**DBMS (R20) 2nd Year 1st Semester 2021-22 AY**

**Unit-5 Syllabus**

**Transaction Concepts**: Transaction State, Transaction Properties, Concurrent Executions, Serializability, Anomalies due to Interleaved Execution, Precedence Graph to Test for Conflict Serializability, Failure Classification, Storage, Recovery and Atomicity, ARIES Recovery algorithm.

**Indexing Techniques**: File Organizations and Indexing, Cluster Indexes, Primary and Secondary Indexes, Index Data Structures: Hash Based Indexing, Tree Based Indexing, Comparison of File Organizations, B+ Trees: Search, Insert, Delete algorithms.

***Transaction Concept***:A ***transaction*** is one execution of a user program on a consistent database. Executing the same program again results in a second transaction.

Ex: Booking railway reservation, shopping online, funds transfer etc.

-A transaction as seen by the DBMS is a series of read and write operations on database objects.

Ex: Consider the following transaction T1.

R(A)

A := A – 10000

W(A)

R(B)

B := B + 1000

W(B)

T1 = { R(A), W(A), R(B), W(B) }

DBMS interleaves actions of several transactions to increase the system performance. Interleaving actions of different transactions result in a schedule.

***Schedule:*** A set of actions from a set of transactions form a schedule. The actions of a transaction in a schedule should appear in the same order as they appear in the transaction.

The result of execution of any schedule should result in an equivalent serial execution of transactions.

Ex :

Consider T1 = { R(A), R(B), W(B) } and T2 = { R(C), W(C) }

Schedule S1 = { R(A), R(B), R(C), W(B), W(C) }

Schedule S2 = { R(A), R(C), R(B), W(C),W(B) }

S1 and S2 are schedules formed due to two different interleaving.

***Complete Schedule***: A schedule in which either commit or abort is specified as the last action for every transaction is called complete schedule.

***Serial Schedule***: A Schedule in which transactions run one after the other without any interleaving is called a serial schedule.

***Serializable Schedule***: Consider the following transactions.

T1: {R(A), W(A), R(B), W(B), commit}

T2: {R(A), W(A), R(B), W(B), commit}

S1={ T1R(A), T1W(A), T2R(A), T2W(A), T1R(B), T1W(B), T2R(B), T2W(B), T1Commit, T2Commit}

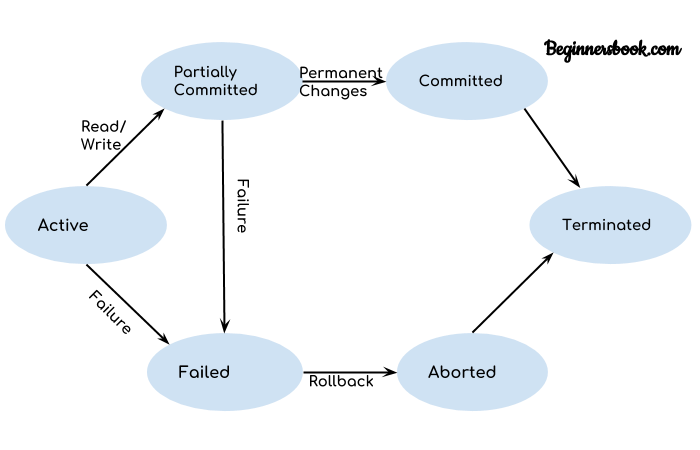
S2={ T1R(A), T1W(A), T1R(B), T1W(B), T1Commit, T2R(A), T2W(A), T2R(B), T2W(B), T2Commit}

S1 is a schedule formed by interleaving of T1 and T2. S2 is the schedule formed by serial execution of transactions T1 and T2. Both S1 and S2 will result in same final state of database. S1 is converted as S2 by repeatedly swapping actions of transactions. As the effect of S1 is equivalent to that of serial schedule S2, S1 is called Serializable schedule.

A ***serializable schedule*** over a set S of committed transactions is a schedule whose effect of running on a consistent database is equivalent to that of a serial schedule.

***Transaction States***:

Following diagram shows various states that a transaction goes through. When a transaction is submitted, it first enters the ***active*** state. A transaction may fail even before starting execution or during execution. If a transaction is failed in execution, it enters ***failed*** state. When a transaction has completed reading and writing database objects, it enters ***partially committed*** state. If changes made by transaction are not allowed to be permanent, then the transaction enters ***failed*** state. If the changes are made permanent, it enters ***committed*** state. When a failed transaction is rolled back, it enters ***abort*** state. Finally any transaction enters ***terminated*** state and leaves the system.



