UNIT-V

TRANSACTION MANAGEMENT

What is a Transaction?

A transaction is an event which occurs on the database. Generally a transaction reads a value from the database or writes a value to the database. If you have any concept of Operating Systems, then we can say that a transaction is analogous to processes.

Although a transaction can both read and write on the database, there are some fundamental differences between these two classes of operations. A read operation does not change the image of the database in any way. But a write operation, whether performed with the intention of inserting, updating or deleting data from the database, changes the image of the database. That is, we may say that these transactions bring the database from an image which existed before the transaction occurred (called the **Before Image** or **BFIM**) to an image which exists after the transaction occurred (called the **After Image** or **AFIM**).

The Four Properties of Transactions

Every transaction, for whatever purpose it is being used, has the following four properties. Taking the initial letters of these four properties we collectively call them the **ACID Properties**. Here we try to describe them and explain them.

Atomicity: This means that either all of the instructions within the transaction will be reflected in the database, or none of them will be reflected.

Say for example, we have two accounts A and B, each containing Rs 1000/-. We now start a transaction to deposit Rs 100/- from account A to Account B.

```
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;
```

Fine, is not it? The transaction has 6 instructions to extract the amount from A and submit it to B. The AFIM will show Rs 900/- in A and Rs 1100/- in B.

Now, suppose there is a power failure just after instruction 3 (Write A) has been complete. What happens now? After the system recovers the AFIM will show Rs 900/- in A, but the same Rs 1000/- in B. It would be said that Rs 100/- evaporated in thin air for the power failure. Clearly such a situation is not acceptable.

The solution is to keep every value calculated by the instruction of the transaction not in any stable storage (hard disc) but in a volatile storage (RAM), until the transaction completes its last instruction. When we see that there has not been any error we do something known as a **COMMIT** operation. Its job is to write every temporarily calculated value from the volatile storage on to the stable storage. In this way, even if power fails at instruction 3, the post recovery image of the database will show accounts A and B both containing Rs 1000/-, as if the failed transaction had never occurred.

Consistency: If we execute a particular transaction in isolation or together with other transaction, (i.e. presumably in a multi-programming environment), the transaction will yield the same expected result.

To give better performance, every database management system supports the execution of multiple transactions at the same time, using CPU Time Sharing. Concurrently executing transactions may have to deal with the problem of sharable resources, i.e. resources that multiple transactions are trying to read/write at the same time. For example, we may have a table or a record on which two transaction are trying to read or write at the same time. Careful mechanisms are created in order to prevent mismanagement of these sharable resources, so that there should not be any change in the way a transaction performs. A transaction which deposits Rs 100/- to account A must deposit the same amount whether it is acting alone or in conjunction with another transaction that may be trying to deposit or withdraw some amount at the same time.

Isolation: In case multiple transactions are executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that they should not create any anomaly in the value stored at the sharable resource.

There are several ways to achieve this and the most popular one is using some kind of locking mechanism. Again, if you have the concept of Operating Systems, then you should remember the semaphores, how it is used by a process to make a resource busy before starting to use it, and how it is used to release the resource after the usage is over. Other processes intending to access that same resource must wait during this time. Locking is almost similar. It states that a transaction must first lock the data item that it wishes to access, and release the lock when the accessing is no longer required. Once a transaction locks the data item, other transactions wishing to access the same data item must wait until the lock is released.

Durability: It states that once a transaction has been complete the changes it has made should be permanent.

As we have seen in the explanation of the Atomicity property, the transaction, if completes successfully, is committed. Once the COMMIT is done, the changes which the transaction has made to the database are immediately written into permanent storage. So, after the transaction has been committed successfully, there is no question of any loss of information even if the power fails. Committing a transaction guarantees that the AFIM has been reached.

There are several ways Atomicity and Durability can be implemented. One of them is called **Shadow Copy**. In this scheme a database pointer is used to point to the BFIM of the database. During the transaction, all the temporary changes are recorded into a Shadow Copy, which is an exact copy of the original database plus the changes made by the transaction, which is the AFIM. Now, if the transaction is required to COMMIT, then the database pointer is updated to point to the AFIM copy, and the BFIM copy is discarded. On the other hand, if the transaction is not committed, then the database pointer is not updated. It keeps pointing to the BFIM, and the AFIM is discarded. This is a simple scheme, but takes a lot of memory space and time to implement.

If you study carefully, you can understand that Atomicity and Durability is essentially the same thing, just as Consistency and Isolation is essentially the same thing.

Transaction States

There are the following six states in which a transaction may exist:

Active: The initial state when the transaction has just started execution.

Partially Committed: At any given point of time if the transaction is executing properly, then it is going towards it COMMIT POINT. The values generated during the execution are all stored in volatile storage.

