Storage Devices, Files, and Indexing

Bipin C. Desai

To be used in the spirit of copy-forward! https://users.encs.concordia.ca/~bcdesai/CopyForward.pdf

Storage Device Selection Criteria

Capacity vs. cost (What will $100 buy?
How much for 1 Megabytes?)

Cost per megabytes of storage has taken a plunge
Alas, the need for it has bounded as well.

Permanence
Portability
Relative cost
Performance (Latency, transfer /access rate)
Record size - buffer size, file size.
Accessing method - random/direct or sequential
Data transfer rate
Seek time - time to move read/write head:
average, minimum, maximum
Latency - rotational delay (rpm)

Memory Hierarchy

Speed	Technology	**Application**
1-10's nsec	I^2L	fast cache
	nmos	high speed MM
	bipolar	buffer
100's nsec	nmos	main memory
	core	
100's μsec	CCD	fast back up
	bubbles	
1-10's msec	floppy disk	main back up
	fixed head disk	
	moving head disk	
10's msec	magnetic tape	security/back up
100s of ms	optical memory	large mass
	tape library	memory,
	system	archives

Data on External Storage

Disks: Can retrieve random page at fixed cost

- But reading several consecutive pages is much cheaper than reading them in random order

Tapes: Can only read pages in sequence

- Cheaper than disks; used for archival storage – extinct??

File organization: Method of arranging a file of records on external storage.

- Record id (RID) is sufficient to physically locate record
- Indexes are data structures that allow us to find the record ids of records with given values in index search key fields

Architecture: Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

Store the database in Main Memory!

Costs too much. $100 will buy 16GB of DRAM or 500GB SSD today.

Main memory is volatile. We want data to be saved between runs. (Obviously!)

Typical storage hierarchy:

- Main memory (RAM) for currently used data.
- Disk for the main database (secondary storage).
- Tapes for archiving older versions of the data (tertiary storage).

EXTERNAL STORAGE MEDIUMS

Read/Write	Write once read many times(WORM) (used for archives)
Magnetic Tape	Disks/tapes
Disk	Robotic storage media
RAID(Redundant array of inexpensive disks	CD-Rom.

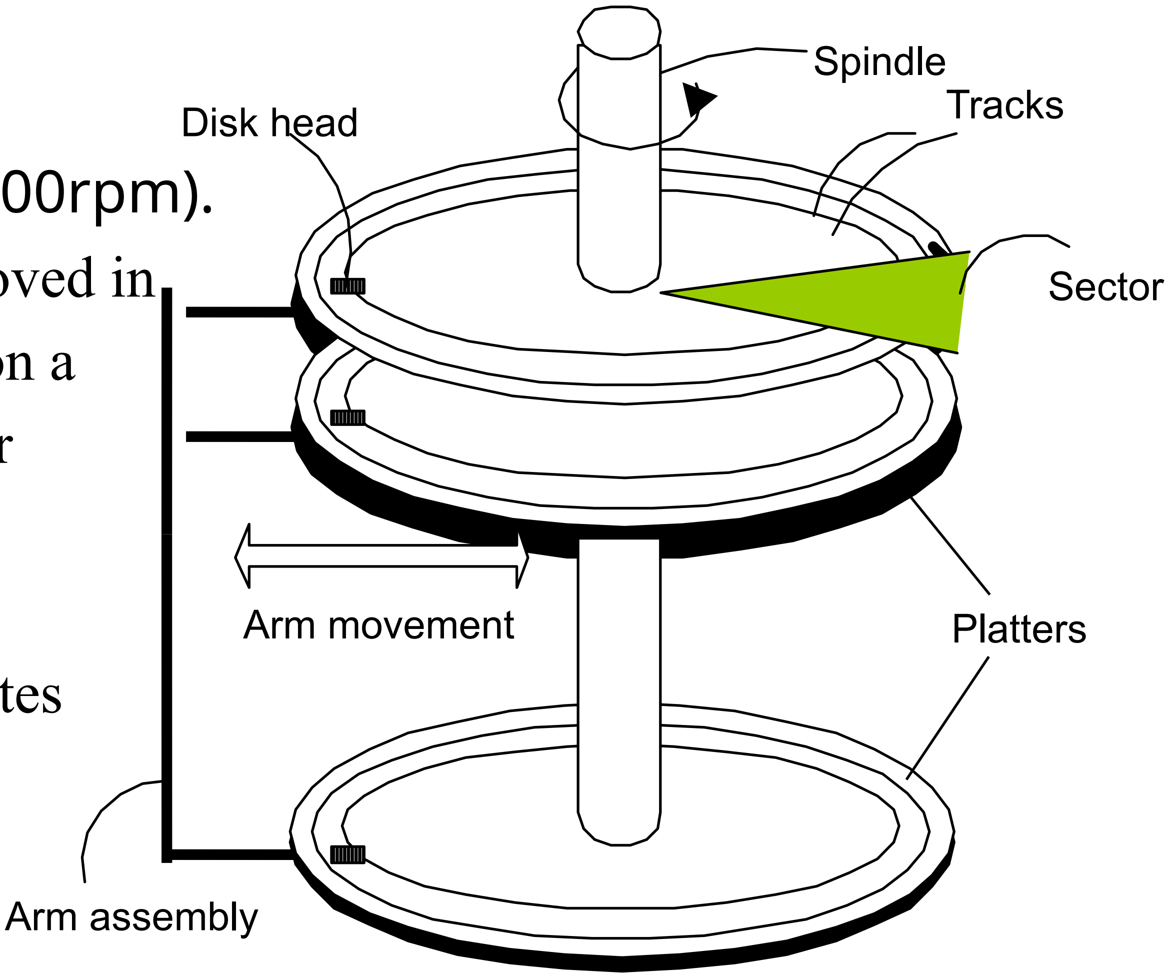
READ: transfer data to main memory.
WRITE: transfer data to external device.
READ/WRITE are much slower than main-memory operations!

Disks

- Secondary storage device of choice.
- Main advantage over tapes: *random access* vs. *sequential*.
- Data is stored and retrieved in units called *disk blocks* or *pages*.
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
  - Therefore, relative placement of pages on disk has major impact on DBMS performance!

Components of a Hard Disk

- The platters spin (7200rpm).
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).
- Only one head reads/writes at any one time.
- *Block size* is a multiple of fixed *sector size*



SSDs do not have moving parts but a finite number of cycles

Seek Time

Seek Time = $c_1 + c_2$ * (number of cylinders to be traversed).

Here c_1 and c_2 are constants for a given model of disk drive.

Average Seek Time = time to move over 1/3 cylinders.

Seek time can be reduced by:

- distributing a file over a number of disk units and
- limiting the range of cylinders on any disk unit.

Rotational Latency

The delay between the completion of the seek and the actual transfer of data.

For a disk rotating at r (RPM)

$$t_1 = \frac{60 * 1000}{2 * r} \text{ milliseconds}$$

RPM	Latency
10000	3 msec
7200	**4.1 msec**
6000	5 msec
3000	10 msec
2400	12.5 msec

Accessing a Disk Page

- Time to a read or a write a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

Response time = seek time + latency time + transfer time
(5-20 msec) (3-5 msec).
Transfer time = size of transfer/rate of transfer.
Size of transfer corresponds to the data of interest
(excluding format information, etc.)
Sequential Read of a number of blocks.
Transfer time = avg. seek time + latency time +
(block transfer time) * number of blocks
+ (min. seek time + latency) * number of cylinders
Problems: Disk scheduling in multi-process environment
Approximation: Transfer time = t_{efb} * # of blocks,
Here, t_{efb} is the effective formatted block transfer time.
$t_{efb} \approx 1.10 * t_b$, where t_b is the block transfer time
to account for the format information and the ignored seek and latency time.
Block transfer time = block size/ rate of transfer

Random Read of a # of blocks

Transfer time = number of blocks * (seek + latency + t_b)

Sequential Read from a number of contiguous cylinders

Transfer time = seek time +latency time + t_{efb} * # of block +
(min seek time + latency time) * (# of cylinders -1)

File Organisation	the storage required for the file organization
sequential	the time required to read a random record
indexed sequential	the time required to read the next record
direct access	the time required to add a record
other method	the time required to update a record
	the time required to read all records
	the time required to reorganize a file