State Diagram of a transaction

***Transaction Properties***: Any transaction that runs on a consistent database should follow four properties. They are: **1**. Atomicity **2**.Consistency **3**.Isolation **4**. Durability

***Atomicity***: Let a transaction T transfers funds from account A to account B. If the transaction failed after deducting money from one account without crediting it into the second account, then the consistency of the database is lost. Transactions may fail in the middle for various reasons. When a running transaction is aborted, then all actions carried out by that transaction until then should be undone. Either all actions of a transaction should complete or none. This ***all*** or ***nothing*** property is called ***Atomicity***.

***Consistency***: When a transaction run, the database changes from one consistent state to a new consistent state. For example, in funds transfer transaction, sum of funds in both the accounts should be same before and after transfer. If this consistency requirement is achieved by transaction, the database moves from one consistent state to other. ***Database consistency is followed by transaction consistency***. If a transaction is consistent by itself, running such a transaction will lead the database from one consistent state to a different state.

***Isolation***: When several transactions run in parallel, DBMS must provide an environment that each transaction must consider as if it is the only transaction running. Achieving this running alone state is called ***isolation***. When several transactions are allowed to run in parallel, DBMS must schedule the transactions in such a way that anomalies do not occur during concurrent execution.

***Durability***: When a transaction has successfully completed and committed, it is the responsibility of DBMS to see that changes done by that transaction should survive crashes and be reflected on the disk. This property is called ***durability***.

**Implementation of Atomicity and Durability**: Transaction atomicity is ensured by removing partial or uncommitted actions done by transaction. Durability is implemented by ensured that changes done by committed transactions as made permanent. These are implemented by recovery manager. During normal execution, recovery manager maintains a log of all modifications done by various transactions. Making use of this log and other structures atomicity and durability are implemented.

DBMS has enough components that coordinate with each other to achieve above properties. Concurrency control manager and recovery manager of DBMS help in achieving these properties.

**Consider the following interleaving instances**:

T1 T2 T1 T2 T1 T2 T1 T2 T1 T2

R(A) W(A) R(A) W(A)

R(A)

R(A) W(A) R(A) R(B) W(A)

1. (b) (c) (d) (e)

T1 T2

W(A)

W(B)

(f)

Above schedules from (a) to (f) shows one action from T1 and T2 each. In all schedules, T1’s action is followed by T2’s action. In schedules (a), (d) and (f), the order of execution of the actions does not matter. i.e., actions can be swapped without any anomaly. Where as in schedules (b), (c) and (e), if the order of execution changes, then the result of execution of the schedule also changes.

It can be observed that two actions from two transactions ***conflict*** if they are on the same database object and if at least one of them is a ‘write’.

- A DBMS must allow the execution of serializable and recoverable schedules on any consistent database whose serial order is equivalent to the order of submission of transactions.

***Conflicts / Anomalies Due to Interleaving Execution***: When conflicting actions are interleaved, following anomalies occur.

(i) RW conflict (Unrepeatable Read)

(ii) WR conflict (Reading Uncommitted Data)

(iii) WW conflict (Overwriting Uncommitted Data)

i) ***Unrepeatable Read***:

Let a transaction T1 has read the value of a database object A. Subsequently, a second transaction T2 has modified the value of A while T1 is still active. When T1 reads the value of A again, it will get a different value even though it has not modified the value of A. This problem is called ***unrepeatable read***.

ii) ***Reading Uncommitted Data***:

Let a transaction T1 has modified a database object A. Subsequently, a 2nd transaction T2 has read the value of A that is modified by T1 while T1 is still active and not committed. If T1 has rolled back for some reason, then the new value produced by T1 is invalid and T2 proceeds with a wrong value of A. Such a read is called ***dirty read***. i.e., T2 has read uncommitted data

iii) ***Overwriting Uncommitted Data***:

Let a transaction T1 has modified the value of A. Subsequently, a 2nd transaction T2 has also modified A overwriting the value produced by T1 while T1 is still active. This problem is called ***lost update*** problem or ***overwriting uncommitted data***.

The reason for all above anomalies is allowing the interleaving of conflicting actions. To disallow such interleaving, a mechanism called ***locking*** is used.

- A ***lock*** is a book keeping object associated with database object.

- It is a kind of permission given to transactions to access database objects.

- When a transaction has to ***read*** a database object, it must acquire a ***shared lock***.

- When a transaction has to ***read and modify*** a database object, it must acquire an ***exclusive lock***.

***Compatibility of locks***: It defines whether a lock requested by a transaction can be granted by lock manager. When a transaction T1 holds a shared lock on a database object A, other transaction T2 can be given a shared lock on A, but not an exclusive lock. When a transaction T1 holds an exclusive lock on a database object A, other transaction T2 can’t be given either a shared lock or an exclusive lock on the same object.

On a database object, two shared locks are compatible. An exclusive lock is not compatible with any lock.