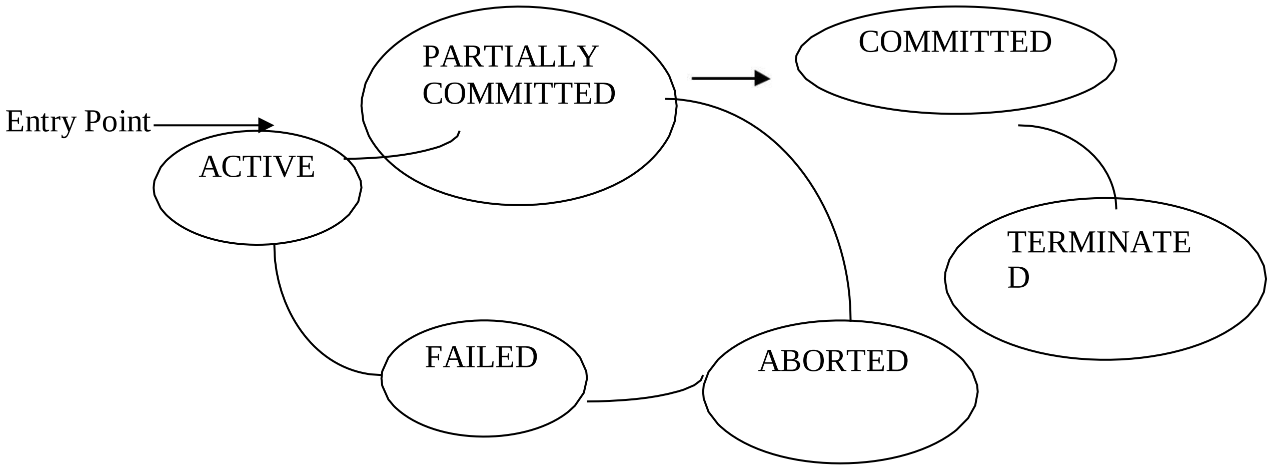
Failed: If the transaction fails for some reason. The temporary values are no longer required, and the transaction is set to **ROLLBACK**. It means that any change made to the database by this transaction up to the point of the failure must be undone. If the failed transaction has withdrawn Rs. 100/- from account A, then the ROLLBACK operation should add Rs 100/- to account A.

Aborted: When the ROLLBACK operation is over, the database reaches the BFIM. The transaction is now said to have been aborted.

Committed: If no failure occurs then the transaction reaches the COMMIT POINT. All the temporary values are written to the stable storage and the transaction is said to have been committed.

Terminated: Either committed or aborted, the transaction finally reaches this state.

The whole process can be described using the following diagram:



Concurrent Execution

A schedule is a collection of many transactions which is implemented as a unit. Depending upon how these transactions are arranged in within a schedule, a schedule can be of two types:

- **Serial:** The transactions are executed one after another, in a non-preemptive manner.
- **Concurrent:** The transactions are executed in a preemptive, time shared method.

In Serial schedule, there is no question of sharing a single data item among many transactions, because not more than a single transaction is executing at any point of time. However, a serial schedule is inefficient in the sense that the transactions suffer for having a longer waiting time and response time, as well as low amount of resource utilization.

In concurrent schedule, CPU time is shared among two or more transactions in order to run them concurrently. However, this creates the possibility that more than one transaction may need to access a single data item for read/write purpose and the database could contain inconsistent value if such accesses are not handled properly. Let us explain with the help of an example.

Let us consider there are two transactions T1 and T2, whose instruction sets are given as following. T1 is the same as we have seen earlier, while T2 is a new transaction.

```
    T1
Read A;
A = A – 100;
Write A;
Read B;
B = B + 100;
Write B;

    T2
Read A;
Temp = A * 0.1;
Read C;
C = C + Temp;
Write C;
```

T2 is a new transaction which deposits to account C 10% of the amount in account A.

If we prepare a serial schedule, then either T1 will completely finish before T2 can begin, or T2 will completely finish before T1 can begin. However, if we want to create a concurrent schedule, then some Context Switching need to be made, so that some portion of T1 will be executed, then some portion of T2 will be executed and so on. For example say we have prepared the following concurrent schedule.

```
      T1                  T2

Read A;
A = A – 100;
Write A;
                          Read A;
                          Temp = A * 0.1;
                          Read C;
                          C = C + Temp;
                          Write C;
Read B;
B = B + 100;
Write B;
```

No problem here. We have made some Context Switching in this Schedule, the first one after executing the third instruction of T1, and after executing the last statement of T2. T1 first deducts Rs 100/- from A and writes the new value of Rs 900/- into A. T2 reads the value of A, calculates the value of Temp to be Rs 90/- and adds the value to C. The remaining part of T1 is executed and Rs 100/- is added to B.

It is clear that a proper Context Switching is very important in order to maintain the Consistency and Isolation properties of the transactions. But let us take another example where a wrong Context Switching can bring about disaster. Consider the following example involving the same T1 and T2

T1	T2
Read A;	
A = A – 100;	
	Read A;
	Temp = A * 0.1;
	Read C;
	C = C + Temp;
	Write C;
Write A;	
Read B;	
B = B + 100;	
Write B;	

This schedule is wrong, because we have made the switching at the second instruction of T1. The result is very confusing. If we consider accounts A and B both containing Rs 1000/- each, then the result of this schedule should have left Rs 900/- in A, Rs 1100/- in B and add Rs 90 in C (as C should be increased by 10% of the amount in A). But in this wrong schedule, the Context Switching is being performed before the new value of Rs 900/- has been updated in A. T2 reads the old value of A, which is still Rs 1000/-, and deposits Rs 100/- in C. C makes an unjust gain of Rs 10/- out of nowhere.

