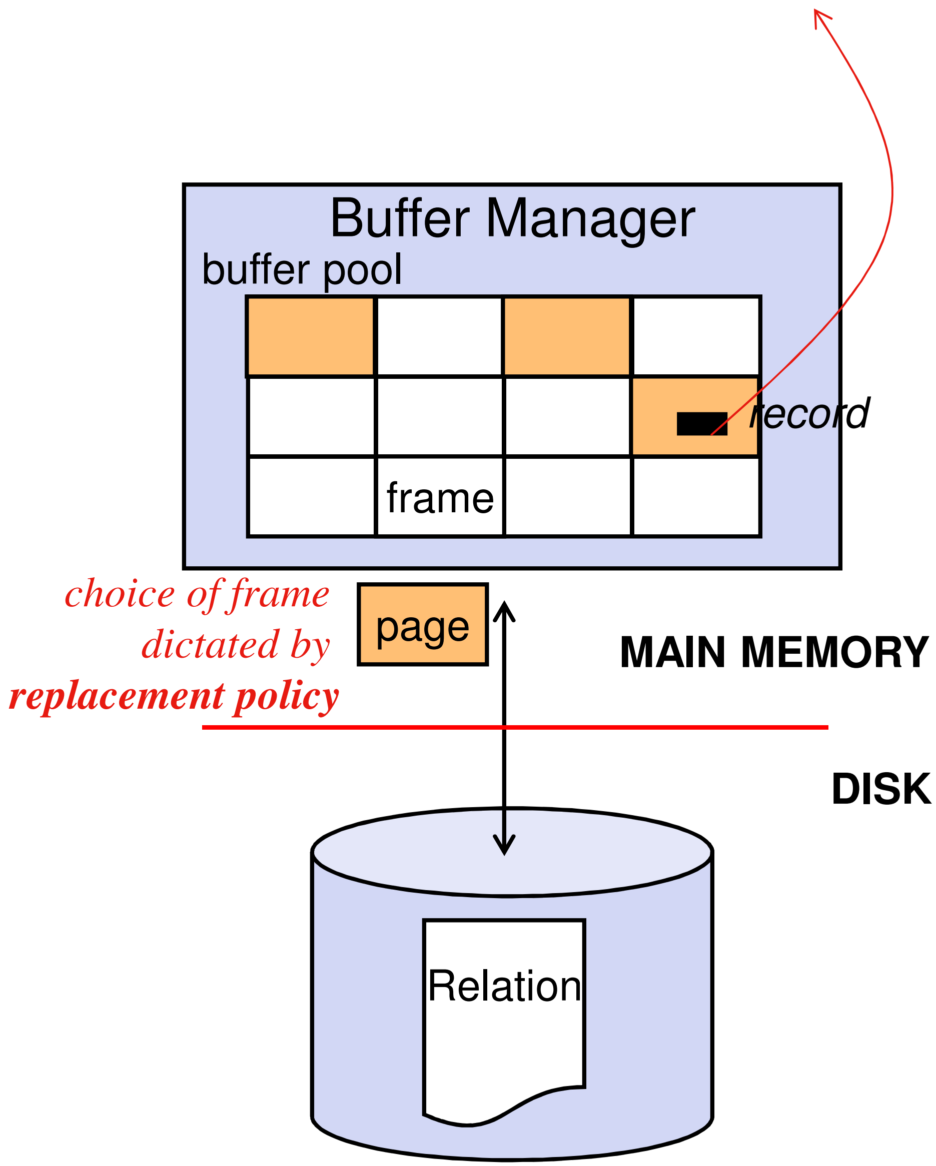
Internal Structure of a DBMS



Buffer Manager

- DBMS calls the buffer manager when it needs a page from disk.

1. If the page is already in the buffer, the address of the page in main memory is returned
2. If the page is not in the buffer,
   a. the buffer manager chooses an empty frame if possible.
   b. if all frames are used, replaces (throwing out) some other page
      - Depends upon Buffer Replacement Policy
      - If the page that is thrown out was modified (marked 'dirty'), it is written back to disk.
   c. Once a frame is allocated in the buffer, the buffer manager reads the page from the disk.



HOW do we store the data?

PAGE AND FILE LAYOUTS

Physical Data Organisation

- Due to the high access latency, we organise data in form of **data pages** on secondary storage
- A relation can be treated as set of pages of records
- But how are individual records stored on these pages?

- Two approaches:
  - ***Fixed-length records***
    - assumes record size is fixed
    - each file has records of one particular type only
    - different files are used for different relations
  - ***Variable-length records***
    - record types that allow variable lengths for one or more fields.

Typical File Organizations

Many alternatives exist, each ideal for some situations, and not so good in others:

- **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 sequential order, based on the value of the search key of each record
  - best if records must be retrieved in some order, or only a `*range*' of records is needed.
- **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.

Exercise: Relation size

Relation(<u>tuplekey</u>, attr, …)

Record schema: assume that each record is 200 bytes long, including a unique key *tuplekey* of 4 bytes, and another attribute *attr* which is also 4 bytes.

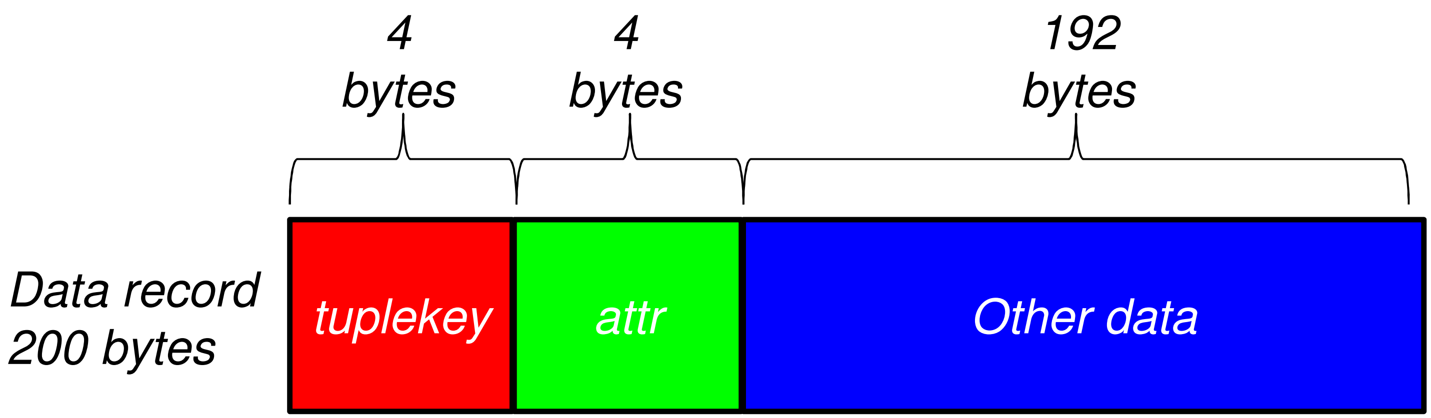
Relation size: there are 2,000,000 records in the relation, among which there are 10,000 different values for *attr*.

General features: assume that each page is 4K bytes, of which 250 bytes are taken for header and array of record pointers. Assume that no record is ever split across several pages.

- *How many bytes per record?*
- *How many records per page?*

Storage of individual records: example

- Most databases store records in rows
  - Eg see [Oracle Database Concepts]
  - Cf Column-store database

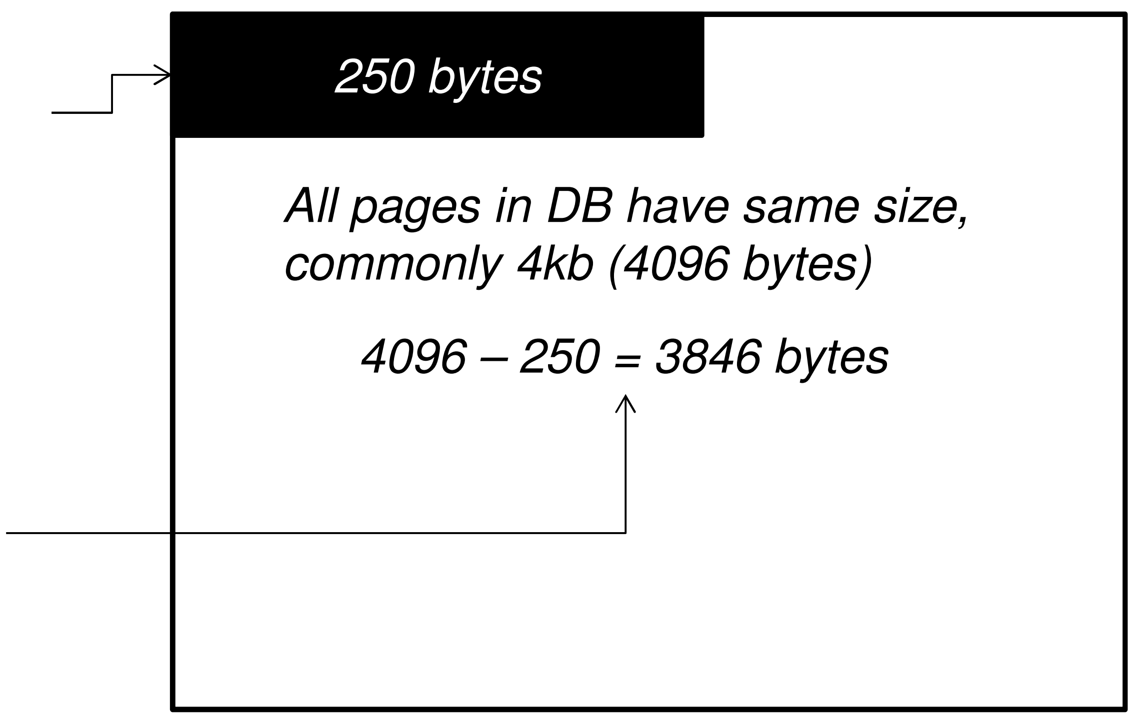


From earlier
*"Record schema**:** assume that each record is 200 bytes long, including a unique key tuplekey of 4 bytes, and another attribute attr which is also 4 bytes."*