***Strict 2 Phase Locking (Strict 2PL)***:

This is a protocol to control the anomalies due to interleaved transactions.

This protocol is stated in 2 rules:

1. A transaction must get either a shared lock or an exclusive lock to read or write a database object.
2. All locks held by transaction are released when the transaction is completed.

This protocol prevents anomalies using compatibility restrictions of locks. When a transaction does not get a requested lock, it is suspended until it is available.

**Test for Serializability**: To test for serializabilty of a schedule, precedence graph is used.

***Conflict Equivalence***:

- Two schedules S1 and S2 are conflict equivalent if they order every pair of conflicting actions in the same way. Non conflicting actions may be ordered in any way.

- A schedule S is conflict serializable if it is conflict equivalent to some serial schedule.

***Precedence Graph***:

It is used to verify whether a given schedule is conflict serializable or not. It is drawn as follows.

i) Draw a node for each transaction in the schedule.

ii) Draw an arc from Ti to Tj if an action ***ai*** of Ti  precedes and conflicts with an action ***aj*** of Tj.

Precedence graph for schedule S1

Consider the following schedules:

S1={WA(T2), RA(T1)} S2={ RA(T1), WA(T2), WA(T1), WA(T3)}

Precedence graph for Schedule S2

If the precedence graph of a schedule contains a cycle, the schedule is not conflict serializable.

***Classification of failures***: There are various types of failures that may occur in a system. Each failure needs to be dealt with in a different manner.

i)***Transaction Failure***: There are two types of failure that may cause a transaction to fail. They are:

---***Logical Error***: Sometimes internal errors like bad input, data not found, overflow or resource limit exceeded may cause transaction failure.

---***System Error***: The system has entered an undesirable state (Ex: Deadlock) so that transaction cannot continue with its normal execution.

ii)***System Crash***: There may be a hardware malfunction or a bug in the database software. Sometimes the operating system may cause loss of content of volatile storage and brings transaction to a halt. The content of non-volatile storage remains uncorrupted.

iii)***Disk Failure***: A disk block may lose its content either due to a head crash or a failure during a data transfer operation. Copies of data on other disks or archival backups are used to recover from failure.

After a failure has occurred, recovery manager consults the log for recovery related information.

**ARIES** (Algorithm for Recovery and Isolation Exploiting Semantics)is a recovery algorithm designed to work with steal and no-force approach. It contains the following phases:

***Analysis***: Identifies dirty pages in the buffer pool and active transactions at the time of crash.

***Redo***: Repeats all actions starting from an appropriate point in the log and restores database to what it was at the time of crash.

***Undo***: Undoes the actions of uncommitted transactions.

To perform these steps, recovery manager uses ***log***.

***LOG***: Is the history of actions executed by the DBMS. It is a file of records stored in a stable storage that survives crashes. Log is also called the ***Trail*** or ***Journal***.

LSN LOG

10 update: T1 writes P5

20 update: T2 writes P3

30 T2 Commit

40 T2 End

50 update: T3 writes P1

60 update: T3 writes P3

X CRASH, RESTART

Execution History with a Crash

-Every log record has a unique id called ***Log Sequence Number*** (LSN).

-A log record is written for the following actions: updating a page, commit, abort, end and undoing an update.

-when an action described by an update log record is undone, a C***ompensation Log Record*** (CLR) is written.

-Each page is associated with a ***page LSN*** which is the LSN of the most recent log record that describes a change to this page.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| prev LSN | Trans ID | Type | Page ID | Length | Offset | Before Image | After Image |

Common to all records Fields specific to update records

Contents of an update log record

***prev LSN***field is used to link all records written by a transaction as a linked list. As log records are written by several transactions in the same log, log records written by a particular transaction at different times are scattered along the log. Hence, ***prev LSN*** is used to link them.

***Trans ID***: This field is the id of the transaction writing the log record. ***Type*** field is used to specify the type of log record. ***Page ID*** is the ID of the page modified by the transaction writing the log record.

***Length*** is the number of bytes modified. ***Offset*** is the position in the page where from the modification is done. ***Before image*** is the value of the changed bytes before change. ***After image*** is the value of the changed bytes after change. Every CLR written by a transaction contains a field ***undoNext LSN*** that indicates the LSN of the next log record that is to be undone for this transaction.

***Other Recovery Related Structures***: In addition to log, DBMS maintains two more tables that contains important recovery related information.

1) ***Transaction Table***: It contains an entry for each active transaction. The entry contains Transaction id, Status (progress commit or abort) and ***last LSN*** (LSN of most recent log record for this transaction).