Serializability

When several concurrent transactions are trying to access the same data item, the instructions within these concurrent transactions must be ordered in some way so as there are no problem in accessing and releasing the shared data item. There are two aspects of serializability which are described here:

Conflict Serializability

Two instructions of two different transactions may want to access the same data item in order to perform a read/write operation. Conflict Serializability deals with detecting whether the instructions are conflicting in any way, and specifying the order in which these two instructions will be executed in case there is any conflict. A **conflict** arises if at least one (or both) of the instructions is a write operation. The following rules are important in Conflict Serializability:

1. If two instructions of the two concurrent transactions are both for read operation, then they are not in conflict, and can be allowed to take place in any order.
2. If one of the instructions wants to perform a read operation and the other instruction wants to perform a write operation, then they are in conflict, hence their ordering is important. If the read instruction is performed first, then it reads the old value of the data item and after the reading is over, the new value of the data item is written. It the write instruction is performed first, then updates the data item with the new value and the read instruction reads the newly updated value.
3. If both the transactions are for write operation, then they are in conflict but can be allowed to take place in any order, because the transaction do not read the value updated by each other. However, the value that persists in the data item after the schedule is over is the one written by the instruction that performed the last write.

View Serializability:

This is another type of serializability that can be derived by creating another schedule out of an existing schedule, involving the same set of transactions. These two schedules would be called View Serializable if the following rules are followed while creating the second schedule out of the first. Let us consider that the transactions T1 and T2 are being serialized to create two different schedules

S1 and S2 which we want to be **View Equivalent** and both T1 and T2 wants to access the same data item.

1. If in S1, T1 reads the initial value of the data item, then in S2 also, T1 should read the initial value of that same data item.
2. If in S1, T1 writes a value in the data item which is read by T2, then in S2 also, T1 should write the value in the data item before T2 reads it.

3. If in S1, T1 performs the final write operation on that data item, then in S2 also, T1 should perform the final write operation on that data item.

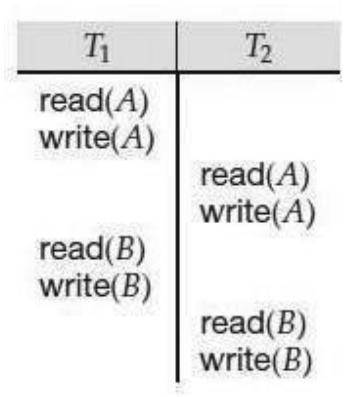
Let us consider a schedule *S* in which there are two consecutive instructions, *I* and *J* , of transactions *Ti* and *Tj* , respectively (*i* _= *j*). If *I* and *J* refer to different data
items, then we can swap *I* and *J* without affecting the results of any instruction
in the schedule. However, if *I* and *J* refer to the same data item *Q*, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

 I = read(*Q*), *J* = read(*Q*). The order of *I* and *J* does not matter, since the same value
of *Q* is read by *Ti* and *Tj* , regardless of the order.

 I = read(*Q*), *J* = write(*Q*). If *I* comes before *J* , then *Ti* does not read the value of *Q* that is written by *Tj* in instruction *J* . If *J* comes before *I*, then *Ti* reads
the value of *Q* that is written by *Tj*. Thus, the order of *I* and *J* matters.

 I = write(*Q*), *J* = read(*Q*). The order of *I* and *J* matters for reasons similar to those of the previous case.

4. *I* = write(*Q*), *J* = write(*Q*). Since both instructions are write operations, the order of these instructions does not affect either *Ti* or *Tj* . However, the value obtained by the next read(*Q*) instruction of *S* is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(*Q*) instruction after *I* and *J* in *S*, then the order of *I* and *J* directly affects the final value of *Q* in the database state that results from schedule *S*.



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Fig: Schedule 3—showing only the read and write instructions.

We say that *I* and *J* conflict if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation. To illustrate the concept of conflicting instructions, we consider schedule 3in Figure above. The write(*A*) instruction of *T*1 conflicts with the read(*A*) instruction of *T*2. However, the write(*A*) instruction of *T*2 does not conflict with the read(*B*) instruction of *T*1, because the two instructions access different data items.

Transaction Characteristics

Every transaction has three characteristics: *access mode*, *diagnostics size*, and *isolation level*. The **diagnostics size** determines the number of error conditions that can be recorded.

If the **access mode** is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE commands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. For transactions with READ ONLY access mode, only shared locks need to be obtained, thereby increasing concurrency.

The **isolation level** controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concurrency at the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Figure given below. In this context, *dirty read* and *unrepeatable read* are defined as usual. **Phantom** is defined to be the possibility that a transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself.

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged (see Section 19.3.1), and holds them until the end, according to Strict 2PL.

REPEATABLE READ ensures that *T* reads only the changes made by committed transactions, and that no value read or written by *T* is changed by any other transaction until *T* is complete. However, *T* could experience the phantom phenomenon; for example, while *T* examines all

Sailors records with *rating=1*, another transaction might add a new such Sailors record, which is missed by *T*.

A REPEATABLE READ transaction uses the same locking protocol as a SERIALIZABLE transaction, except that it does not do index locking, that is, it locks only individual objects, not sets of objects.