RDMS data is stored within pages

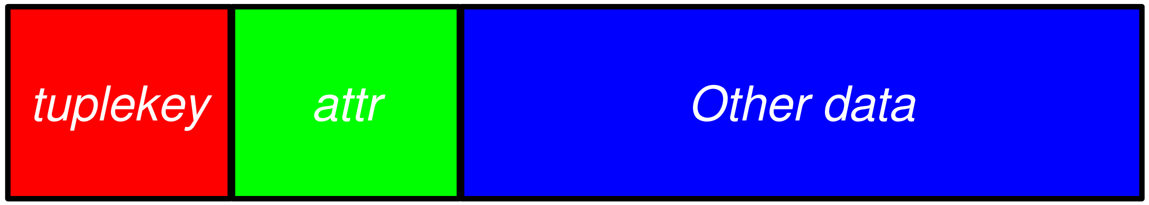
Some space consumed by header/pointer data

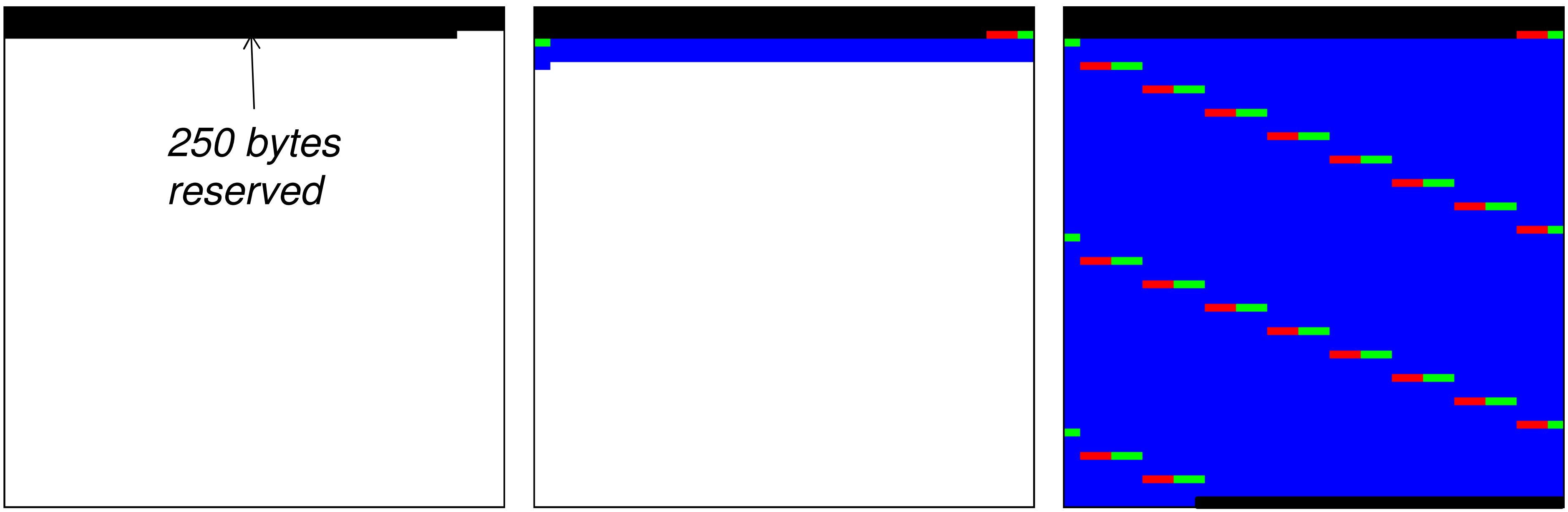
Remaining space available for storing data, e.g., record data, index entries



Fitting record data into pages

Each example data record 200 bytes





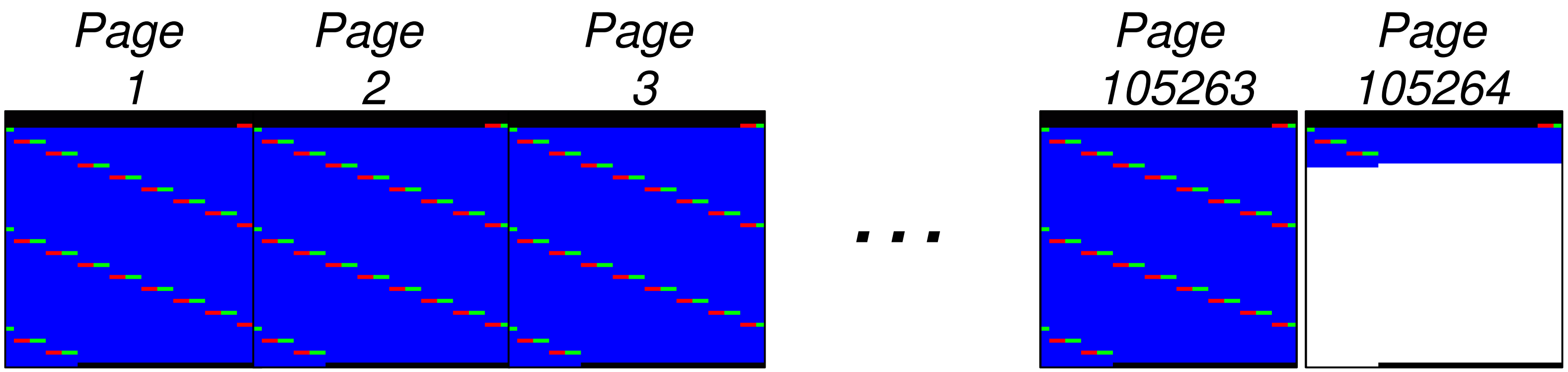
records: 0
empty space: 3846 bytes

records: 1
empty space: 3646 bytes

records: 19
empty space: 46 bytes

$\lfloor$3846 bytes/page ÷ 200 bytes/record$\rfloor$= 19 records/page plus 46 remaining bytes

Holding a whole relation in pages

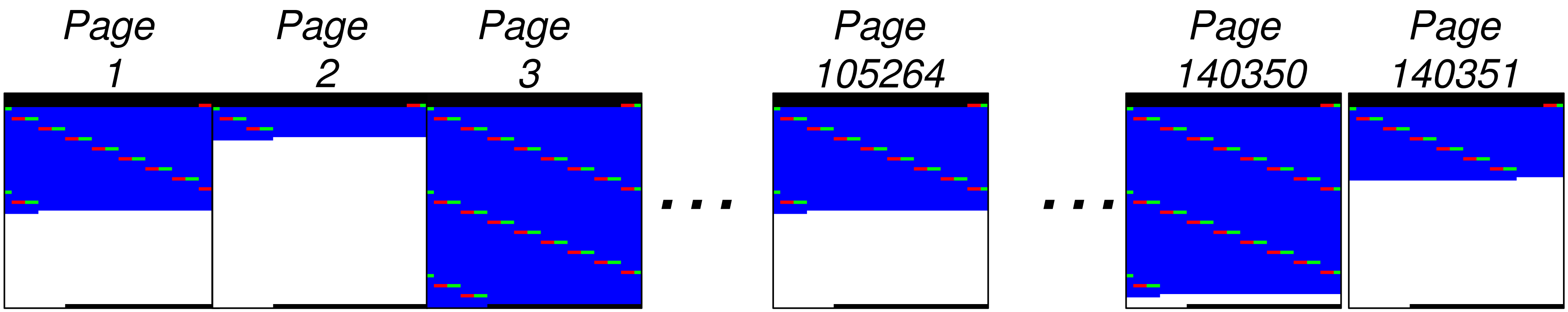


$$\frac{2{,}000{,}000 \text{ records}}{19 \text{ records/page}} = 105263 \text{ full pages, plus 3 remaining records}$$

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

105,264 pages × 4096 bytes/page = 431,161,344 bytes (8% overhead)

Pages have spare capacity



$$\frac{\text{2,000,000 records}}{\text{19 records/page} \times \text{75\% average occupancy}} = \text{140351 pages (rounded)}$$

140,351 pages × 4096 bytes/page = 574,877,696 bytes (44% overhead)

Sorted File

- Rows are sorted based on some attribute(s)
  - Access method is ***binary search***
  - Equality or range query based on that attribute has cost $log_2 B$ to retrieve page containing first row
  - Successive rows are in same (or successive) page(s) and cache hits are likely
  - By storing all pages on the same track, seek time can be minimized
- Problem: Maintaining sorted order
  - After the correct position for an insert has been determined, shifting of subsequent tuples necessary to make space (very expensive)
  - Hence sorted files typically are not used per-se by DBMS, but rather in form of index-organised (clustered) files (cf. next chapter)

Unordered (Heap) Files

- Simplest file structure contains records in no particular order.