2) ***Dirty page table***: It contain one entry for each dirty page. The entry contains a field ***recLSN*** which is the LSN of first log record that caused this page to become dirty.

|  |  |
| --- | --- |
| Page ID | Rec LSN |
| P500 |  |
| P600 |  |
| P505 |  |

Dirty Page Table

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Prev LSN | Trans ID | Type | Page ID | Length | Off Set | Before Image | After Image |
|  | T1000 | update | P500 | 3 | 21 | ABC | DEF |
|  | T2000 | update | P600 | 3 | 41 | HIJ | KLM |
|  | T2000 | update | P500 | 3 | 20 | GDE | QRS |
|  | T1000 | update | P505 | 3 | 21 | TUV | WXY |

|  |  |
| --- | --- |
| Trans ID | Last LSN |
| T1000 |  |
| T2000 |  |

Transaction Table

LOG

An instance of log and Transaction Table

***Write Ahead Log protocol***: (WAL)

-Before writing a page to disk, every update log record that describes a change to this page must be forced to stable storage. This is achieved by forcing all log records up to and including page LSN record to stable storage before writing the page to disk.

-WAL is the fundamental rule which ensures that a record of every change to the database is available while attempting to recover from a system crash.

-When a transaction is committed, the log tail (i.e the most recent portion of log) is forced to stable storage even if a no-force approach is used.

***Check pointing***:

When a system crash occurs, the recovery manager consults the log to determine which transactions need to be undone and which needs to be redone. To get this information, the entire log must be scanned which consumes a lot of time. To reduce this overhead, check points are used. During checkpoint, no updates are permitted and all modified buffer blocks are written to disk. It is a periodic process.

A checkpoint is performed as follows:

--All log records presently residing in main memory are sent to stable storage.

--All modified buffer blocks are output to disk.

--A log record of the form <checkpoint, **L**> is written to stable storage. **L** is a list of transactions active at the time of the checkpoint.

***Recovering From System Crash***: (Working of **ARIES**)

Undo LOG

A Oldest record of transactions

active at the time of crash.

Redo

B Smallest RecLSN in dirty page

table at the end of analysis.

Analysis

C Most recent check point.

CRASH (End of log)

Three phases of restart in ARIES

***Analysis phase***:

It begins by examining the most recent begin checkpoint record (i.e., C). This phase performs three tasks.

1. It determines the point in the log at which to start the Redo phase.

2. It determines the pages in the buffer pool that were dirty at the time of crash.

3. It identifies the transactions that were active at the time of crash and must be undone.

Analysis scans the log from the most recent begin checkpoint record to end of the log.

-If an ***End*** log record for a transaction T is found, then T is removed from transaction table.

-If any other log record is found for T, an entry for T is added to the transaction table if it was not already there. If there is an entry, the entry for T is modified as follows:

1. The LastLSN field is set to the LSN of this log record.

2. If the log record is a ***Commit*** record, the status is set to '***C***', otherwise it is set to '***U***'(undo).

-If a log record affecting page ***P*** is encountered and if ***P*** is not there in the dirty page table, an entry is made into this table and RecLSN is set to LSN of this log record.

***Redo phase***:

During this phase, ARIES reapplies the updates of all transactions.

-This phase begins with the log record that has the smallest RecLSN of all pages in the dirty page table.

-Starting from this record, Redo scans forward until the end of log.

-For each Redoable log record (i.e., update or CLR) encountered, Redo checks whether the logged action is to be redone or not.

A logged action is not redone in the following cases:

1) The effected page is not in the dirty page table.

2) The effected page is in dirty page table but, the RecLSN of this page is greater than the LSN of log record being checked.

3) The page LSN is greater than or equal to the LSN of log record being checked.

If the logged action is to be redone,

-The logged action is reapplied.

-The Page LSN of the page is set to LSN of the redone log record.

***Undo phase***:

This page scans backward from the end of the log. The goal of this phase is to undo the actions of all transactions active at the time of crash. The transactions are identified from the transaction table constructed during the analysis phase. Such transactions are called loser transactions.

-Consider the set of lastLSN values of all loser transactions. Call this set as "A".

-Undo repeatedly choose the largest LSN value in this set and processes it until "A" is empty.

-To process a log record:

1) ***If it is a CLR*** and the UndoNextLSN value is not null, the UndoNextLSN value is added to "A". If the UndoNextLSN value is null, an end log record is written for this transaction because it is completely undone.

2)***If it is an update record***, a CLR is written and the corresponding action is undone and the PrevLSN value in the update log record is added to "A".

When the set "A" became empty, the undo phase is complete. Restart is now complete and system can proceed with normal operations.

**Introduction to Indexing**: The basic abstraction of data in a DBMS is a collection of records (called a file), and each file consists of one or more pages. The files and access methods layer organizes data carefully to support fast access to desired subsets of records. A ***file organization*** is a method of arranging the records in a file when the file is stored on the disk. Each file organization makes certain operations efficient but other operations expensive.