Choice

- external storage device available	simple
- use of the file - type of queries	x = y
- number of keys	range
- mode of retrieval - seq. random	Boolean x=y
- mode of update	batch
- economy of storage	on-line
- frequency of use of a file	
- growth potential of a file	
- methods available in the development environment	

Updates:

- insert in sequence
  at end, at first available location
- delete - compress first available location
  flag as deleted
- modify selected record   space for update
  record size with respect to size of original record
- modify all records

Primary Key Retrieval

Four (three) possible choices -

- serial file - no order (pile)
- sequential - ordered wrt primary key
- indexed sequence
- direct access

Serial Files (PILE)

Access a random record
Access to Next Record
Inserting Record
Deleting a Record
Modifying a Record
Reorganisation
Single Disk Drive
Two or more Disk Drives

Sequential File

Access a random record
Access to Next Record
Inserting Record
Deleting a Record
Modifying a Record
Reorganisation
Single Disk Drive
Two or more Disk Drives

Access to Next Record

Probability of record in same block = $1 - 1/ b_f$

Probability of record not same block = $1/b_f$

Expected time to get next record.

$$= 0 * (1 - 1/ b_f) + 1 * (1/ b_f)*(t_s + t_l + t_b)$$

$$= 1/ b_f*(t_{efb})$$

Modify-in-place or Delete a Record

- Find it in T_f (Time to find random record)
- Max. time to modify or mark it as deleted, and wait
  $2T_l$ - block txf time
- Rewrite it in time = block txf time
  Total time = $T_f + 2T_l$

Sector Addressable Disks

- fixed length arcs of a track - track is divided into an integral number of sectors.
- amount of data is fixed by O.S. or by the hardware.
  - simplifies allocation of storage space
  - simplifies address calculations
  - simplifies synchronisation of I/O & computation in sequential processing.

The division of a track into **sectors**:

- may be implemented completely by **hardware** or
- by **software** controlled formatting operation.

Block is a fixed number of bytes that is moved as a unit between storage devices and the main memory. Made up a number of disk sectors.

Arranging Pages on Disk

- `*Next*' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Blocks in a file should be arranged sequentially on disk (by `next'), to minimize seek and rotational delay.
- For a sequential scan, *pre-fetching* several pages at a time
- "De-fragmentation" to increase access

RAID

- Disk Array: Arrangement of several “inexpensive” disks that gives abstraction of a single, large disk.
- Goals: Increase performance and reliability.
- Two main techniques:
  - Data striping: Data is partitioned; size of a partition is called the striping unit. Partitions are distributed over several disks.
  - Redundancy: More disks => more failures. Redundant information allows reconstruction of data if a disk fails.

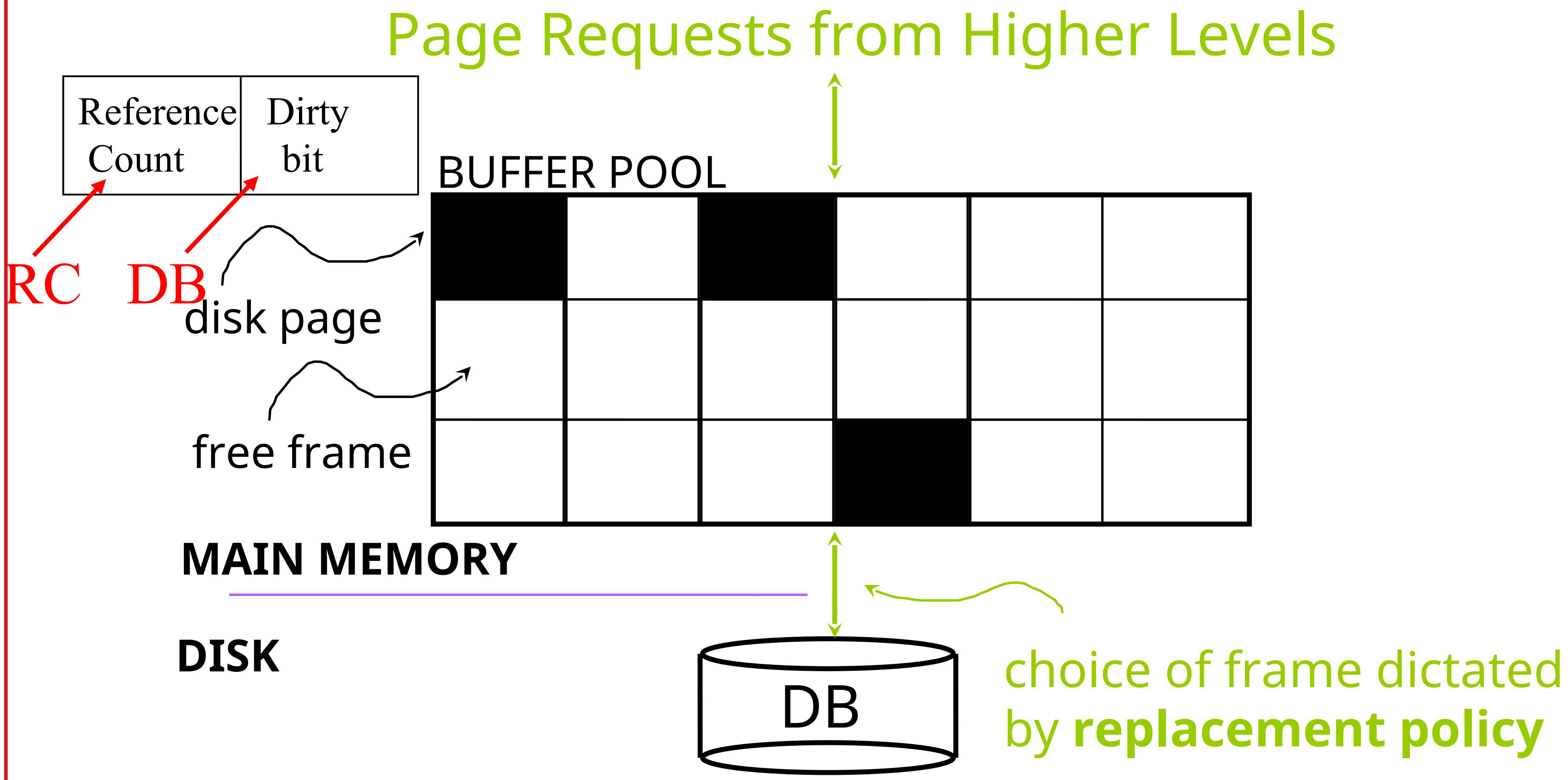
- Level 0: No redundancy
- Level 1: Mirrored (two identical copies)
  - Each disk has a mirror image (check disk)
  - Parallel reads, a write involves two disks.
  - Maximum transfer rate = transfer rate of one disk

- Level 0+1: Striping and Mirroring
  - Parallel reads, a write involves two disks.
  - Maximum transfer rate = aggregate bandwidth
- Level 3: Bit-Interleaved Parity
  - Striping Unit: One bit. One check disk.
  - Each read and write request involves all disks; disk array can process one request at a time.
- Level 4: Block-Interleaved Parity
  - Striping Unit: One disk block. One check disk.
  - Parallel reads possible for small requests, large requests can utilize full bandwidth
  - Writes involve modified block and check disk
- Level 5: Block-Interleaved Distributed Parity
  - Similar to RAID Level 4, but parity blocks are distributed over all disks

Disk Space Management

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done, or how free space is managed.

DBMS: Buffer Management


Page Requests from Higher Levels
Reference Count
Dirty bit
RC
DB
BUFFER POOL
disk page
free frame
MAIN MEMORY
DISK
DB
choice of frame dictated by **replacement policy**

- *DBMS operates on data in main memory*
- *Buffer management maintains a table <frame#, pageid>*

When a Page is Requested ...

- If requested page is not in pool:
  - Choose a frame for *replacement (LIFO, FIFO, LRU(RC), modified (DB), etc.)*
  - If frame is dirty (changed since read into buffer), write it to disk(replacement frame scheme looks for non-dirty frame
  - Read requested page into chosen frame
- *Increment the* ***reference count*** *(RC) of* the page and return its address.