- Access method is a ***linear scan***
  - In average half of the pages in a file must be read,
    in the worst case even the whole file
    - Efficient if all rows are returned (SELECT * FROM *table*)
    - Very inefficient if a *few* rows are requested

Exercise: Relation Statistics

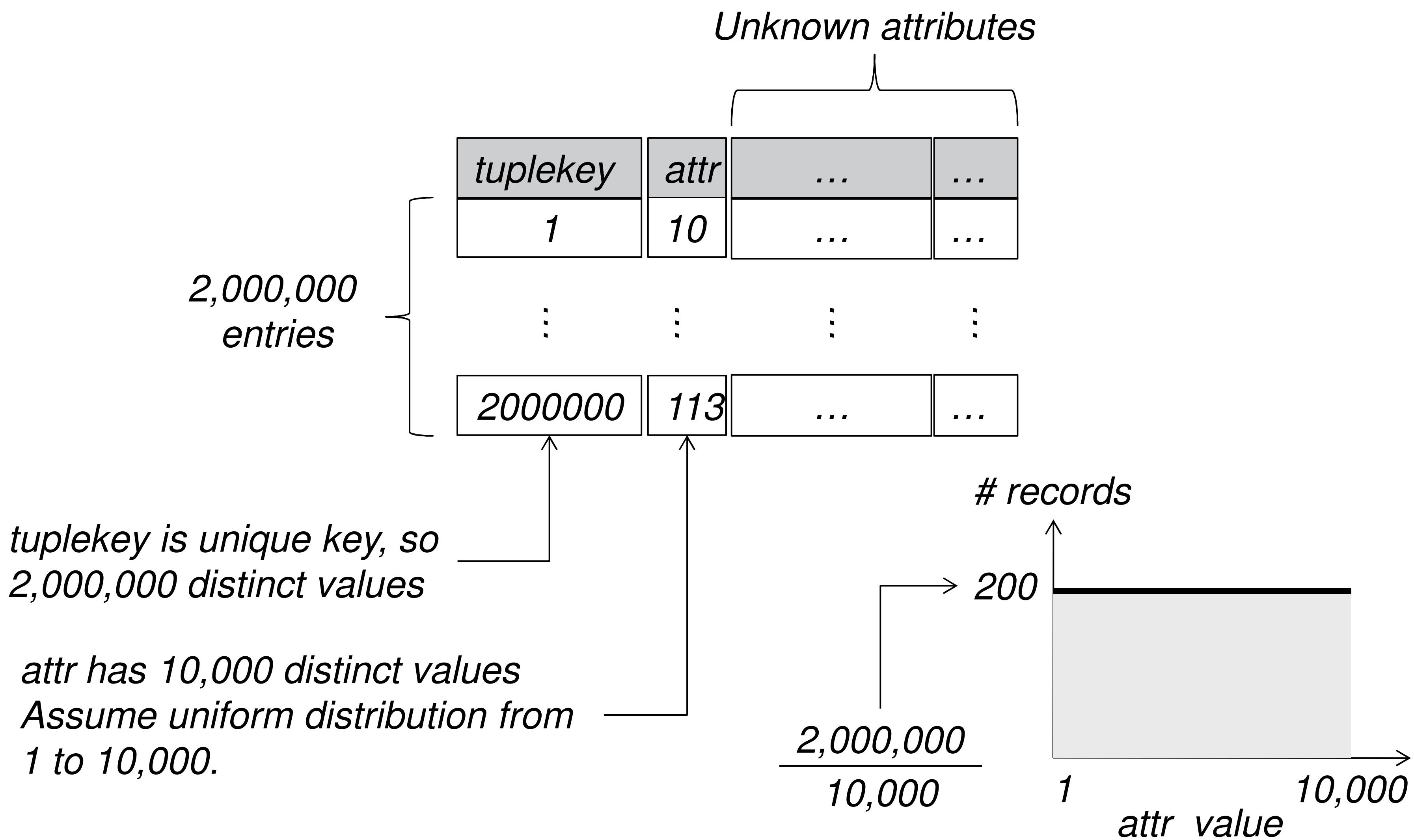
Relation(<u>tuplekey</u>, attr, …)

Relation size: there are 2,000,000 records in the relation, among which there are 10,000 different values for *attr*.

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

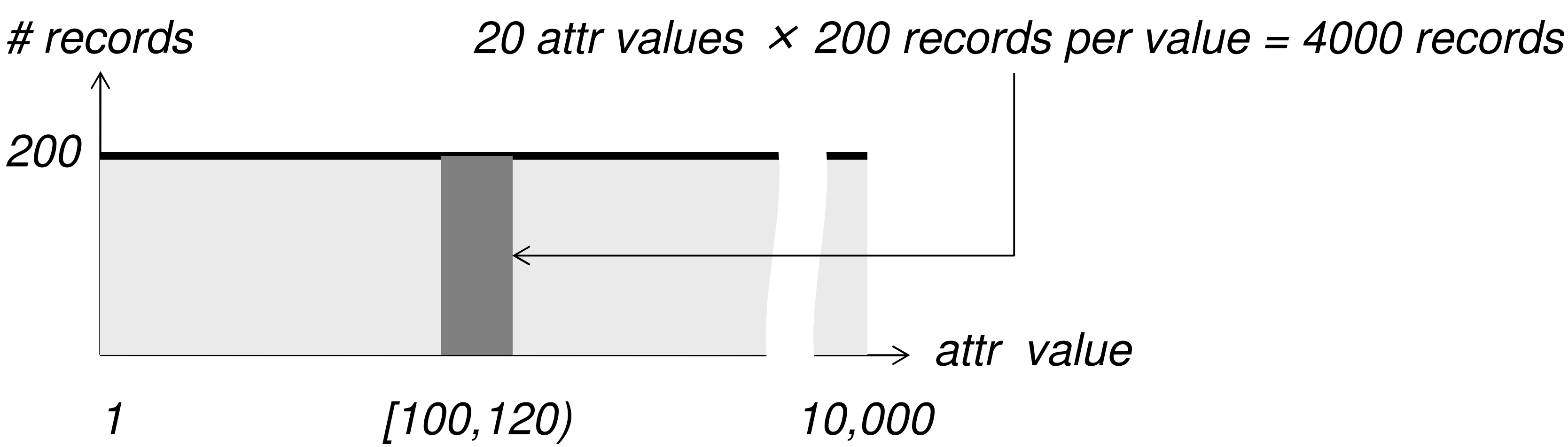
Assuming a uniform distribution of values, how many results do you expect to receive from this range query?

Relation statistics visualized



Range Query solution

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



Access Paths

- An **Access Path** is a route to our data
- Refers to the algorithm + data structure (*e.g.,* an index) used for retrieving and storing data in a table
- 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)**
- This choice can have a major effect on the execution time of the SQL statement
- Simplest access path is a linear scan (TABLE SCAN) of the records, reading each page in turn.