READ COMMITTED ensures that *T* reads only the changes made by committed transactions, and that no value written by *T* is changed by any other transaction until *T* is complete. However, a value read by *T* may well be modified by another transaction while *T* is still in progress, and *T* is, of course, exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that *every* SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself - a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects, and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance.
For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level, or even the READ UNCOMMITTED level, because a few incorrect or missing values will not significantly affect the result if the number of sailors is large. The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READONLY

When a transaction is started, the default is SERIALIZABLE and READ WRITE.

PRECEDENCE GRAPH

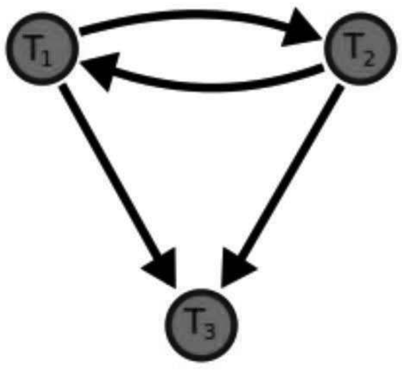
A precedence graph, also named conflict graph and serializability graph, is used in the context of concurrency control in databases.

The precedence graph for a schedule S contains:

A node for each committed transaction in S
An arc from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions.

Precedence graph example

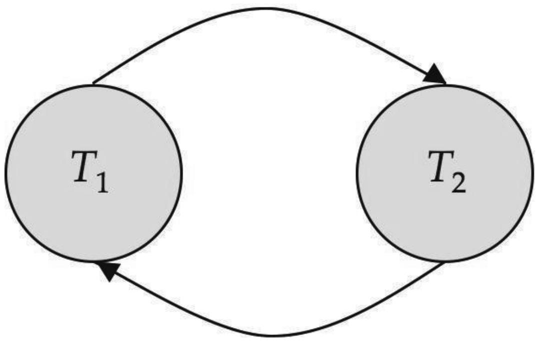
$$D = \begin{bmatrix} T1 & T2 & T3 \\ & R(B) & \\ R(C) & W(A) & \\ W(C) & & \\ R(D) & & \\ & & W(B) \\ W(D) & & W(A) \\ & & \end{bmatrix}$$



A precedence graph of the schedule D, with 3 transactions. As there is a cycle (of length 2; with two edges) through the committed transactions T1 and T2, this schedule (history) is not Conflict serializable.

The drawing sequence for the precedence graph:-

 For each transaction T_i participating in schedule S, create a node labelled T_i in the precedence graph. So the precedence graph contains T_1, T_2, T_3
 For each case in S where T_i executes a write_item(X) then T_j executes a read_item(X), create an edge (T_i --> T_j) in the precedence graph. This occurs nowhere in the above example, as there is no read after write.
3. For each case in S where T_i executes a read_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edge from T_1 to T_2.
4. For each case in S where T_i executes a write_item(X) then T_j executes a write_item(X), create an edge (T_i --> T_j) in the precedence graph. This results in directed edges from T_2 to T_1, T_1 to T_3, and T_2 to T_3.
5. The schedule S is conflict serializable if the precedence graph has no cycles. As T_1 and T_2 constitute a cycle, then we cannot declare S as serializable or not and serializability has to be checked using other methods.



TESTING FOR CONFLICT SERIALIZABILITY

1 A schedule is conflict serializable if and only if its precedence graph is acyclic.

2 To test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm.Cycle-detection algorithms exist which take order n2 time, where n is the number of vertices in the graph.

(Better algorithms take order n + e where e is the number of edges.)

3 If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. That is, a linear order consistent with the partial order of the graph.

For example, a serializability order for the schedule (a) would be one of either (b) or (c)

4 A serializability order of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph.

RECOVERABLE SCHEDULES

- Recoverable schedule — if a transaction Tj reads a data item previously written by a transaction Ti , then the commit operation of Ti must appear before the commit operation of Tj.

- The following schedule is not recoverable if T9 commits immediately after the read(A) operation.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T8 should abort, T9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

CASCADING ROLLBACKS

- Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T10 fails, T11 and T12 must also be rolled back.

- Can lead to the undoing of a significant amount of work

CASCADELESS SCHEDULES

- Cascadeless schedules — for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless
- Example of a schedule that is NOT cascadeless

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

CONCURRENCY SCHEDULE

- A database must provide a mechanism that will ensure that all possible schedules are both:
- Conflict serializable.
- Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency

- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur

- Testing a schedule for serializability after it has executed is a little too late!

- Tests for serializability help us understand why a concurrency control protocol is correct

- Goal – to develop concurrency control protocols that will assure serializability.

WEEK LEVELS OF CONSISTENCY

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable

- E.g., a read-only transaction that wants to get an approximate total balance of all accounts

- E.g., database statistics computed for query optimization can be approximate (why?)

- Such transactions need not be serializable with respect to other transactions

- Tradeoff accuracy for performance

LEVELS OF CONSISTENCY IN SQL

- Serializable — default

- Repeatable read — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.

- Read committed — only committed records can be read, but successive reads of record may return different (but committed) values.

- Read uncommitted — even uncommitted records may be read.