If requests can be predicted (e.g., sequential scans) pages can be pre-fetched

More on Buffer Management

- When a frame is released by an application, the RC is decemented and if the frame is changed, the dirty bit for the frame is set.
- A frame in the buffer may be requested many times, concurrently(reads – not update/write)
  - a *RC* is used to indicate the number of concurrent use of a frame. A frame is a candidate for replacement iff *RC* = 0.
  - Priority if dirty bit is not set(not modified)
- Concurrency control and recovery may entail additional I/O when a frame is chosen for replacement.

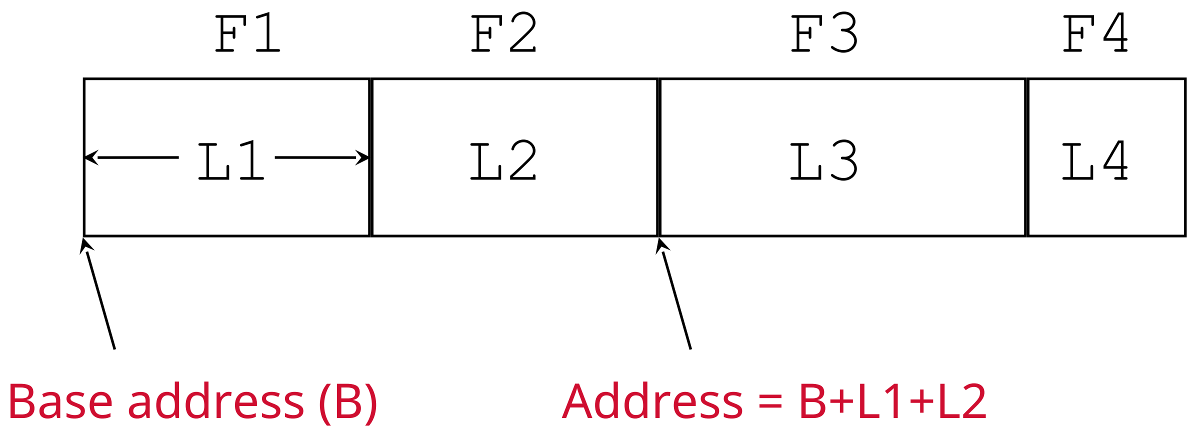
Buffer Replacement Policy

- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock, MRU etc.
- Policy can have big impact on # of I/O's; depends on the *access pattern.*
- *Sequential flooding*: Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O.

DBMS vs. OS File System

- Differences in different level of support in different OS: portability issues
- Some limitations, e.g., files can't span disks.
- Buffer management in DBMS requires ability to:
  - Manage RC and DB of frames in buffer pool, force a page to disk (important for implementing concurrency control and recovery),
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.

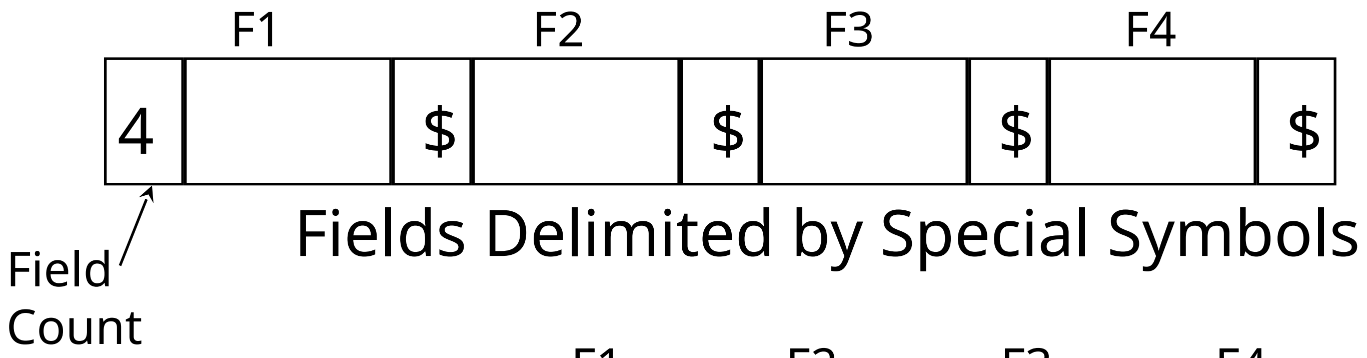
Data Record Formats: Fixed Length



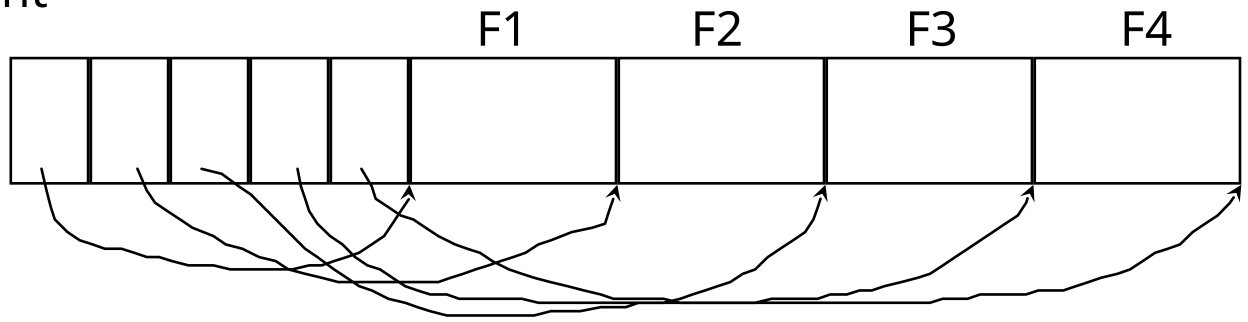
- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field does not require scan of record.

Data Record Formats: Variable Length

- Two alternative formats (# fields is fixed):