***Indexing*** is a technique which helps when we have to access a collection of records in multiple ways.

Regarding data storage and retrieval, following points are to be kept in mind.

---Disks are most important external storage devices. They allow random access of data at fixed cost per page.

---Tapes are sequential access devices that allow reading data one age after the other. They are mostly used to archive data that is not needed on a regular basis.

---Each record in a file has a unique identifier called ***record- id*** or ***rid***.

**File Organizations and Indexing**: The simplest file structure is an unordered file or ***heap file***. Heap file supports retrieval of all records or a specific record using its rid.

---An ***index*** is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations. An index has a ***search key*** associated with it. We can retrieve all records that satisfy search conditions on search key field of the index.

---An index file is a collection of index records. A record stored in an index file is called a ***data entry***. A data entry with a search key value ***k*** has enough information to locate all data records with search key value *k*.

There are three alternatives regarding what to store as a data entry in an index.

1) A data entry ***k***\* is an actual data record with search key value k.

2) A data entry is a <*k*, *rid*> pair, where *rid* is the record id of a data record with a search key value *k*.

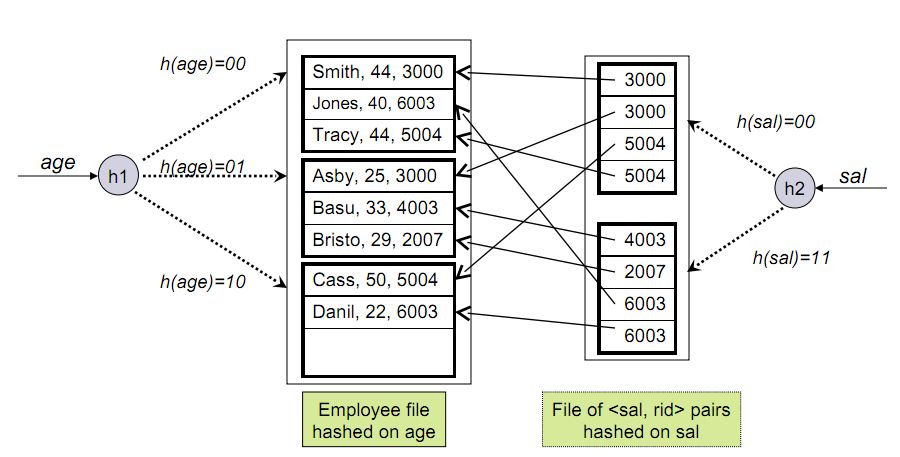
3) A data entry is a <*k*, *rid-list*> pair, where *rid-list* is a list of record ids of data records with search key value *k*.

**Clustered Indexes**: In a ***clustered index***, the order of data entries in index file is more or less same as the order of data records in the data file. An index that uses alternative (1) to store data entries is clustered. Indexes that use alternatives (2) or (3) are usually ***unclustered***. The cost of answering a range search query reduces drastically if the index is clustered. This is because ***rid***s of qualifying records point to contiguous memory locations.

**Primary and Secondary Indexes**: An index created on a set of fields that include the primary key is called ***primary index***. Other indexes are called ***secondary indexes***. An index that uses alternative (1) is called primary index and that uses alternatives (2) or (3) is called secondary index.

**Index Data Structures**: There are two approaches to organize data entries in an index file. One is to hash data entries on the search key field. Other is to build a tree-like data structure that directs a search for data entries.

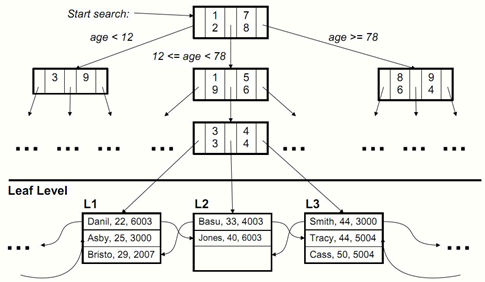
**Hash-based Indexing**:



Consider Employee records described with <Name, Age, Salary>. Let the records are hashed on field *Age*. In this approach, records in a file are grouped into ***buckets***. A bucket consists of a primary page and few additional pages linked in a chain. A ***hash function*** is applied to know the bucket in which a particular record presents. Once we know the bucket number, the primary page for the bucket can be retrieved in one or two disk I/Os.

In the figure above, data is stored in a file hashed on age. The hash function used in this example is quite simple. It returns two least significant bits of binary representation of age as the bucket number. The figure also shows index with search key *Sal* that contains <sal, rid> pairs as data entries. The *rid* component of a data entry in this second index is a pointer to a record with search key value *sal*.