- Lower degrees of consistency useful for gathering approximate information about the database

- Warning: some database systems do not ensure serializable schedules by default

- E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

TRANSACTION DEFINITION IN SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
- Commit work commits current transaction and begins a new one.
- Rollback work causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
- Implicit commit can be turned off by a database directive
- E.g. in JDBC, connection.setAutoCommit(false);

RECOVERY SYSTEM

Failure Classification:

- Transaction failure :
- Logical errors: transaction cannot complete due to some internal error condition
- System errors: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash: a power failure or other hardware or software failure causes the system to crash.
- Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted as result of a system crash
- Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure: a head crash or similar disk failure destroys all or part of disk storage
- Destruction is assumed to be detectable: disk drives use checksums to detect failures

RECOVERY ALGORITHMS

- Consider transaction Ti that transfers $50 from account A to account B
- Two updates: subtract 50 from A and add 50 to B

 Transaction Ti requires updates to A and B to be output to the database.

 A failure may occur after one of these modifications have been made but before both of them are made.

 Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state

 Not modifying the database may result in lost updates if failure occurs just after transaction commits

 Recovery algorithms have two parts

1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures

2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

STORAGE STRUCTURE

 Volatile storage:

 does not survive system crashes

 examples: main memory, cache memory

 Nonvolatile storage:

 survives system crashes

 examples: disk, tape, flash memory,

non-volatile (battery backed up) RAM

 but may still fail, losing data

 Stable storage:

 a mythical form of storage that survives all failures

 approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
- copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies.

Block transfer can result in

- Successful completion
- Partial failure: destination block has incorrect information
- Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
- Execute output operation as follows (assuming two copies of each block):

1. Write the information onto the first physical block.

2. When the first write successfully completes, write the same information onto the second physical block.

3. The output is completed only after the second write successfully completes.

- Copies of a block may differ due to failure during output operation. To recover from failure:

1. First find inconsistent blocks:

1. Expensive solution: Compare the two copies of every disk block.

2. Better solution:

- Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
- Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
- Used in hardware RAID systems

2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

DATA ACCESS

- Physical blocks are those blocks residing on the disk.
- System buffer blocks are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
- input(B) transfers the physical block B to main memory.
- output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Each transaction Ti has its private work-area in which local copies of all data items accessed and updated by it are kept.
- Ti's local copy of a data item X is denoted by xi.
- BX denotes block containing X
- Transferring data items between system buffer blocks and its private work-area done by:
- read(X) assigns the value of data item X to the local variable xi.
- write(X) assigns the value of local variable xi to data item {X} in the buffer block.
- Transactions
- Must perform read(X) before accessing X for the first time (subsequent reads can be from local copy)
- The write(X) can be executed at any time before the transaction commits
- Note that output(BX) need not immediately follow write(X). System can performthe output operation when it seems fit.

Lock-Based Protocols

A lock is a mechanism to control concurrent access to a data item

Data items can be locked in two modes :

1. exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction.
2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

1) A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

2) Any number of transactions can hold shared locks on an item,

but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

3) If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Example of a transaction performing locking:

T_2: **lock-S***(A)*;
read *(A)*;
unlock*(A)*;
lock-S*(B)*;
read *(B)*;
unlock*(B)*;
display*(A+B)*

Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Consider the partial schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*.

Such a situation is called a **deadlock**.

1 To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

2. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

3. **Starvation** is also possible if concurrency control manager is badly designed. For example:

   a. A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.

   b. The same transaction is repeatedly rolled back due to deadlocks.

4.Concurrency control manager can be designed to prevent starvation.

THE TWO-PHASE LOCKING PROTOCOL

1.This is a protocol which ensures conflict-serializable schedules.

2.Phase 1: Growing Phase

   a.transaction may obtain locks

   b.transaction may not release locks

3.Phase 2: Shrinking Phase

   a.transaction may release locks

   b.transaction may not obtain locks

4.The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

5.Two-phase locking *does not* ensure freedom from deadlocks

6. Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
7. **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.
8. There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
9. However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

TIMESTAMP-BASED PROTOCOLS

1. Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
2. The protocol manages concurrent execution such that the time-stamps determine the serializability order.
3. In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
   a. W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully.
   b. R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully.

4. The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
5. Suppose a transaction T_i issues a **read**(Q)
   1. If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
      n Hence, the **read** operation is rejected, and T_i is rolled back.
   2. If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

6. Suppose that transaction T_i issues **write**(Q).
   1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
      n Hence, the **write** operation is rejected, and T_i is rolled back.
   2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
      n Hence, this **write** operation is rejected, and T_i is rolled back.
   3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Thomas' Write Rule

1. We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol i.e., Timestamp ordering Protocol . Let us consider schedule 4 of Figure below, and apply the timestamp-ordering protocol. Since $T27$ starts before $T28$, we shall assume that TS($T27$) < TS($T28$). The read(Q) operation of $T27$ succeeds, as does the write(Q) operation of $T28$. When $T27$ attempts its write(Q) operation, we find that TS($T27$) < W-timestamp(Q), since Wtimestamp(Q) = TS($T28$). Thus, the write(Q) by $T27$ is rejected and transaction $T27$ must be rolled back.

2. Although the rollback of $T27$ is required by the timestamp-ordering protocol, it is unnecessary. Since $T28$ has already written Q, the value that $T27$ is attempting to write is one that will never need to be read. Any transaction Ti with TS(Ti) < TS($T28$) that attempts a read(Q)will be rolled back, since TS(Ti)<W-timestamp(Q).

3. Any transaction Tj with TS(Tj) > TS($T28$) must read the value of Q written by $T28$, rather than the value that $T27$ is attempting to write. This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol.