Exercise: Finding records in a heap file

How many pages (out of 140351) to find records in:

- `SELECT * FROM Relation WHERE tuplekey=715`
- `SELECT * FROM Relation WHERE attr BETWEEN 100 AND 119`

- Data is unsorted, use sequential search:

  For each page:

  1) Load page into page buffer (cost 1 I/O);

  2) Check each record in page for match;

- For **equality search**, *tuplekey* is **unique**, so can terminate on first match. If a matching record is present <u>on average</u> will have to look through half of all pages so require 70,176 I/Os
- For **zero matching records** or **non-unique** attribute need to check every record, so require 140,351 I/Os
- For **range search** need to check each record, so 140,351 I/Os

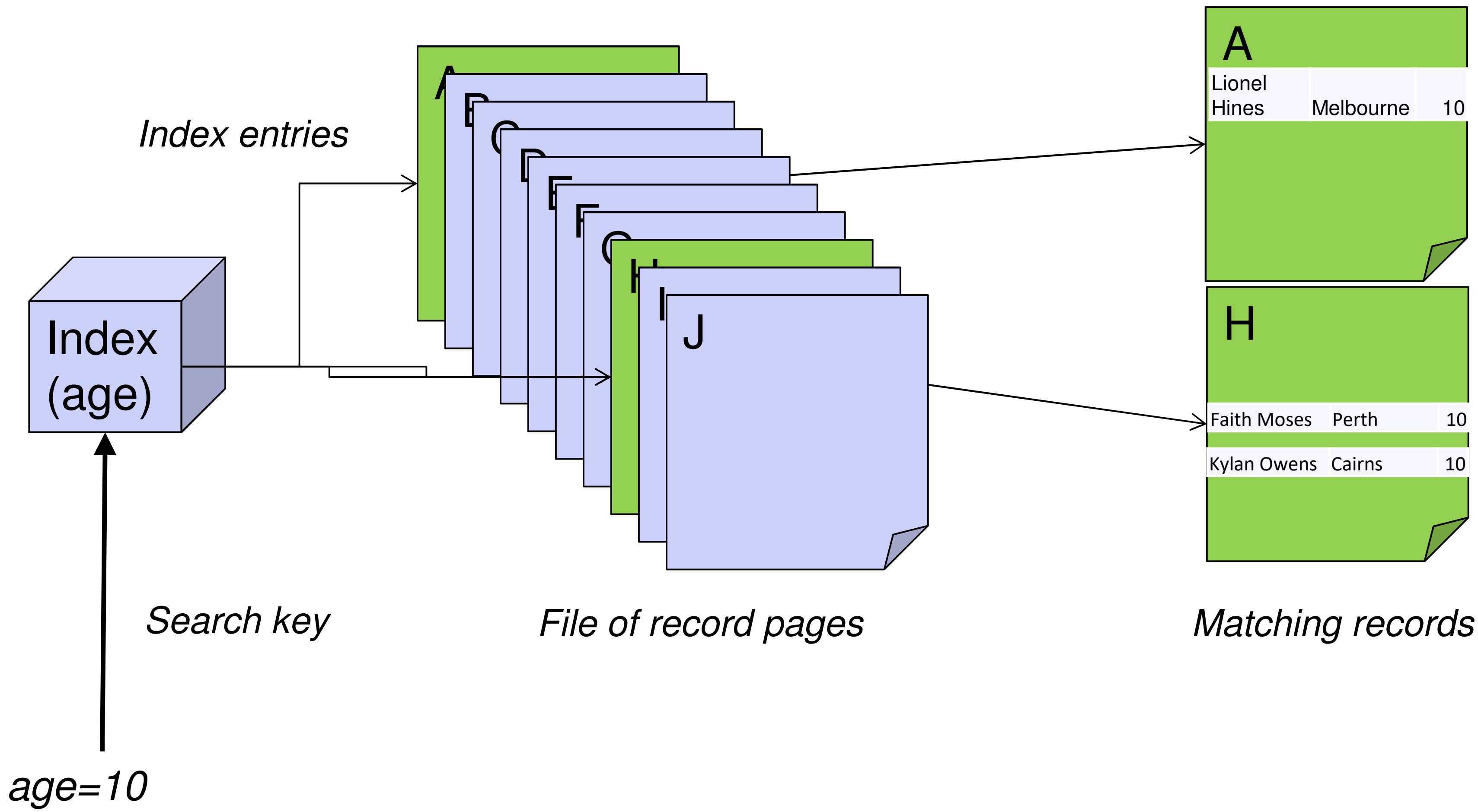
Indexes

Principle

- Database Indexing illustrates the fundamental principle of space-time trade-off.

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

Index Function



Indexes - The Downside

- Additional I/O to access index pages
  (except if index is small enough to fit in main memory)
  - The hope is that this is less than the saving through more efficient finding of data records

- Index must be updated when table is modified.
  - depending on index structure, this can become quite costly

- Not all query types are supported by all index types

Index Classification

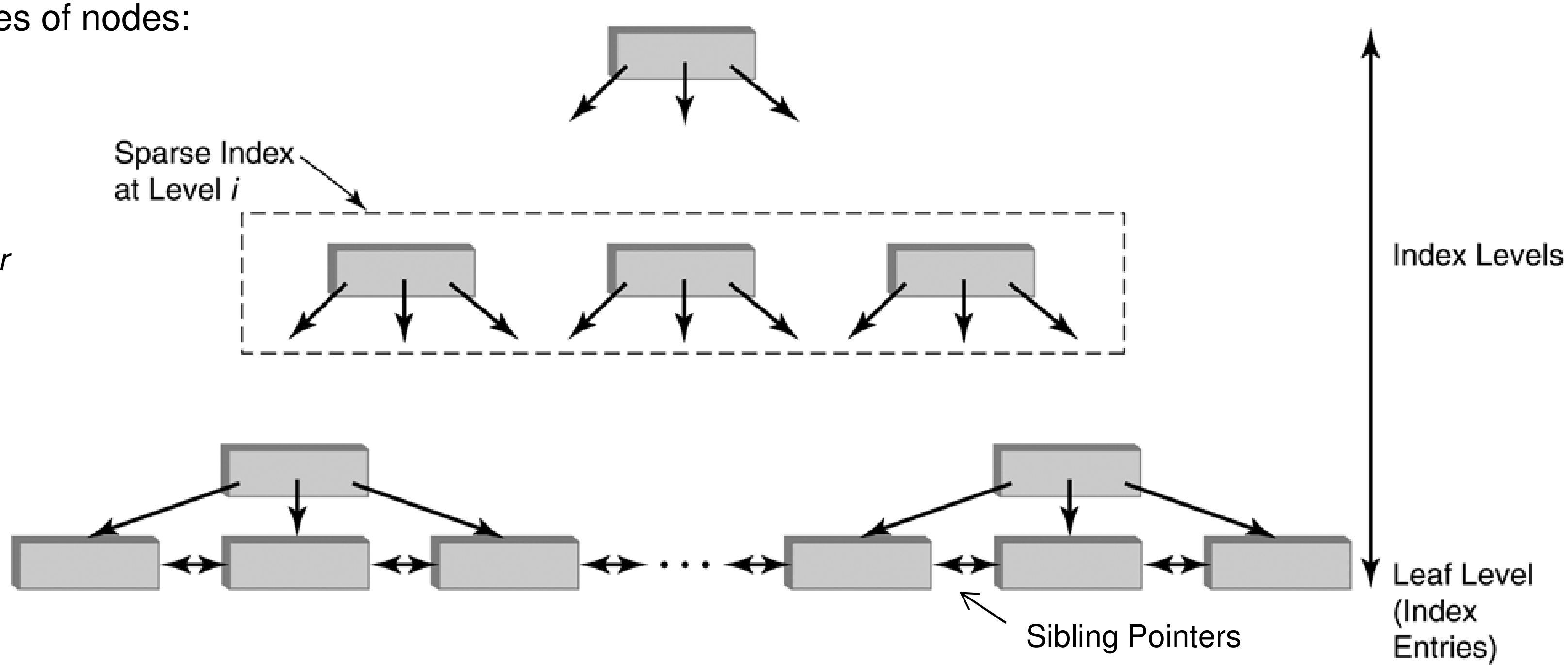
- Primary (Main) Index vs. Secondary Index
  - an index whose search key specifies the sequential order of the file is called the **primary index**.
    - Typically implemented by index entries containing actual data rows
    - Oracle calls this *integrated storage structure* an 'index-organised table' (IOT)
    - "Primary key" ≠ "Primary index"
  - Otherwise **secondary index**
- Unique vs. Non-Unique
  - an index over a candidate key is called a **unique index** (no duplicates)
- Clustered vs. Unclustered
  - If data records and index entries are ordered the same way, then called **clustered index**.
- Dense vs. Sparse
  - Index each value or only subset of it (e.g. each page)?