**Tree-Based Indexing**: In this technique, records are organized using a tree-like data structure. The data entries are arranged in sorted order by search key value. A hierarchical search is done which direct the searches to correct page containing data entries. Each node in the figure below is a physical page. The lowest level of the tree, called the ***leaf level***, contains the data entries. All searches begin at the ***root*** and terminate at the leaf level. Non-leaf pages contain node pointers separated by search key values. The node pointer to the left of a key value k points to a sub-tree that contains entries less than k. The node pointer to the right of a key value k points to a sub-tree that contains entries greater than or equal to k.



The **B+ tree** is an index structure that is balanced in height. i.e., all paths from root to any leaf are of same length.

**Comparison of File Organizations**: Consider a collection of employee records and a composite search key <*age,sal*>. Let all selection operations are specified on these fields. In this section, the costs of following simple operations are compared for various file organizations.

**Scan**: Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. For locating each record on the page, CPU overhead per record is also involved.

**Search with Equality Selection**: Fetch all records that satisfy an equality selection. Pages that contain qualifying records must be fetched from disk. Qualifying records must be located within the retrieved pages.

**Search with Range Selection**: Fetch all records that satisfy a range selection.

**Insert a Record**: To insert a record into the file, identify the page in the file into which the new record must be inserted. The page must be fetched from disk, new record should be inserted and modified page must be written back. Based on the file organization, other pages may also need to be modified.

**Delete a Record**: Delete a specific record using it’s *rid*. Identify the page that contain the record, fetch it from disk, modify it and write it back. Based on the file organization, other pages may also need to be modified.

Various file organizations considered are:

* File of randomly ordered employee records, or heap file.
* File of employee records sorted on <age,sal>
* Clustered B+ tree file with search key <age,sal>
* Heap file with an un clustered B+ tree index on <age,sal>
* Heap file with an un clustered hash index on <age,sal>

**Cost Model**: The cost of different database operations are estimated in terms of execution time. Let B is the number of data pages, R is the number of records per page, D is the average time to read or write a disk page and C is the average time to process a record.

In hashed file organization, a *hash function* is used to map a record into a range of numbers. H is the time required to apply the hash function to a record. For tree-based indexing, *Fan-Out* (number of pointers to child nodes in a node), F is used whose value is typically 100 or more. I/O cost is of the order of milliseconds while H and C are of the order of nanoseconds. Hence, in total cost of an operation, major component is I/O cost.

Costs of different operations for various file organizations are depicted in the following table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| File Types | ***Scan*** | ***Equality Search*** | ***Range Search*** | ***Insert*** | ***Delete*** |
| ***Heap*** | BD+BRC | 0.5BD+0.5BRC | B(D+RC) | 2D+C | Cost of Search+D+C |
| ***Sorted*** | BD+BRC | D*log*2B+ C*log*2R | D(*log*2B+ Number of Matching Pages) | Cost of Search+ B(D+RC) | Cost of Search+ B(D+RC) |
| ***Clustered*** | 1.5B (D+RC) | D *log*F1.5B + C*log*2R | D( *log*F1.5B + Number of Matching Pages) | D *log*F1.5B + C*log*2R + D | D *log*F1.5B + C*log*2R + D |
| ***Unclustered Tree Index*** | BD(R+0.15) | D(1+*log*F0.15B) + C*log*2 6.7R | D( *log*F0.15B + Number of Matching Pages) | D(3+ *log*F0.15B) | D(1+*log*F0.15B) + C*log*2 6.7R +2D |
| ***Unclustered Hash Index*** | 0.125B (D+8RC) | H+2D+4RC | B(D+RC) | 4D+2C+H | H+4D+4RC |

In the table above, components of cost involving R and C can be ignored as they are small compared to disk I/O cost.