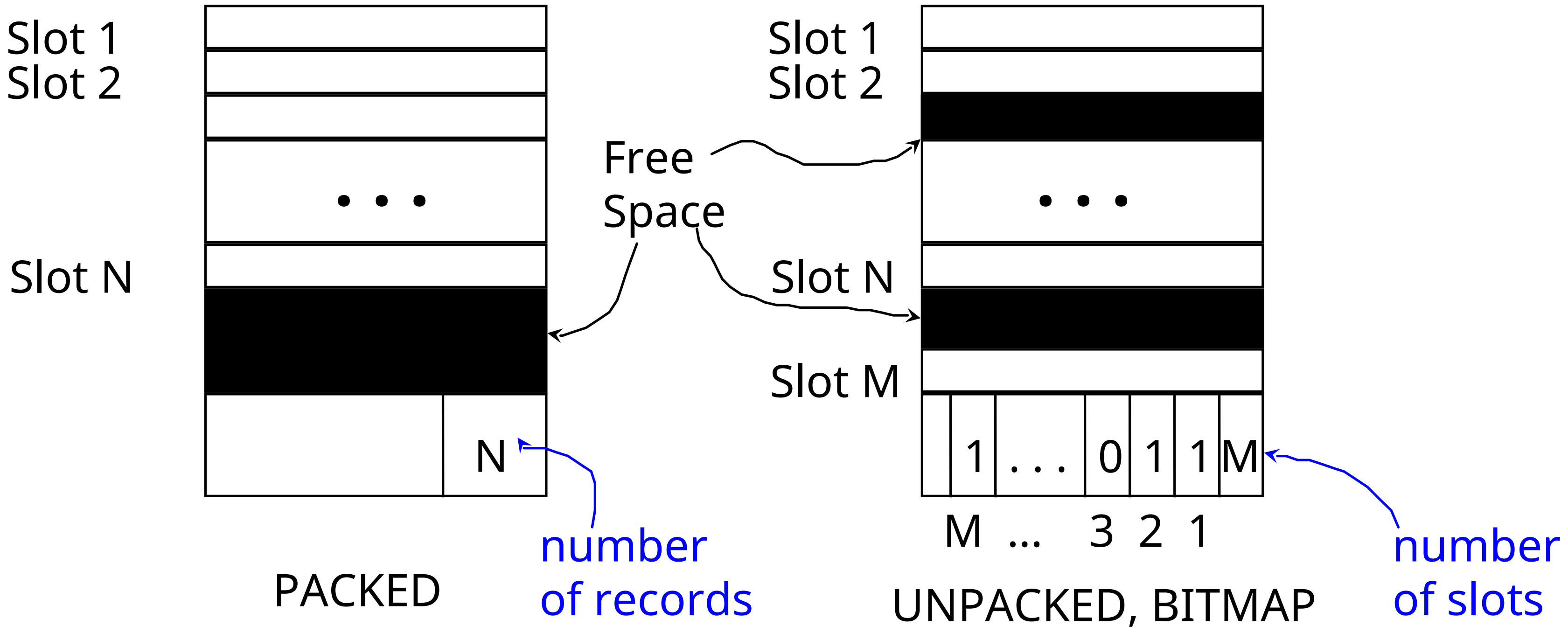
Fields Delimited by Special Symbols



Array of Field Offsets

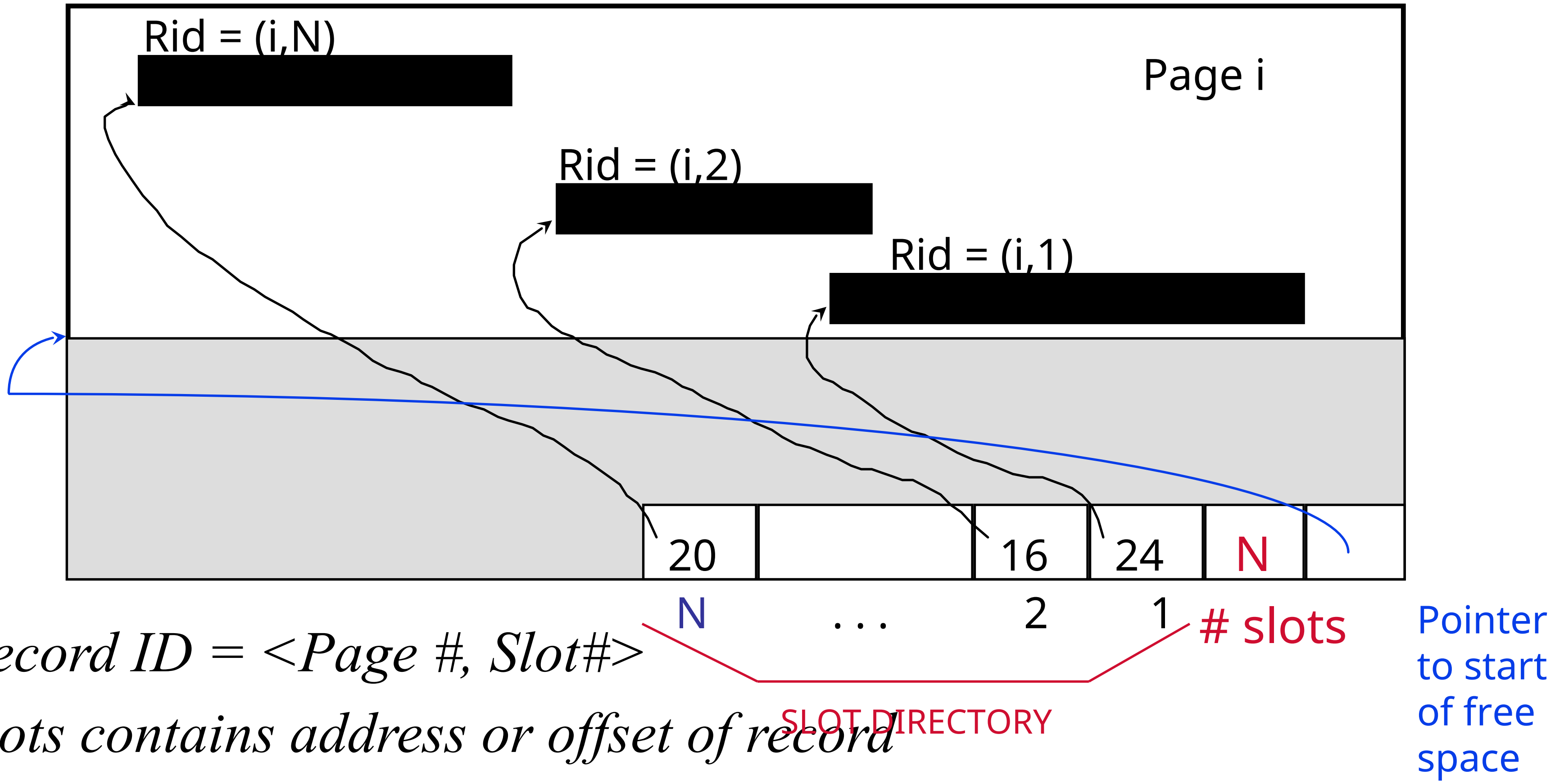
☛ Second offers direct access to i’th field, efficient storage of *nulls* (special *don’t know* value); small directory overhead.

Page Formats: Fixed Length Records



Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.

Page Formats: Variable Length Records



Record ID = <Page #, Slot#>

Slots contains address or offset of record

Can move records on the page without changing the record ID (RID);

Can also be used for fixed-length records!

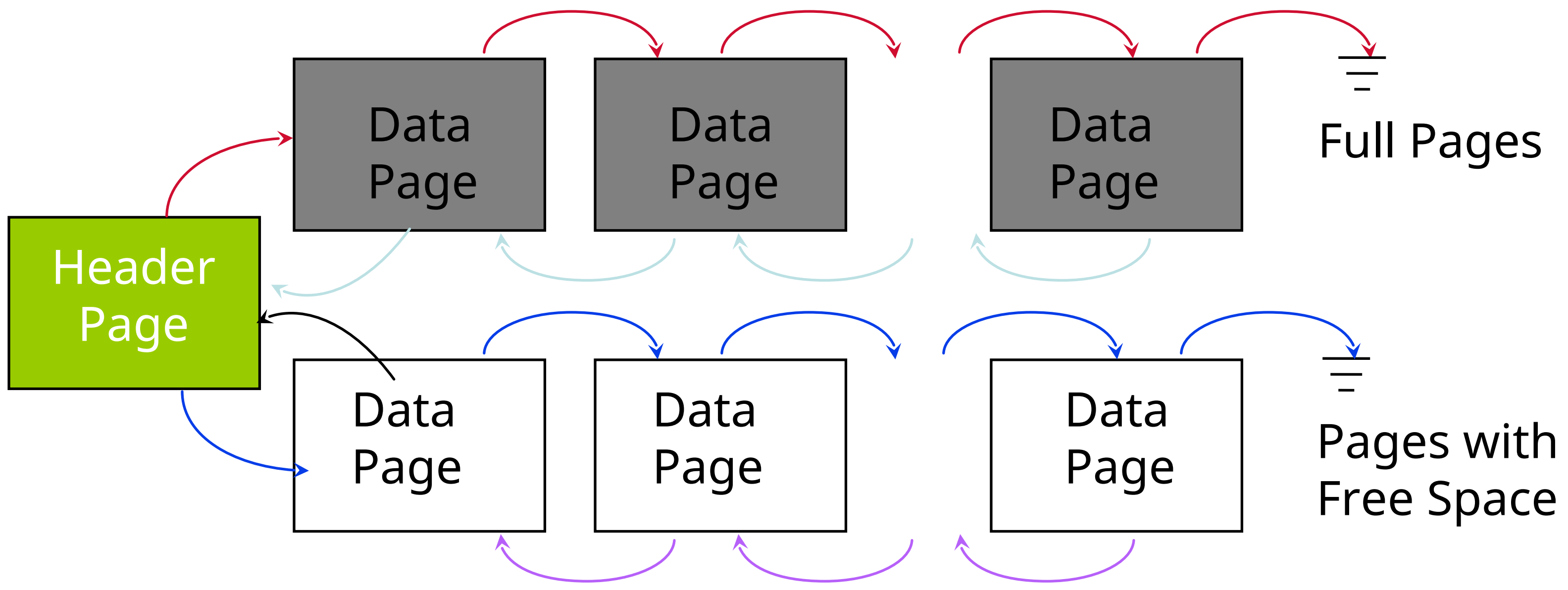
Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