B+ tree Indexes

B+ Tree Index Structure

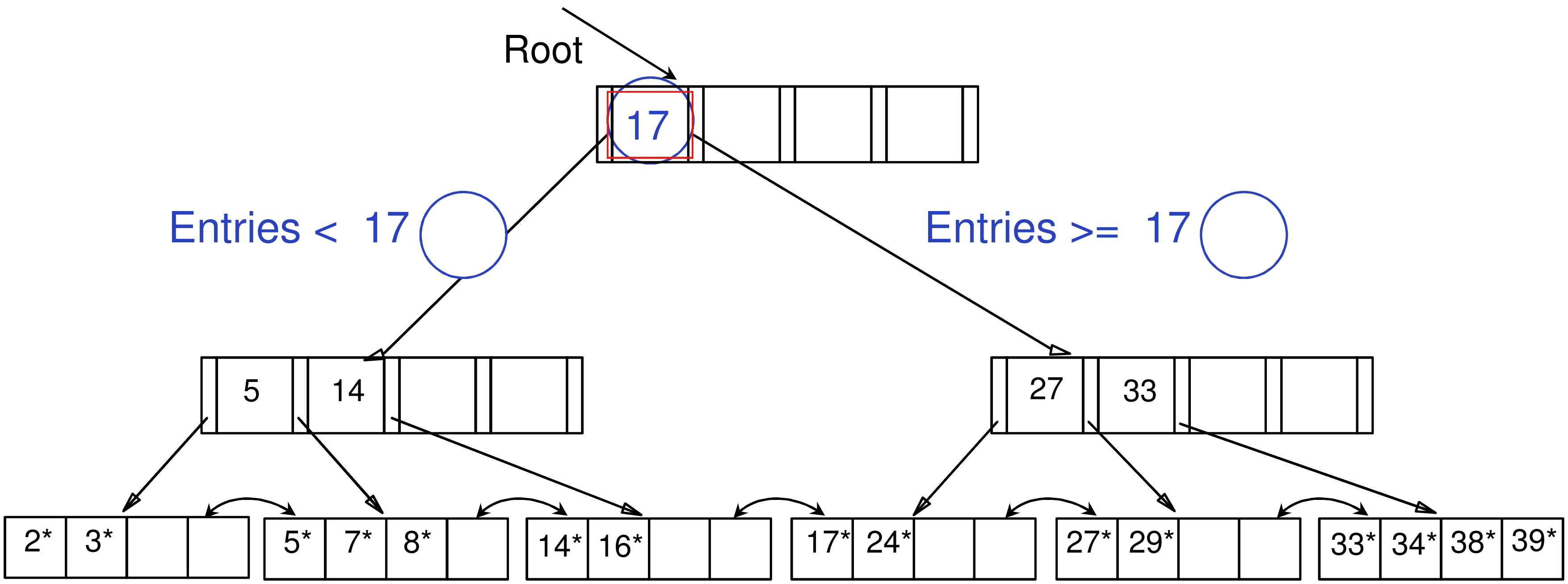
Three types of nodes:

- *Root*
- *Interior (separator entries)*
- *Leaf*



- *Leaf level* is a (sorted) linked list of index entries
  - *Sibling pointers* support range searches in spite of allocation and deallocation of leaf pages (but leaf pages might not be physically contiguous on disk)
- Non-leaf nodes have *separator entries*; only used to direct searches

Example of a B+ Tree Index



- Note how data entries in the leafs are sorted and linked
  - Primary index: leaves hold records themselves, else pointers to records
- Find 14? 29? All values >20 and <30?

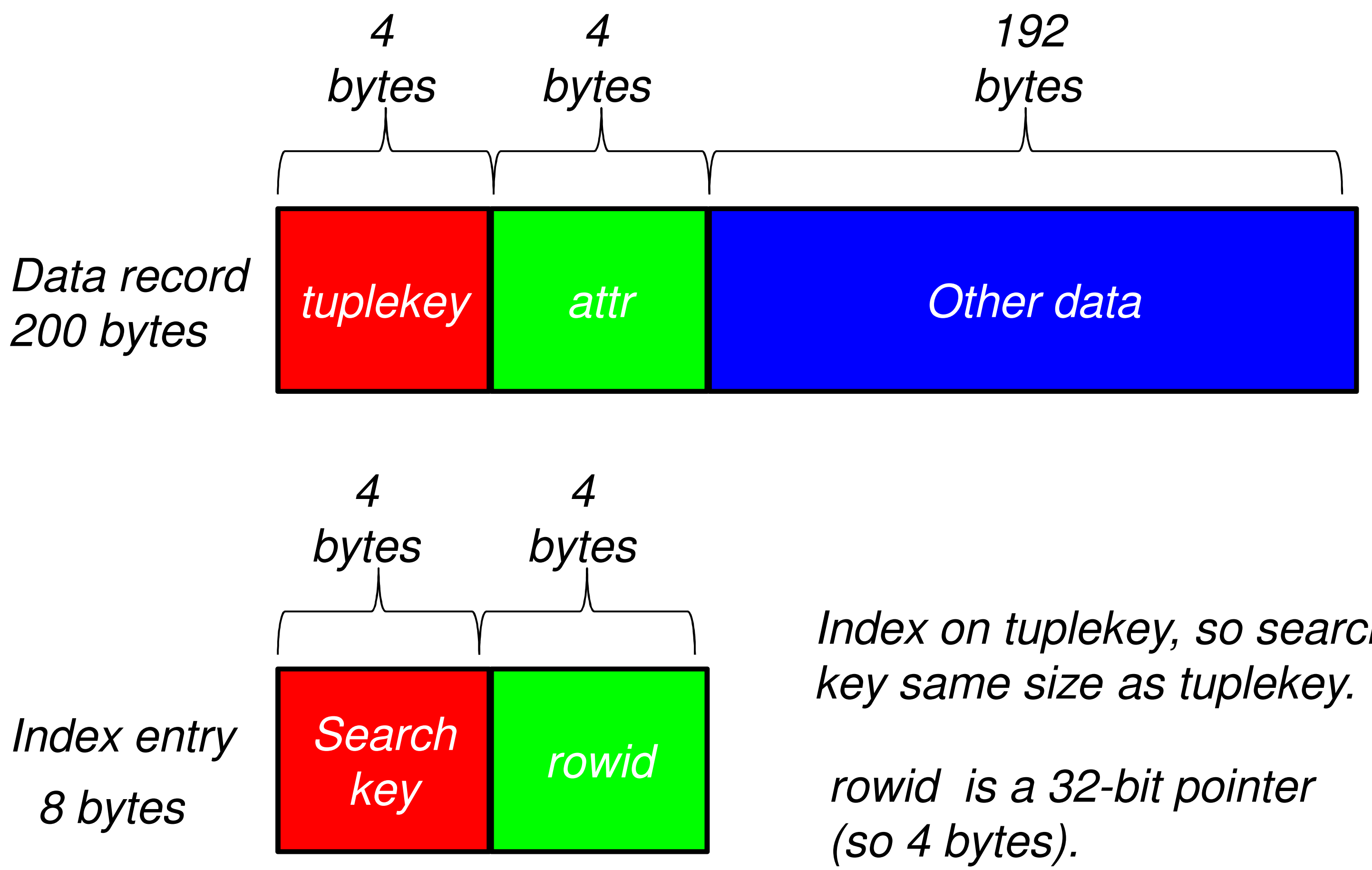
Estimates for Tree Index

- Start at leaf, and work upwards!
- How many leaf entries?
  - for dense index, equal to number of data records
  - for sparse index, equal to number of data blocks
- How many leaf blocks?
  - (number of leaf entries)/(number of leaf index records per block)
- How many index entries at next level?
  - one per leaf block
- How many blocks at next level?
  - (number of entries at this level)/(number of index entries per block)
- Repeat for each higher level, till a level has only 1 block (its the root of the tree)

Example: Clustered B+ Tree Index

- Suppose the relation is stored as a B+-tree, clustering on the attribute *tuplekey*. Assuming that the data pages are filled to 75% on average, how many pages are used by the data records themselves?
- How many pages are occupied by the pointers to data records?
- How many pages are used at each level of the index above this?
- How much I/O is needed to find the record with *tuplekey*=715?
- Is the index any help when finding the records with 100<=*attr*<120?