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this: Suppose that transaction *Ti* issues write(*Q*).

1. If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.

2. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, this write operation can be ignored.

3. Otherwise, the system executes the write operation and setsW-timestamp(*Q*) to TS(*Ti*).

VALIDATION-BASED PROTOCOLS

Phases in Validation-Based Protocols

1) Read phase. During this phase, the system executes transaction Ti. It reads the values of the various data items and stores them in variables local to Ti. It performs all write operations on temporary local variables, without updates of the actual database.

2) Validation phase. The validation test is applied to transaction Ti. This determines whether Ti is allowed to proceed to the write phase without causing a violation of serializability.

If a transaction fails the validation test, the system aborts the transaction.

3) Write phase. If the validation test succeeds for transaction Ti, the temporary local variables that hold the results of any write operations performed by Ti are copied to the database. Read-only transactions omit this phase.

MODES IN VALIDATION-BASED PROTOCOLS

1. Start(Ti)
2. Validation(Ti)
3. Finish

MULTIPLE GRANULARITY.

multiple granularity locking (MGL) is a locking method used in database management systems (DBMS) and relational databases.

In MGL, locks are set on objects that contain other objects. MGL exploits the hierarchical nature of the contains relationship. For example, a database may have files, which contain pages, which further contain records. This can be thought of as a tree of objects, where each node contains its children. A lock on such as a shared or exclusive lock locks the targeted node as well as all of its descendants.

Multiple granularity locking is usually used with non-strict two-phase locking to guarantee serializability. The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction *Ti* that attempts to lock a node *Q* must follow these rules:

- Transaction *Ti* must observe the lock-compatibility function of Figure above.
- Transaction *Ti* must lock the root of the tree first, and can lock it in anymode.
- Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q* locked in either IX or IS mode.
- Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.
- Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (that is, *Ti* is two phase).
- Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of *Q* locked.

STORAGE AND INDEXING : Database file organization, file organization on disk, heap files and sorted files, hashing, single and multi-level indexes, dynamic multilevel indexing using B-Tree and B+ tree, index on multiple keys.

DATABASE FILE ORGANIZATION:

Records and Record Types:

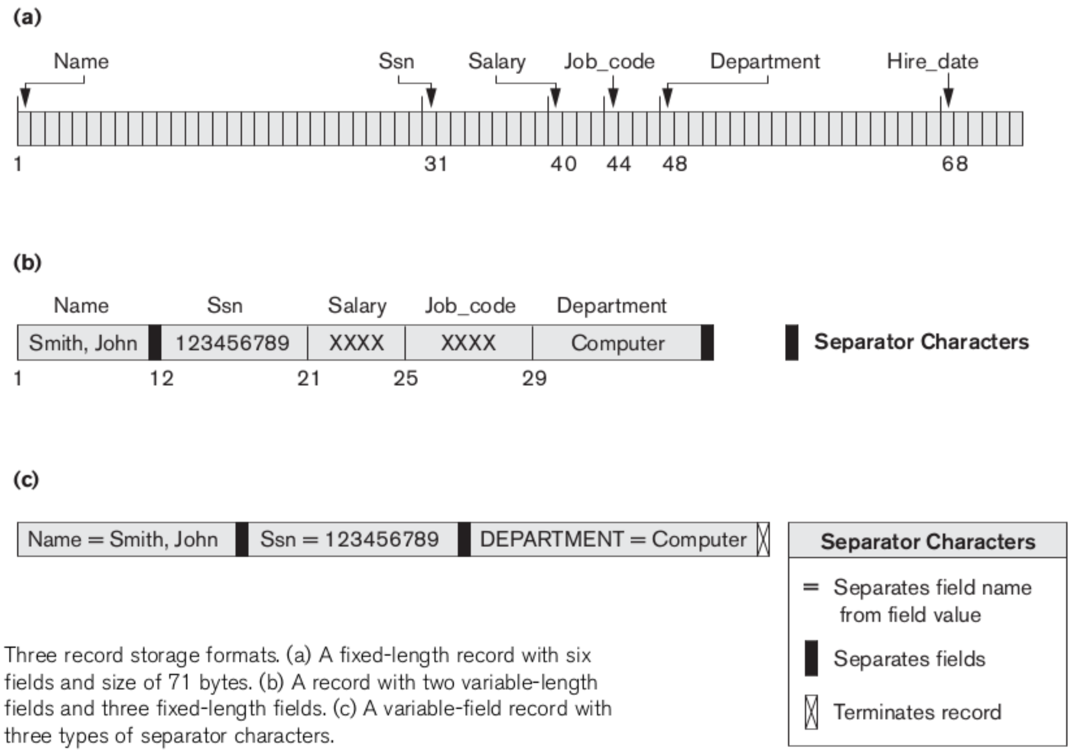
Data is usually stored in the form of records. Each record consists of a collection of related data values or items, where each value is formed of one or more bytes and corresponds to a particular field of the record. Records usually describe entities and their attributes. A collection of field names and their corresponding data types constitutes a record type or record format definition. A data type, associated with each field, specifies the types of values a field can take.

In recent database applications, the need may arise for storing data items that consist of large unstructured objects, which represent images, digitized video or audio streams, or free text. These are referred to as BLOBs (Binary Large Objects). A BLOB data item is typically stored separately from its record in a pool of disk blocks, and a pointer to the BLOB is included in the record.