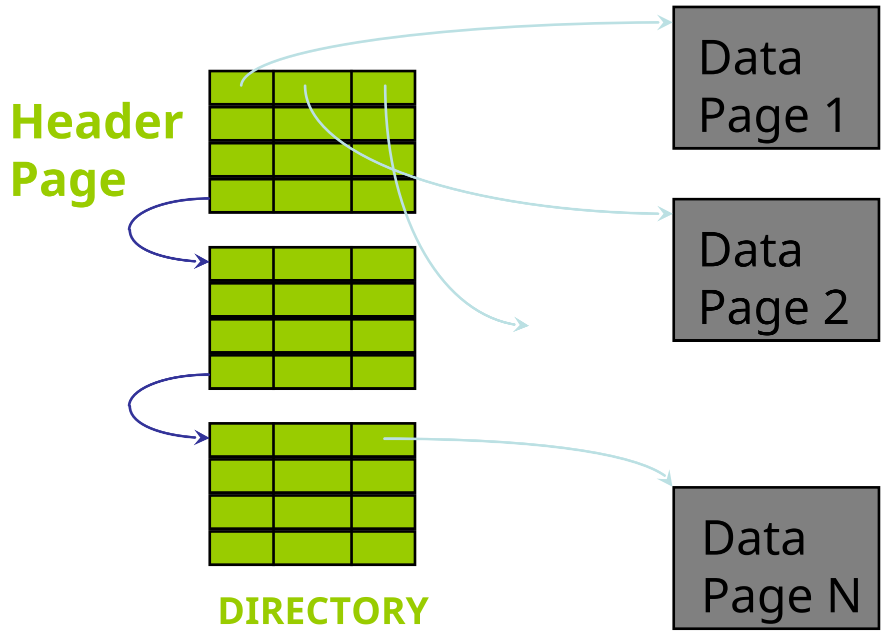
- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of these details.

Heap File Implemented as a List



- The Heap file name and its header page address must be stored in a catalog.
- Each page contains 2 `pointers’ (forward, reverse) plus data.

Heap File Using a Page Directory



- The entry for a page can include the amount of free space on the page.
- The directory is a collection of pages; for example implemented as a linked list
- *Much smaller than linked list of all HF pages!*

System Catalogs

- *Catalogs are stored as tables.*
- For each table:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- For each index:
  - structure (e.g., B+ tree) and search key fields
- Plus statistics, authorization, buffer pool size, etc.

Alternative File Organizations

Heap files: Suitable when typical access requires access to all records is a file.

Sorted Files: Suitable in cases where the records must be retrieved in some order wrt a “key”, or access to a records in a `range’ of key values is needed.

Hashed Files: Suitable when random access to records with a given key value is required.

B: The number of blocks (pages) for data

b_f: Blocking factor(# records per block)

t_{efb} **:** Effective time to read or write block

	Heap File	Sorted File	Hashed File
Scan all recs	**B**t_{efb}	**B**t_{efb}	**1.25 B**t_{efb}
Equality Search	**0.5 B**t_{efb}	t_{efb} **$\log_2$B**	t_{efb}
Range Search	**B**t_{efb}	t_{efb} **($\log_2$B + # of blocks with matches)**	**1.25 B**t_{efb}
Insert	**2**t_{efb}	**Search + B**t_{efb}	**2**t_{efb}
Delete	**Search +** t_{efb}	**Search + B**t_{efb}	**2**t_{efb}

INDEX

An index is created to speed up access to the records in a file with a given value for a **search key fields**.

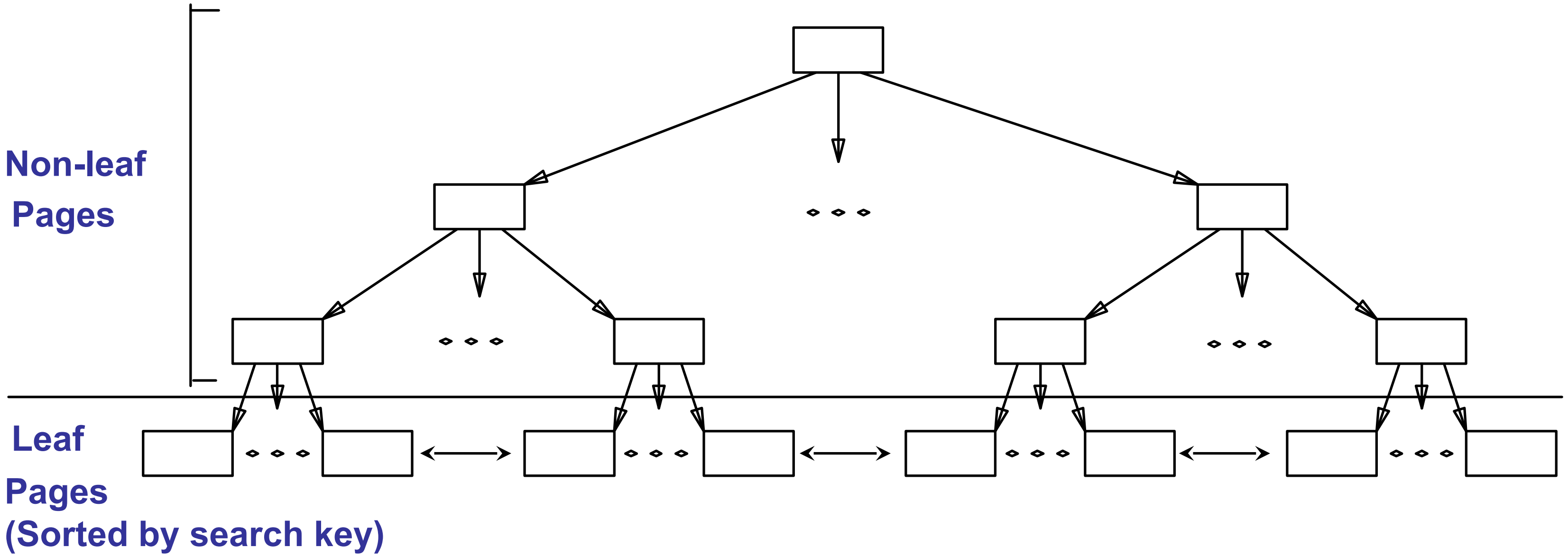
Any subset of the fields of a record can be used as search key for an index on the relation.

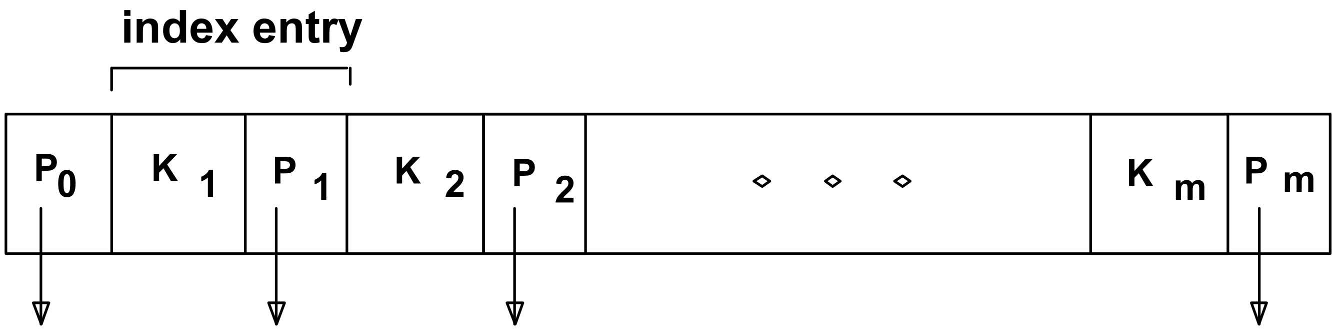
Search key may not be the same as primary key

An index contains a collection of data entries, and supports efficient retrieval of all records with a given search key value **K**.