Running example: Storage of B+ tree index entries

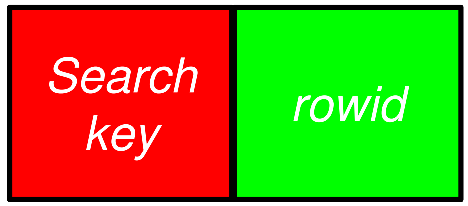


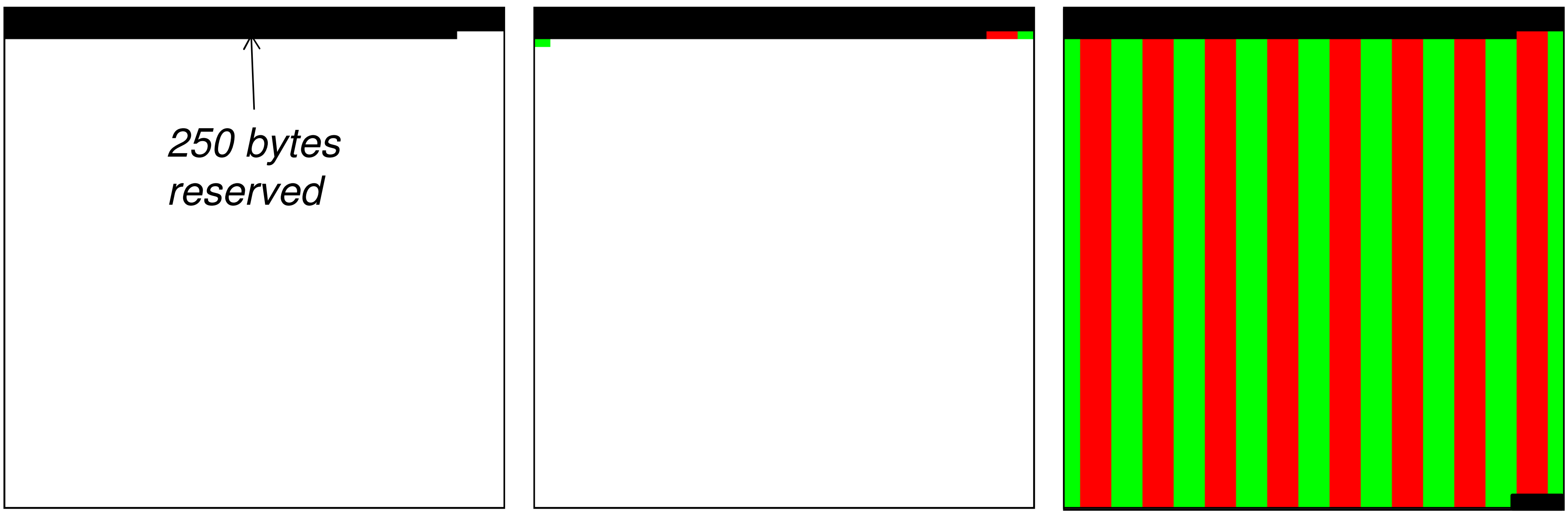
Index on tuplekey, so search key same size as tuplekey.

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

Fitting index data into pages

Each example index entry 8 bytes





 # entries: 0
empty space: 3846 bytes

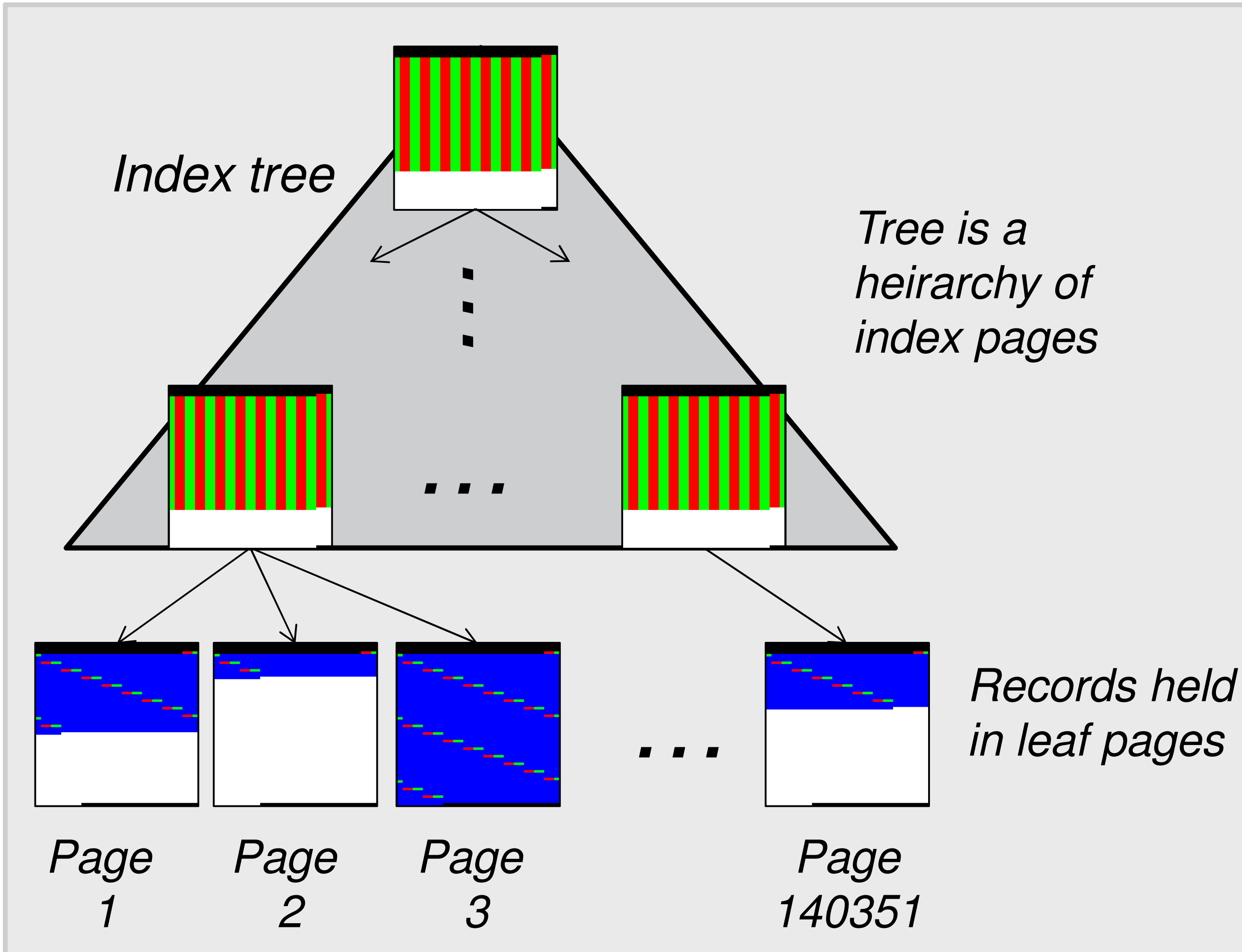
 # entries: 1
empty space: 3838 bytes

 # entries: 480
empty space: 6 bytes

$\lfloor 3846 \text{ bytes/page} \div 8 \text{ bytes/entry} \rfloor = 480 \text{ entries/page}$ *plus 6 remaining bytes*

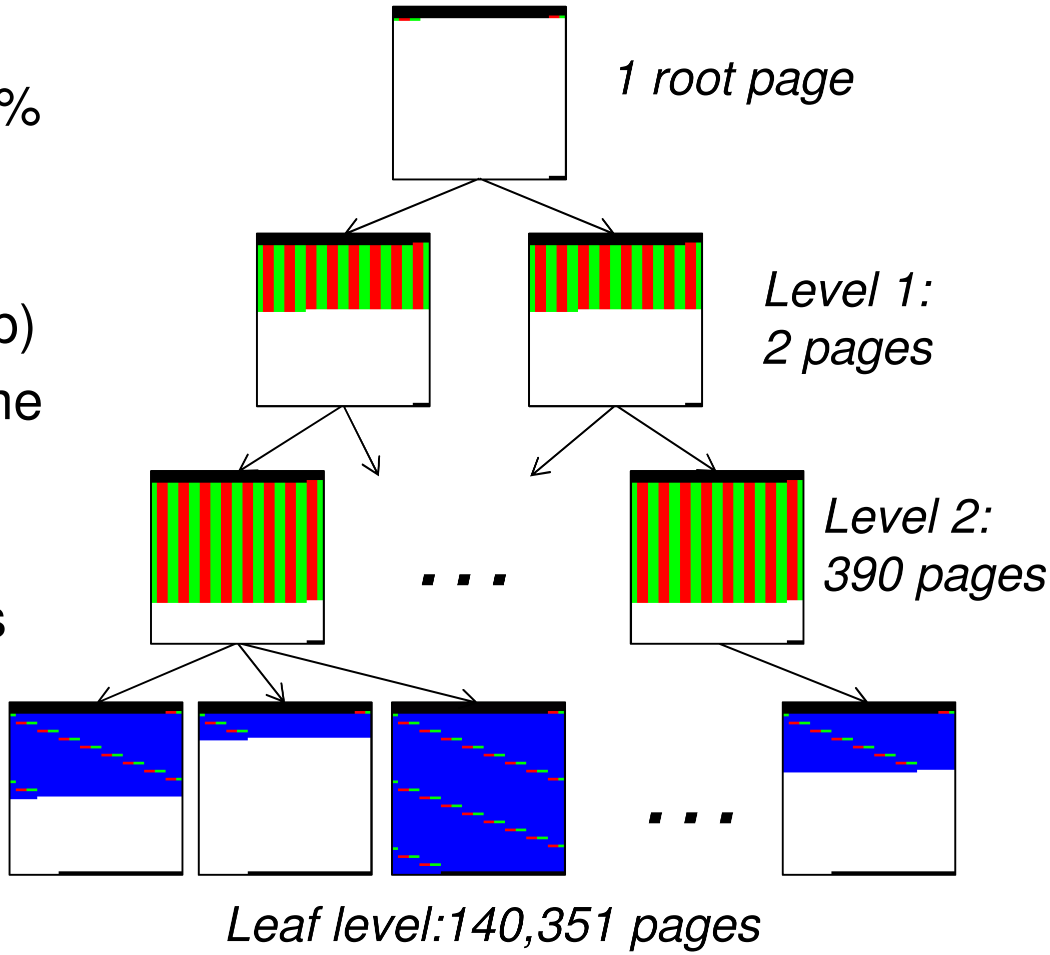
File of records stored as tree index

Integrated (/main/primary) index forms part of the file, so must be clustered and sparse