Files, Fixed-length Records, and Variable-length Records:

A file is a *sequence* of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of fixed-length records. If different records in the file have different sizes, the file is said to be made up of variable-length records. A file may have variable-length records for several reasons:

1. The file records are of the same record type, but
   one or more of the fields are of varying size (variable-length fields) (or)
   one or more of the fields are optional (or)
   one or more of the fields may have multiple values for individual records; such a field is called a repeating field and a group of values for the field is often called a repeating group.
2. The file contains records of *different record types* and hence of varying size (mixed file).



Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

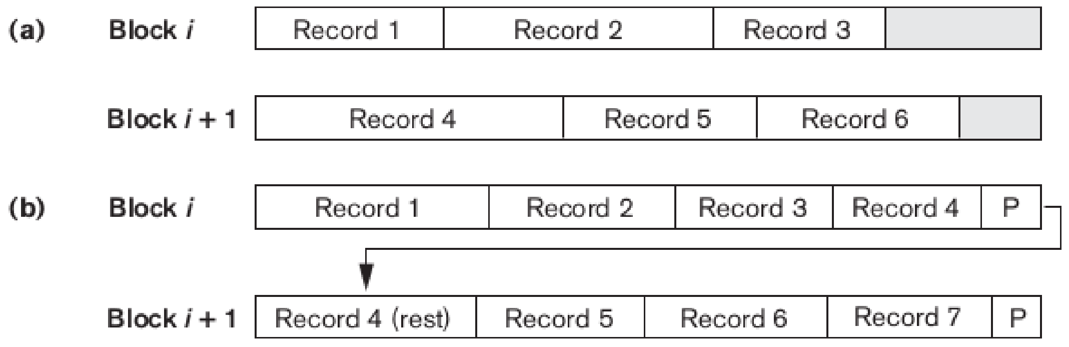
Record Blocking and Spanned Versus Un spanned Records:

The records of a file must be allocated to disk blocks because a block is the *unit of data transfer* between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block.

Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with B > R, we can fit *bfr* = floor(B/R) records per block, where the floor(x) *rounds down* the number x to an integer. The value *bfr* is called the blocking factor for the file. In general, R may not divide B exactly, so we have some unused space in each block equal to B - *(bfr* * R) bytes

To utilize this unused space, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk. This organization is called spanned, because records can span more than one block. Whenever a record is larger than a block, we *must* use a spanned organization. If records are not allowed to cross block boundaries, the organization is called un spanned. This is used with fixed-length records having B > R because it makes each record start at a known location in the block, simplifying record processing. For variable-length records, either a spanned or an un spanned organization can be used. If the average record is large, it is advantageous to use spanning to reduce the lost space in each block.

Types of record organization.
(a) Unspanned.
(b) Spanned.



For variable-length records using spanned organization, each block may store a different number of records. In this case, the blocking factor *bfr* represents the *average* number of records per block for the file. We can use *bfr* to calculate the number of blocks *b* needed for a file of *r* records:
b = *Ceil(r/bfr)* blocks where the Ceil(x) rounds the value x up to the next integer.

File Headers

A file header or file descriptor contains information about a file that is needed by the system programs that access the file records. The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions,

To search for a record on disk, one or more blocks are copied into main memory buffers. Programs then search for the desired record or records within the buffers, using the information in the file header. If the address of the block that contains the desired record is not known, the search programs must do a linear search through the file blocks.

File Operations:

Operations on database files can be broadly classified into two categories −

- **Update Operations**
- **Retrieval Operations**

Update operations change the data values by insertion, deletion, or update. Retrieval operations, on the other hand, do not alter the data but retrieve them after optional conditional filtering. In both types of operations, selection plays a significant role. Other than creation and deletion of a file, there could be several operations, which can be done on files.

- **Open** − A file can be opened in one of the two modes, **read mode** or **write mode**. In read mode, the operating system does not allow anyone to alter data. In other words, data is read only. Files opened in read mode can be shared among several entities. Write mode allows data modification. Files opened in write mode can be read but cannot be shared.
- **Locate** − Every file has a file pointer, which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using find (seek) operation, it can be moved forward or backward.
- **Read** − By default, when files are opened in read mode, the file pointer points to the beginning of the file. There are options where the user can tell the operating system where to locate the file pointer at the time of opening a file. The very next data to the file pointer is read.
- **Reset -** Sets the file pointer of an open file to the beginning of the file.
- **FindNext** - Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record.
- **Delete** - Deletes the current record and (eventually) updates the file on disk to reflect the deletion.
- **Modify** - Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.
- **Insert** - Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.
- **Write** − User can select to open a file in write mode, which enables them to edit its contents. It can be deletion, insertion, or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allows to do so.
- **Close** − This is the most important operation from the operating system's point of view. When a request to close a file is generated, the operating system
  - removes all the locks (if in shared mode),
  - saves the data (if altered) to the secondary storage media, and
  - releases all the buffers and file handlers associated with the file.

The organization of data inside a file plays a major role here. The process to locate the file pointer to a desired record inside a file various based on whether the records are arranged sequentially or clustered.

FILES OF UNORDERED RECORDS (HEAP FILES):

In this type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. This organization is often used with additional access paths, such as the secondary indexes.