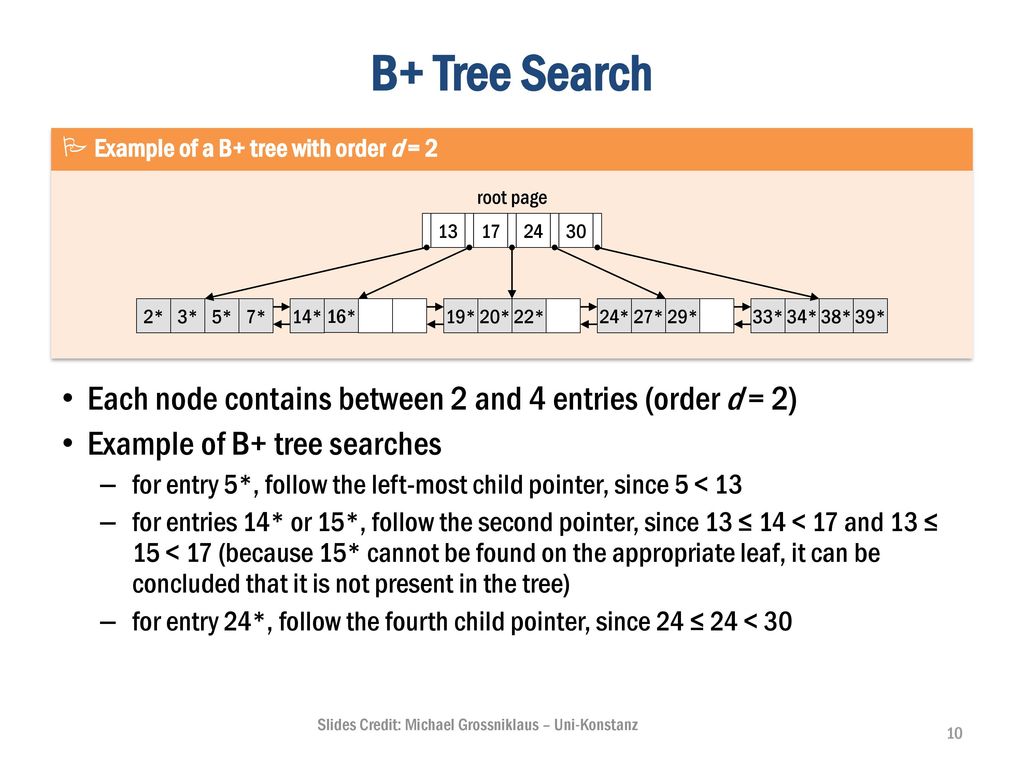
**B+ Trees**: It is a dynamic index structure which is more flexible and can adjust easily to inserts and deletes. The tree is balanced in height and all nodes other than leaf nodes are used only to direct the search operation. The leaf nodes contain data entries. The leaf nodes are organized as double linked list to allow sequential access. Important features of B+ tree are:

--- Insert and delete operations on the tree keep the tree balanced.

--- Searching for a record requires just a traversal from the root to the appropriate leaf. The length of the path from root to leaf is called ***height*** of the tree. Length of the path from root to any leaf is same.

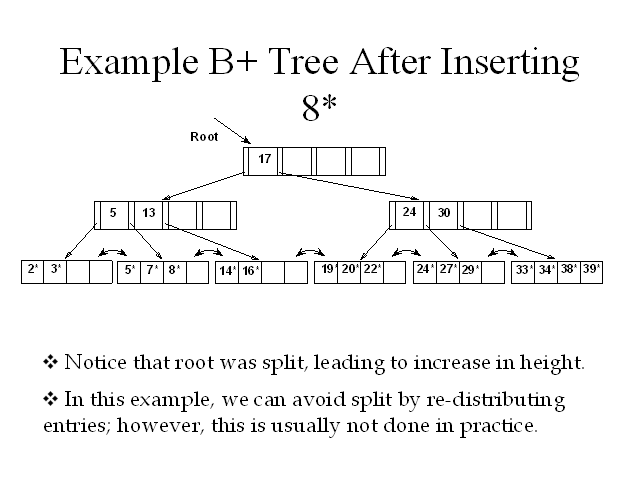
--- Every node contain *m* entries, where *d*<=*m*<=2*d*. The value *d* is a parameter of the B+ tree called ***order*** of the tree. The root node has m entries, where 1<=*m*<=2*d*.

**Search**: Consider the search operation on B+ Tree. Following B+ Tree is of order d=2. i.e., any node contains 2 to 4 entries. The root node has 4 entries and 5 pointers. All entries with search key values less than 13 are present in the left most leaf node. All entries with search key values greater than 30 are present in the right most leaf node. Other entries are present in the middle nodes.