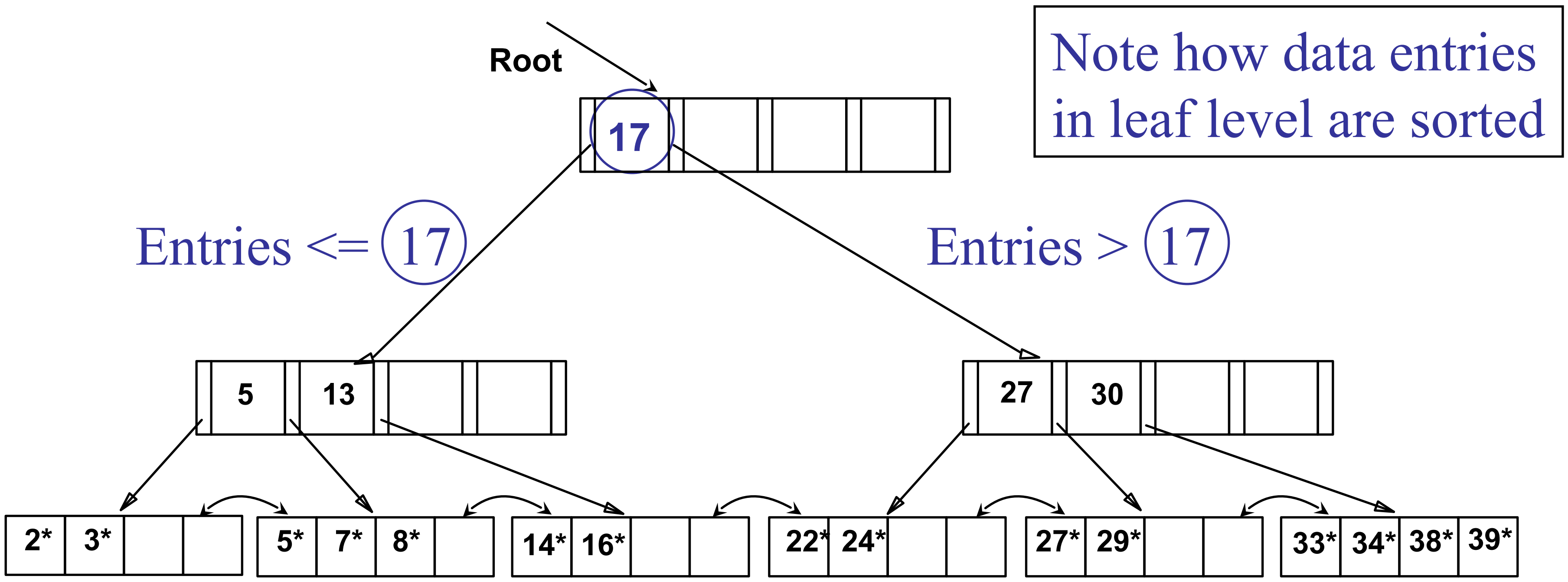
B+ Tree Indexes

- Internal nodes (pages) have *index entries;* only used for navigation:
- Leaf pages contain *data entries*, and are chained (prev & next)





Example B+ Tree



Note how data entries in leaf level are sorted

- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

Hash-Based Indexes

- Good for equality selections.
- Index is a collection of *buckets*.
  - Bucket = *primary* page plus zero or more *overflow* pages.
  - Buckets contain data entries.
- *Hashing function* **h**: **h**(*r*) = bucket in which (data entry for) record *r* belongs. **h** looks at the *search key* fields of *r.*
  - *No need for "index entries" in this scheme.*

Alternatives for contents of an Index

- In an index entry k* we can store:
  - Alternative 1: The actual data record with key value **k**, or
  - Alternative 2: <**k**, rid of data record with search key value **k**>, or
  - Alternative 3 <**k**, list of rids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Alternatives for Data Entries

- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)
  - If data records are very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large.

Alternatives for Data Entries

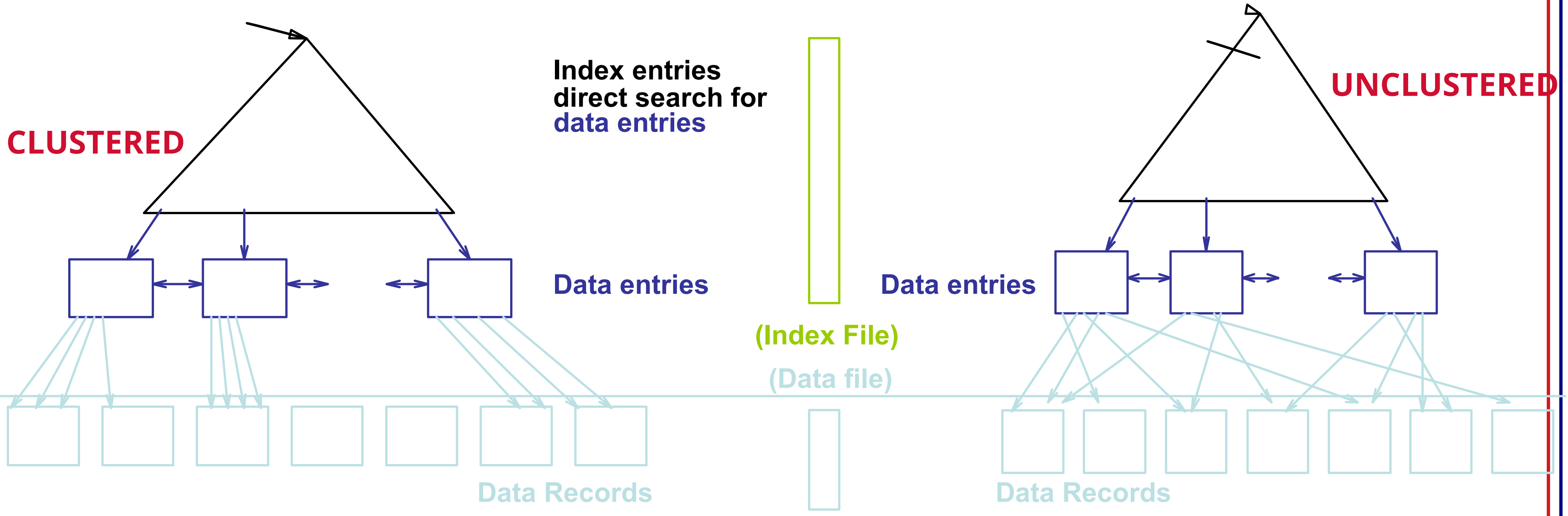
- Alternatives 2 and 3:
  - Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Index Classification

- *Primary* vs. *secondary*: If search key contains primary key, then it is called primary index.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *un-clustered*: If the order of the data records is the same as, or `close to', the order of the data entries, then the index is called a clustered index: else un-clustered.
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

These are only estimates to show the overall trends!

Comparing File Organizations

- Heap files (random order; insert at eof)
- Sorted files, sorted on <*age, sal*>
- Clustered B+ tree file, Alternative (1), search key <*age, sal*>
- Heap file with unclustered B + tree index on search key <*age, sal*>
- Heap file with unclustered hash index on search key <*age, sal*>

Operations to Compare

- S
c
a
n
:

F
e
t
c
h

a
l

Assumptions

- Heap Files: Equality selection on key; exactly one match.
- Sorted Files: Files compacted after deletions.
- Indexes: Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy => File size = 1.25 data size
  - Tree: 67% occupancy (this is typical).
    - Implies file size = 1.5 data size
- Scans: Leaf levels of a tree-index are chained.
  - Index data-entries plus actual file scanned for unclustered indexes.
- Range searches:We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

Cost of Operations

B: The number of data pages
R: Number of records per page
D: (Average) time to read or write disk page

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	$D\log_2 B$	$D(\log_2 B$ + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B$ + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D(\log_F 0.15B$ + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

These are estimates using many simplifying assumptions

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Choice of Indexes

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree?
- One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!
  - For now, we discuss simple 1-table queries.
- Before creating an index, must also consider the impact on updates in the workload!

Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Examples of Clustered Indexes

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

B+ tree index on E.age can be used to get qualifying tuples.

- How selective is the condition?
- Is the index clustered?

Consider the GROUP BY query.

- If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
- Clustered *E.dno* index may be better!