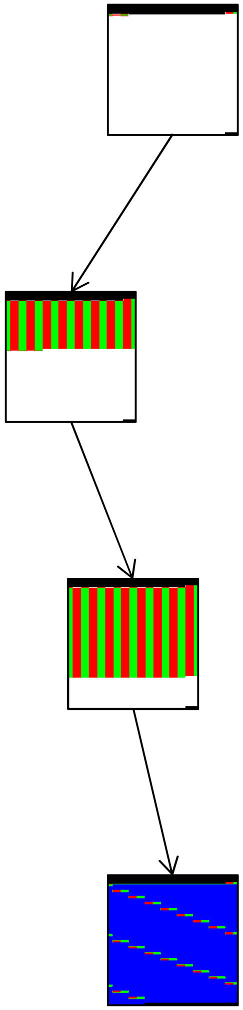
Size of integrated tree index

- Recall our index pages hold up to 480 entries (order 240)
- Stated average occupation 75% (often 67%)
- Fan-out is 480 × 75% = 360
- 140351/360 = 390 (rounded up)
- 390/360 = 2 (could also assume 1 since 360<480)
- 2/360 = 1 (root)
- 390 + 2 + 1 = 393 index pages
- 393/140351 = 0.2% increase



Finding records in a tree index file

- Data is sorted on *tuplekey* so for **equality search** can use index:
  1) Load index root into page buffer (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 page in leaf level;
  7) Load matching leaf page (cost 1 I/O);
  8) Check each record in page for match.
- Total of 4 I/Os vs 70,176 I/Os for heap file
- For **range search** on *attr* , for which index is no use, so still need to check each leaf page sequentially, so 140,351 I/Os

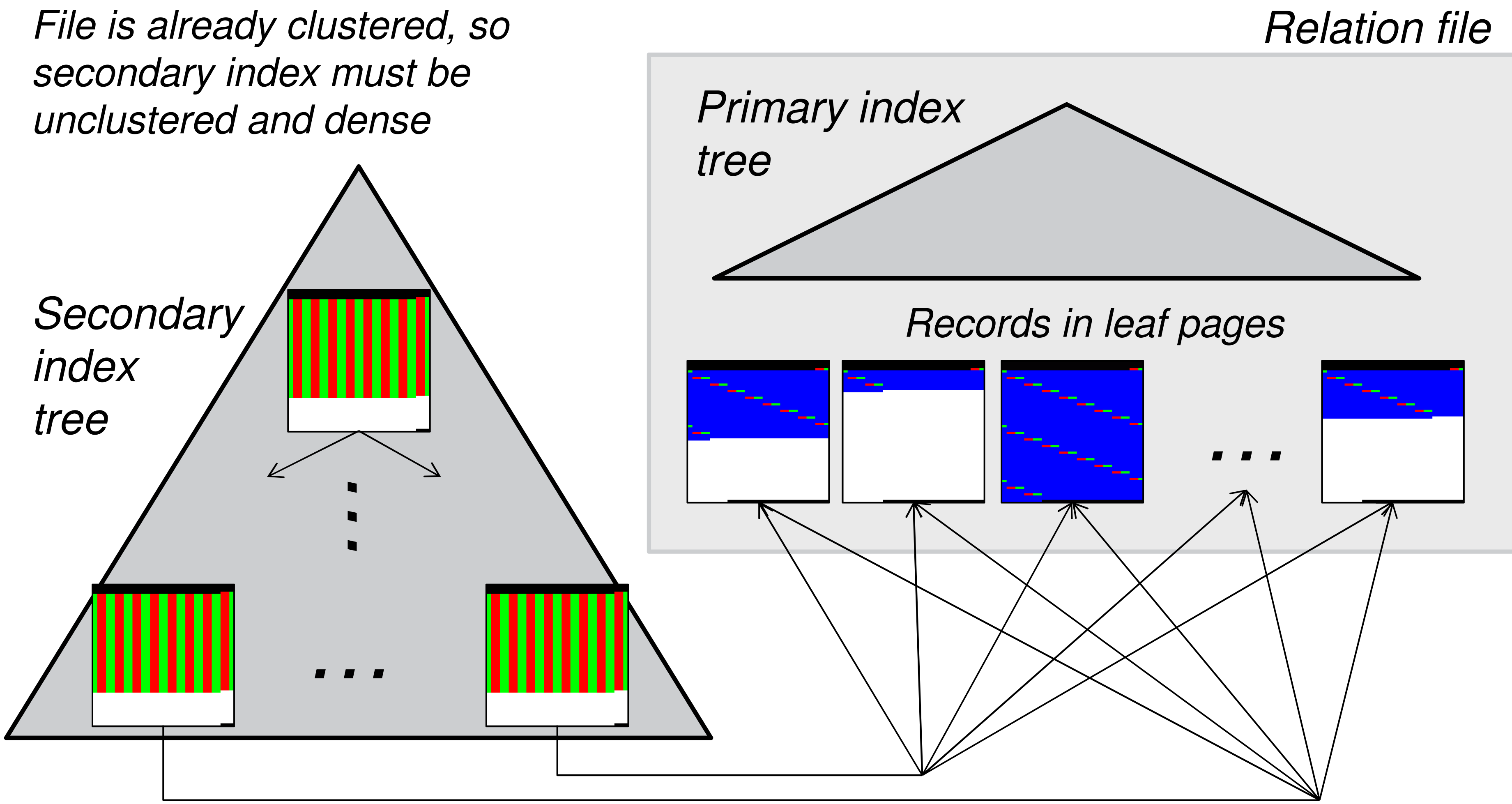


Example: Unclustered secondary B+ Tree index

- Suppose the relation is stored as a B+-tree, clustering on the attribute tuplekey, and that there is also an unclustered B+-tree index on attr.
- How many pages are used for the unclustered index?
- How much I/O is needed to find the records with 100<=*attr*<120?

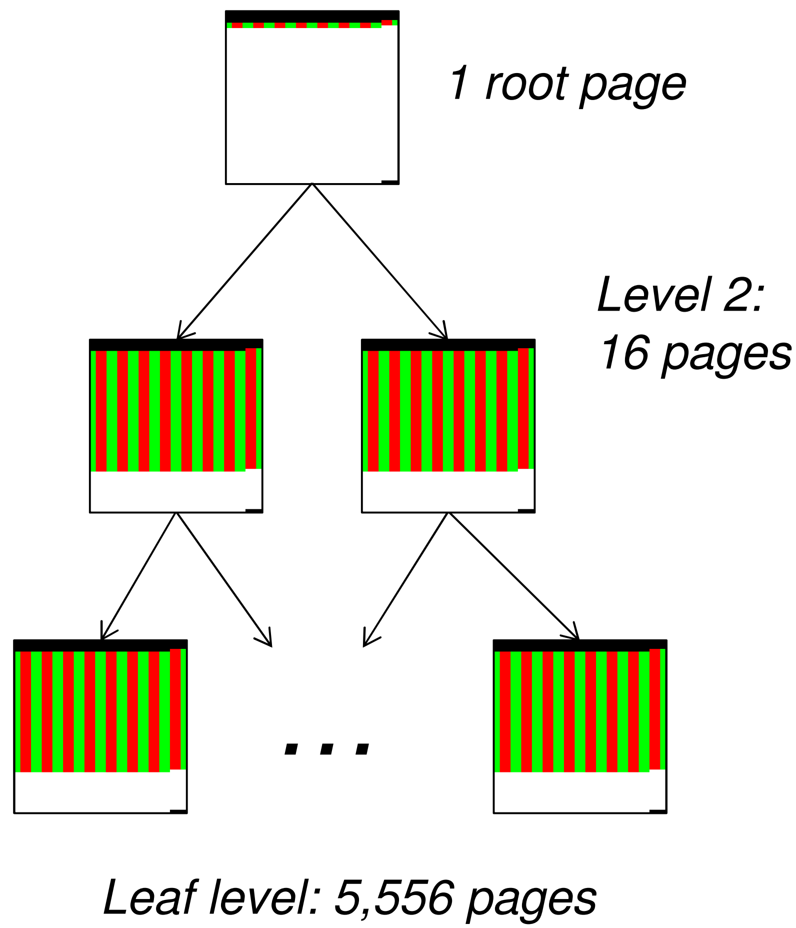
File of records with secondary tree index