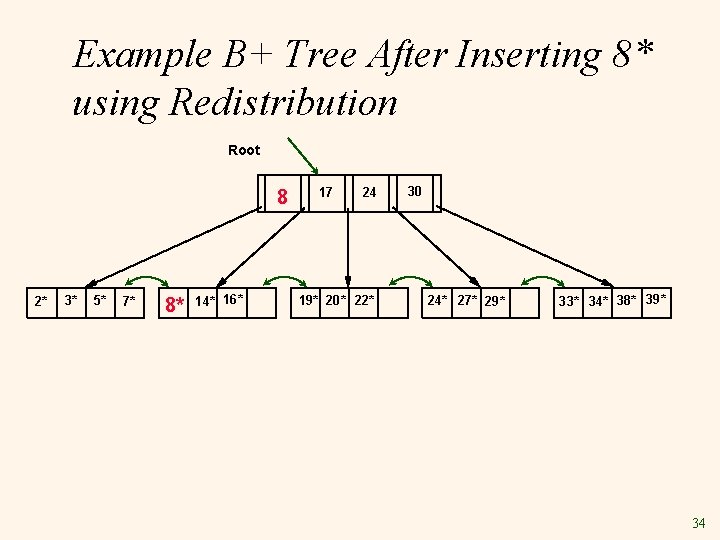


**Insert**: To insert a new data entry, traverse to appropriate leaf node starting from the root. If the node has space to hold an entry, then insert the new entry there. If not, a new node is to be created and its parent should be modified accordingly. Consider the tree used for search operation. If we want to insert 8\*, it should be inserted in the first node which is full. Hence, we create a new node between 1st and 2nd nodes. But 8\* alone can’t be placed in newly created node because a node must have minimum 2 nodes in order 2 binary tree. Hence, 5\*and 7\* from first node are removed and a new node is created with 3 entries, 5\*, 7\* and \*8.

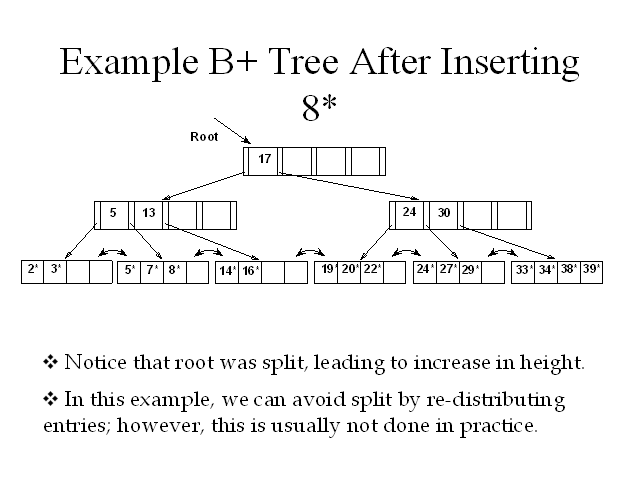
Now we need to create a new level to accommodate the child nodes. The entries in root node and other newly created index nodes are modified accordingly. This process results in a tree as shown below.

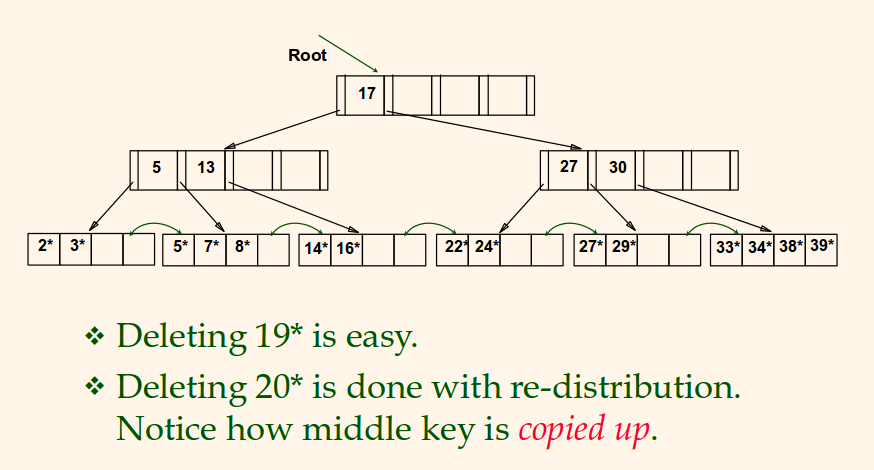


Instead of creating a new node and hence a new level, the existing leaf nodes can be redistributed. The 2nd node can be used to accommodate 8\*. After inserting 8\* into 2nd node, the root entries must be modified accordingly. The resulting tree is shown below.



**Delete**: Consider the deletion of the data entry 19\* from the B+ tree shown below. We can simply remove 19\* and nothing else is needed because even after removing 19\*, the node has two entries. Now, if we subsequently remove 20\*, then that node contains only 22\*. As its sibling contains three entries, we move 24\* to the node containing 22\*. The splitting key in the parent node is changed to 27 in place of 24. This modification is shown in second figure.





**B+ Tree**

**Deletion Algorithm**:

**STEP 1** Find leaf L containing (key,pointer) entry to delete

**STEP 2** Remove entry from L

**STEP 2a** If L meets the "half full" criteria, then we're done.

**STEP 2b** Otherwise, L has too few data entries.

**STEP 3** If L's right sibling can spare an entry, then move smallest entry in right sibling to L

**STEP 3a** Else, if L's left sibling can spare an entry then move largest entry in left sibling to L

**STEP 3b** Else, merge L and a sibling

**STEP 4** If merging, then recursively deletes the entry (pointing to L or sibling) from the parent.

**STEP 5** Merge could propagate to root, decreasing height