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

Equality queries and duplicates:

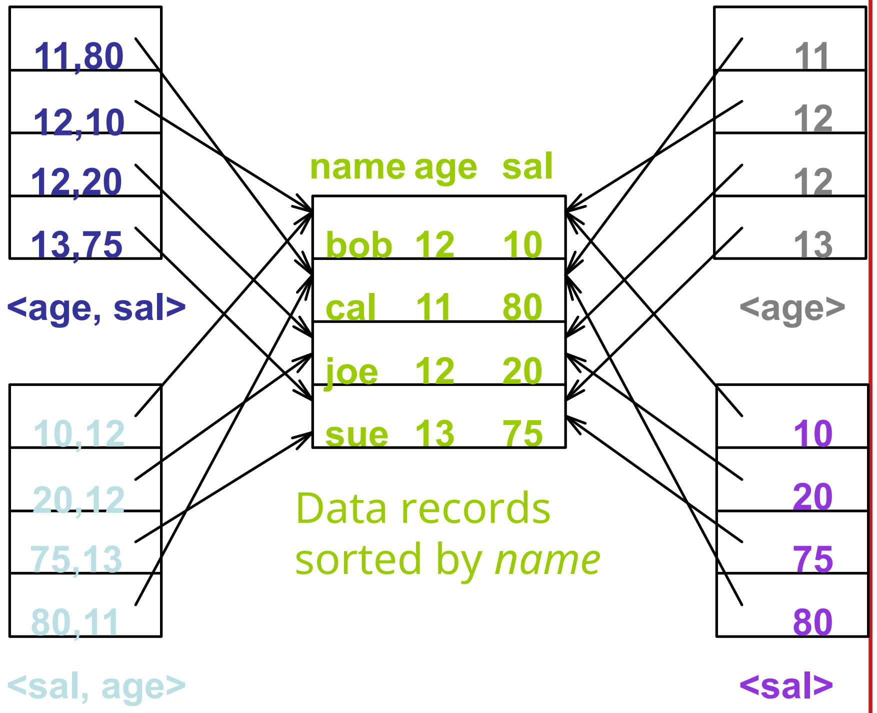
- Clustering on *E.hobby* helps!

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Indexes with Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. e.g.:age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



Data records sorted by *name*

Data entries in index sorted by *<sal,age>*

Data entries sorted by *<sal>*

Composite Search Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
  - Choice of index key orthogonal to clustering etc.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the tables involved if a suitable index is available.

<E.dno>

```
SELECT  E.dno, COUNT(*)
FROM  Emp E
GROUP BY  E.dno
```

<E.dno,E.sal>

Tree index!

```
SELECT  E.dno, MIN(E.sal)
FROM  Emp E
GROUP BY  E.dno
```

<E. age,E.sal>

or

<E.sal, E.age>

Tree index!

```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

Index-Only Plans (Contd.)

- Index-only plans are possible if the key is <dno,age> or we have a tree index with key <age,dno>
  - Which is better?
  - What if we consider the second query?

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age=30
GROUP BY E.dno
```

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>30
GROUP BY E.dno
```

Index-Only Plans (Contd.)

- Index-only plans can also be found for queries involving more than one table;

<E.dno>

```
SELECT  D.mgr
FROM  Dept D, Emp E
WHERE  D.dno=E.dno
```

<E.dno,E.eid>

```
SELECT  D.mgr, E.eid
FROM  Dept D, Emp E
WHERE  D.dno=E.dno
```

Summary

- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.

Summary (Contd.)

- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to *indexing technique* used to locate data entries with a given key value.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.

Summary (Contd.)

- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.

Database Index & Performance Optimization

Bipin C. DESAI

```
MariaDB [test]> desc member;
+-----------------+------------------+------+-----+---------------------+----------------+
| Field           | Type             | Null | Key | Default             | Extra          |
+-----------------+------------------+------+-----+---------------------+----------------+
| userid          | int(10) unsigned | NO   | PRI | NULL                | auto_increment |
| username        | varchar(45)      | NO   | UNI |                     |                |
| password        | varchar(45)      | NO   |     |                     |                |
| salutation      | varchar(45)      | NO   |     |                     |                |
| lastname        | varchar(64)      | NO   |     |                     |                |
| middle_name     | varchar(30)      | NO   |     |                     |                |
| firstname       | varchar(64)      | NO   |     |                     |                |
| organization    | varchar(120)     | NO   |     |                     |                |
| department      | varchar(255)     | NO   |     |                     |                |
| address         | varchar(255)     | NO   |     |                     |                |
| city            | varchar(70)      | NO   |     |                     |                |
| province        | varchar(70)      | NO   |     |                     |                |
| country         | varchar(70)      | NO   |     |                     |                |
| postcode        | varchar(10)      | NO   |     |                     |                |
| email           | varchar(70)      | NO   |     |                     |                |
| fax             | varchar(70)      | NO   |     |                     |                |
| phone           | varchar(70)      | NO   |     |                     |                |
| status          | varchar(45)      | NO   |     |                     |                |
| register_date   | datetime         | NO   |     | 0000-00-00 00:00:00 |                |
| last_login_date | datetime         | NO   |     | 0000-00-00 00:00:00 |                |
| last_conference | int(10)          | YES  |     | NULL                |                |
| receive_email   | varchar(20)      | NO   |     | NULL                |                |
+-----------------+------------------+------+-----+---------------------+----------------+
```

Create INDEX

```
MariaDB [test]> create index cntr_indx on member(country);
Query OK, 0 rows affected (0.061 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Creating an index on multiple columns

MariaDB/MySQL allows composite(multi-column) index
(up to 16 columns)
  Usually 2 or 3 columns are sufficient

```
CREATE INDEX index_name ON TableName (Col1, COL2, COl3);
```

Drop INDEX

Drop index syntax

```
ALTER TABLE table_name DROP INDEX index_name;
```

Rename index syntax

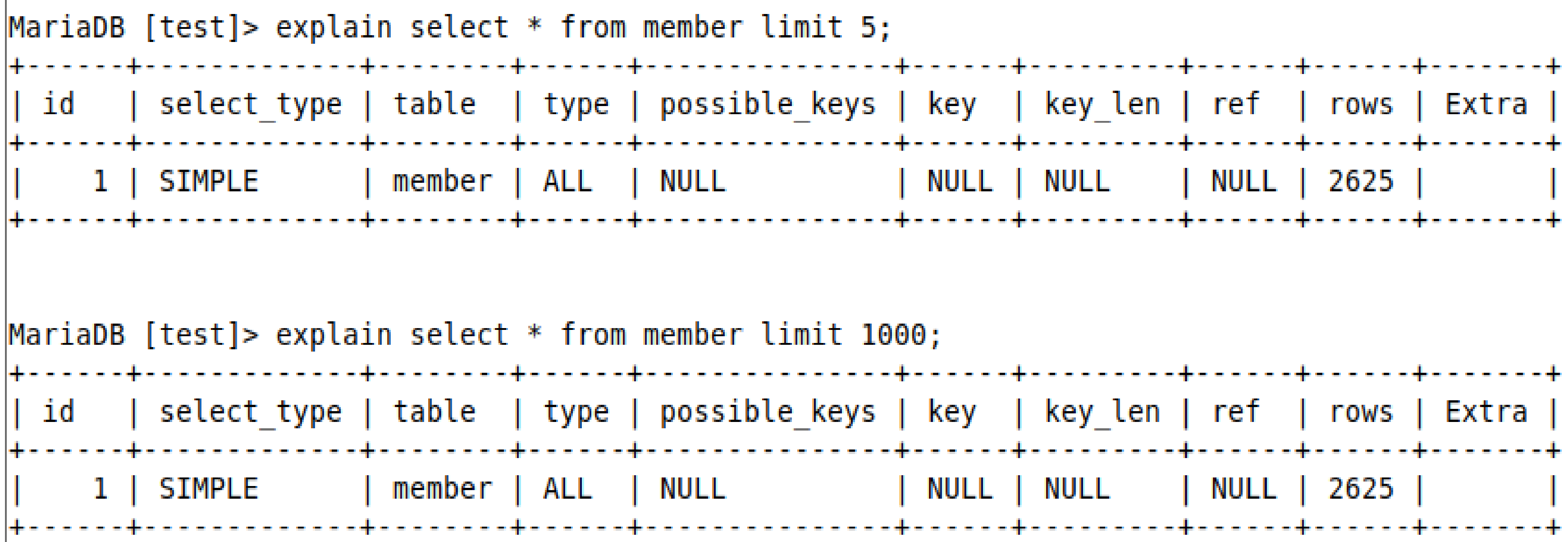
```
ALTER TABLE table_name RENAME INDEX index_name
    TO new_index_name;
```