File is already clustered, so secondary index must be unclustered and dense



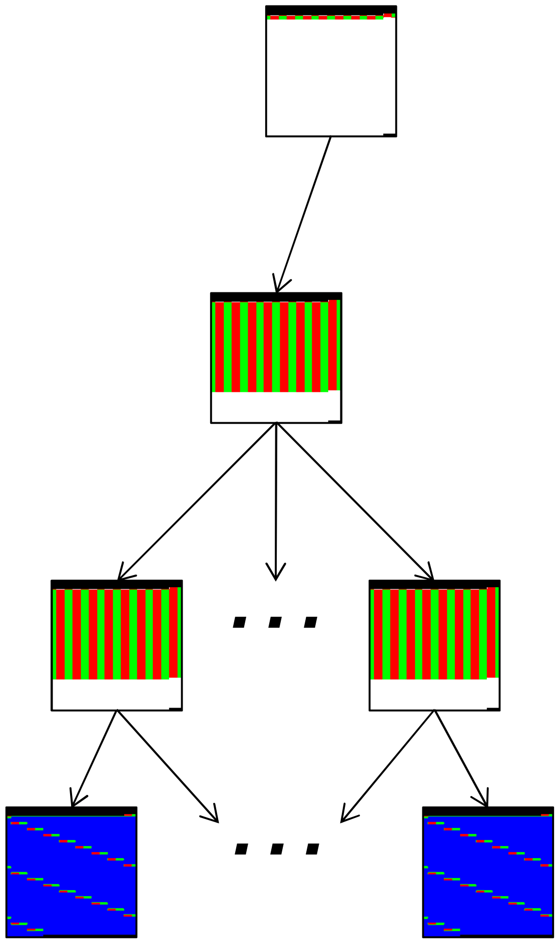
Size of secondary tree index

- Fan-out remains 480 × 75% = 360
- Dense so one leaf entry per record
- 2,000,000 records/360 entries = 5556 leaf pages
- 5556 leaf pages/360 = 16 pages
- 16/360 = 1 (root)
- 5556 + 16 + 1 = 5573 index pages
- 5573 index pages/140351 record pages = 4% increase
- 3 I/Os to find location of matching record page, plus 1 I/O to retrieve record page.



Finding records with a secondary tree index file

- Index is only suitable for searches on *attr*, such as the **range search** for values in [100,120) – same algorithm as before.
- Recall we expect 4000 matching records. Records are not sorted by *attr* so matches may be spread across up to 4000 pages
- In index the entries are sorted, so in 4000/360=12 matching leaf pages (rounded up)
- 12/360 = 1 level 2 page, plus root
- Total of 4000 + 12 + 1 + 1 = 4014 I/Os vs 140351 I/Os for heap file (factor of 35 improvement)



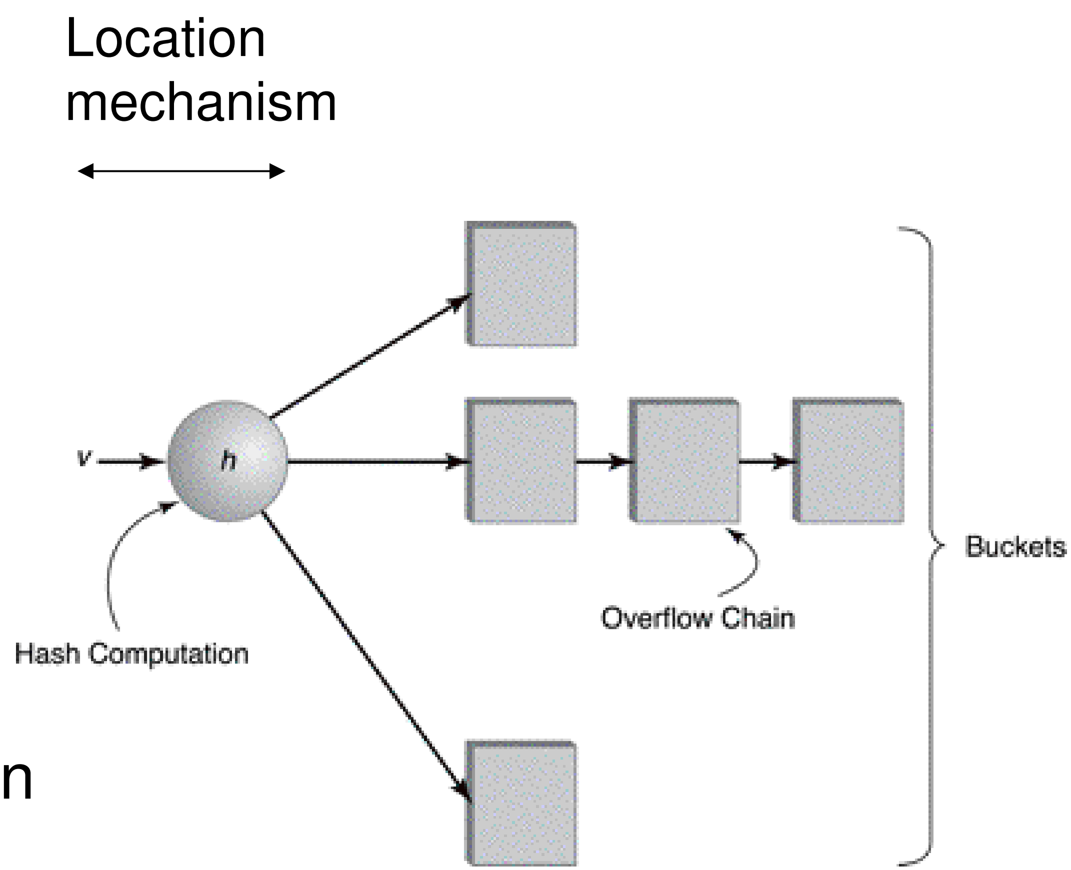
4000 matching record pages (not sequential)

Hash Indexes

Hash Index

- Index entries partitioned into **buckets** in accordance with a **hash function**, ***h(v)***, where *v* ranges over search key values
- Each bucket is identified by an address, *a*
- Bucket at address *a* contains all index entries with search key *v* such that $h(v) = a$
- A bucket is a unit of storage containing one or more entries that is stored in a page (with possible overflow chain)
- If index entries contain rows, set of buckets forms an integrated storage structure; else set of buckets forms an (unclustered) secondary index

Equality Search with Hash Index



Given *v*:

1. Compute *h(v)*
2. Fetch bucket at *h(v)*
3. Search bucket

Cost = number of pages in bucket (cheaper than B^+ tree, if no overflow chains)

Hash-Based Indexing

- As for any index, two alternatives for index entries **k***:
  - Integrated Index: Data record with key value **k**
  - Secondary Index: <**k**, TID of data record with search key value **k**>
  - Choice orthogonal to the *indexing technique*
- Hash-based indexes are best for *equality selections*.
  Not support range searches.

DBMS Comparison: Index Types

	DB2 UDB 8.2	Oracle 12c	SQLServer 2008	Sybase ASE 12.5	Postgres 9	MySQL 5
B+-Tree	yes	yes	yes	yes	yes	yes
Hash Index	---	no	---	---	yes	MEMORY tables
Bitmap Index	(yes) (called EVI)	yes (since v8.1)	*Bitmap filter* ---	yes (in Adaptive IQ)	*bitmap scan* (since v8.1)	---
Specialities	*R-Tree(*)*	*R-Tree (*)*	Quad Tree; fulltext index	---	Inverted idx; GiST	Fulltext *R-Tree*
Integrated (Main) Index	no?	yes	yes	yes	**---**	InnoDB (always PK)
Clustered Index	yes	yes (only as so-called index-organised table)	yes (every clustered index is an integrated index)	yes (a clustered index is an integrated index)	yes	InnoDB (always PK)
Unique Index	yes	yes	yes	yes	yes	yes
Multi-Column Index	yes	yes	yes	yes	yes	yes

(*) spatial index via extension module

Summary

- Disks provide cheap, non-volatile storage.
  - Random access, but cost depends on location of page on disk.
- Buffer manager brings pages into main memory.
  - Page stays in memory until released by requestor.
  - Choice of frame to replace based on *replacement policy.*
  - Tries to *pre-fetch* several pages at a time; dirty pages written deferred.
- Classical DBMS storage architecture is a ‘row store’:
  - Variable length record format with field offset directory offers support for direct access to i’th field and null values.
  - Slotted page format supports variable length records and allows records to move on page; rows are indirectly addressed using tuple-identifier.
  - Data Compression on row and page/table level for better storage efficiency
- File layer keeps track of pages in a file, and supports abstraction of a collection of records.

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
- [Oracle Database Concepts]
  Oracle Corporation: Oracle 12c Documentation, *Database Concepts. – links in tutorial worksheet*