Show indexes syntax

```
SHOW INDEX FROM tableName;
```

EXPLAIN

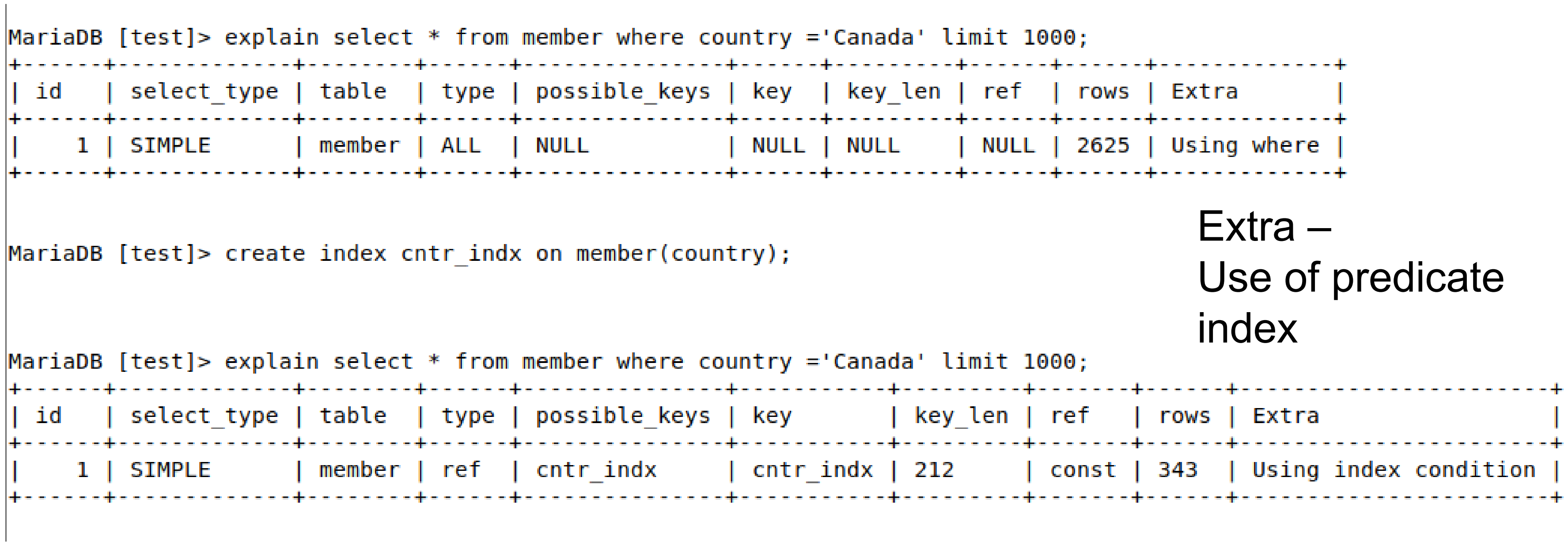
One can use EXPLAIN to see how the DB executes a DML statement
DML statements are:
SELECT, DELETE, INSERT, REPLACE, and UPDATE statements.
EXPLAIN gives execution plan information from the built-in DB optimizer



```
MariaDB [test]> explain select * from member limit 5;
+------+-------------+--------+------+---------------+------+---------+------+------+-------+
| id   | select_type | table  | type | possible_keys | key  | key_len | ref  | rows | Extra |
+------+-------------+--------+------+---------------+------+---------+------+------+-------+
|    1 | SIMPLE      | member | ALL  | NULL          | NULL | NULL    | NULL | 2625 |       |
+------+-------------+--------+------+---------------+------+---------+------+------+-------+


MariaDB [test]> explain select * from member limit 1000;
+------+-------------+--------+------+---------------+------+---------+------+------+-------+
| id   | select_type | table  | type | possible_keys | key  | key_len | ref  | rows | Extra |
+------+-------------+--------+------+---------------+------+---------+------+------+-------+
|    1 | SIMPLE      | member | ALL  | NULL          | NULL | NULL    | NULL | 2625 |       |
+------+-------------+--------+------+---------------+------+---------+------+------+-------+
```

The null for the “possible_keys” and “key” above are both NULL
This indicates that the DB does not have an index it can use
The DB will access 2625 rows to generate the result

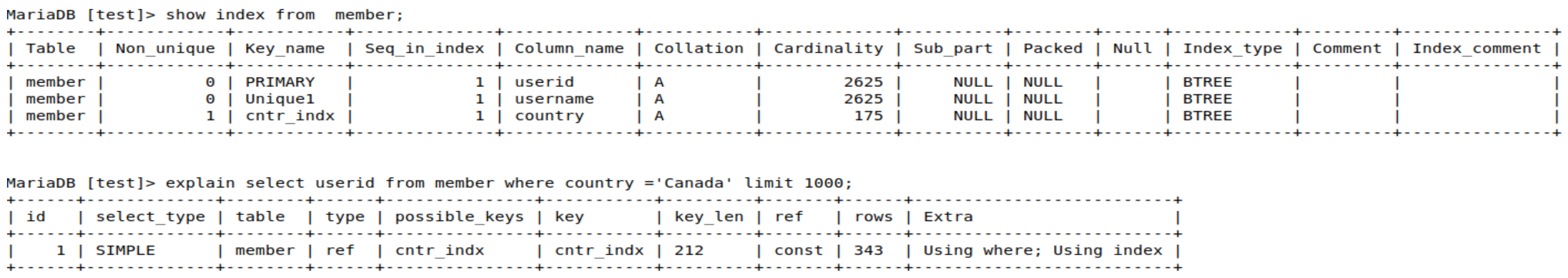


```
MariaDB [test]> explain select * from member where country ='Canada' limit 1000;
+------+-------------+--------+------+---------------+------+---------+------+------+-------------+
| id   | select_type | table  | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+------+-------------+--------+------+---------------+------+---------+------+------+-------------+
|    1 | SIMPLE      | member | ALL  | NULL          | NULL | NULL    | NULL | 2625 | Using where |
+------+-------------+--------+------+---------------+------+---------+------+------+-------------+

MariaDB [test]> create index cntr_indx on member(country);

MariaDB [test]> explain select * from member where country ='Canada' limit 1000;
+------+-------------+--------+------+---------------+-----------+---------+-------+------+-----------------------+
| id   | select_type | table  | type | possible_keys | key       | key_len | ref   | rows | Extra                 |
+------+-------------+--------+------+---------------+-----------+---------+-------+------+-----------------------+
|    1 | SIMPLE      | member | ref  | cntr_indx     | cntr_indx | 212     | const | 343  | Using index condition |
+------+-------------+--------+------+---------------+-----------+---------+-------+------+-----------------------+
```

Extra –
Use of predicate index



```
MariaDB [test]> show index from  member;
+--------+------------+-----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table  | Non_unique | Key_name  | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+--------+------------+-----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| member |          0 | PRIMARY   |            1 | userid      | A         |        2625 |     NULL | NULL   |      | BTREE      |         |               |
| member |          0 | Unique1   |            1 | username    | A         |        2625 |     NULL | NULL   |      | BTREE      |         |               |
| member |          1 | cntr_indx |            1 | country     | A         |         175 |     NULL | NULL   |      | BTREE      |         |               |
+--------+------------+-----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+

MariaDB [test]> explain select userid from member where country ='Canada' limit 1000;
+------+-------------+--------+------+---------------+-----------+---------+-------+------+--------------------------+
| id   | select_type | table  | type | possible_keys | key       | key_len | ref   | rows | Extra                    |
+------+-------------+--------+------+---------------+-----------+---------+-------+------+--------------------------+
|    1 | SIMPLE      | member | ref  | cntr_indx     | cntr_indx | 212     | const | 343  | Using where; Using index |
+------+-------------+--------+------+---------------+-----------+---------+-------+------+--------------------------+
```