Inserting: Inserting a new record is *very efficient:* the last disk block of the file is copied into a buffer; the new record is added; and the block is then rewritten back to disk. The address of the last file block is kept in the file header.

Searching: searching for a record using any search condition involves a linear search through the file block by block-an expensive procedure. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record. For a file of *b* blocks, this requires searching *(b/2)* blocks, on average. and b blocks in worst case.

Deleting: To delete a record, a program must first find its block, copy the block into a buffer, then delete the record from the buffer, and finally rewrite the block back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space.

Another technique used for record deletion is to have an extra byte or bit, called a deletion marker, stored with each record. A record is deleted by setting the deletion marker to a certain value. A different value of the marker indicates a valid (not deleted) record. Search programs consider only valid records in a block when conducting their search. Both of these deletion techniques require periodic reorganization of the file to reclaim the unused space of deleted records. During reorganization, the file blocks are accessed consecutively, and records are packed by removing deleted records.

Another possibility is to use the space of deleted records when inserting new records, although this requires extra bookkeeping to keep track of empty locations.

Reading: To read all records in order of the values of some field, we create a sorted copy of the file. Sorting is an expensive operation for a large disk file, and special techniques for external sorting are used.

FILES OF ORDERED RECORDS (SORTED FILES):

We can physically order the records of a file on disk based on the values of one of their fields-called the ordering field. This leads to an ordered or sequential files If the ordering field is also a key field of the file-a field guaranteed to have a unique value in each record-then the field is called the ordering key for the file. Figure shows an ordered file with NAME as the ordering key field.

Ordered records have some advantages over unordered files. They are:

1. Reading the records in order of the ordering key values becomes extremely efficient, because no sorting is required.
2. Finding the next record from the current one in order of the ordering key usually requires no additional block accesses, because the next record is in the same block as the current one
3. Using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches

A binary search for disk files can be done on the blocks rather than on the records. Suppose that the file has *b* blocks numbered 1, 2, ... , *b;* the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is K. Assuming that disk addresses of the file blocks are available in the file header. A binary search usually accesses $log_2(b)$ blocks, whether the record is found or not-an improvement over linear searches, where, on the average, *(b/2)* blocks are accessed when the record is found and *b* blocks are accessed when the record is not found.

Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
Block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
	⋮					
Block *n*−1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
Block *n*	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

Insertion and deletion: Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered. To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position. For a large file this can be very time consuming because, on the average, half the records of the file must be moved to make space for the new record. This means that half the file blocks must be read and rewritten after records are moved among them. For record deletion, the problem is less severe if deletion markers and periodic reorganization are used.

One option for making insertion more efficient is to keep some unused space in each block for new records. However, once this space is used up, the original problem resurfaces.

Modifying: Modifying a field value of a record depends on two factors: (1) the search condition to locate the record and (2) the field to be modified. If the search condition involves the ordering key field, we can locate the record using a binary search; otherwise we must do a linear search. A non ordering field can be modified by changing the record and rewriting it in the same physical location on disk-assuming fixed-length records. Modifying the ordering field means that the record can change its position in the file, which requires deletion of the old record followed by insertion of the modified record.
Reading: Reading the file records in order of the ordering field is quite efficient if we ignore the records in overflow, since the blocks can be read consecutively using double buffering. To include the records in overflow, we must merge them in their correct positions; in this case, we can first reorganize the file, and then read its blocks sequentially.

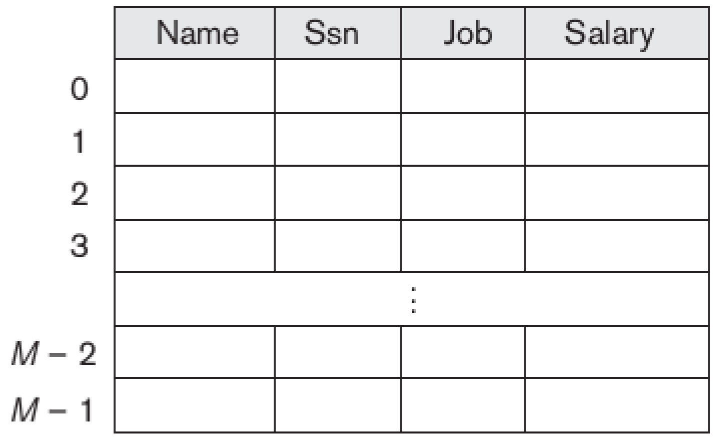
Ordered files are rarely used in database applications unless an additional access path, called a primary index, is used; this results in an **indexed sequential file**. This further improves the random access time on the ordering key field. If Ordering attribute is not key then the file is **Clustered file.**

HASHING:

Another type of primary file organization is based on hashing, which provides very fast access to records on certain search conditions. This organization is usually called a hash file. The search condition must be an equality condition on a single field, called the hash field of the file. In most cases, the hash field is also a key field of the file, in which case it is called the hash key. The idea behind hashing is to provide a function *h,* called a hash function or randomizing function, that is applied to the hash field value of a record and yields the *address* of the disk block in which the record is stored.

Internal Hashing:

For internal files, hashing is typically implemented as a hash table through the use of an array of records. For array index range is from 0 to M - 1 have M slots whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and M - 1.One common hash function is the *h(K)* = K mod M function.



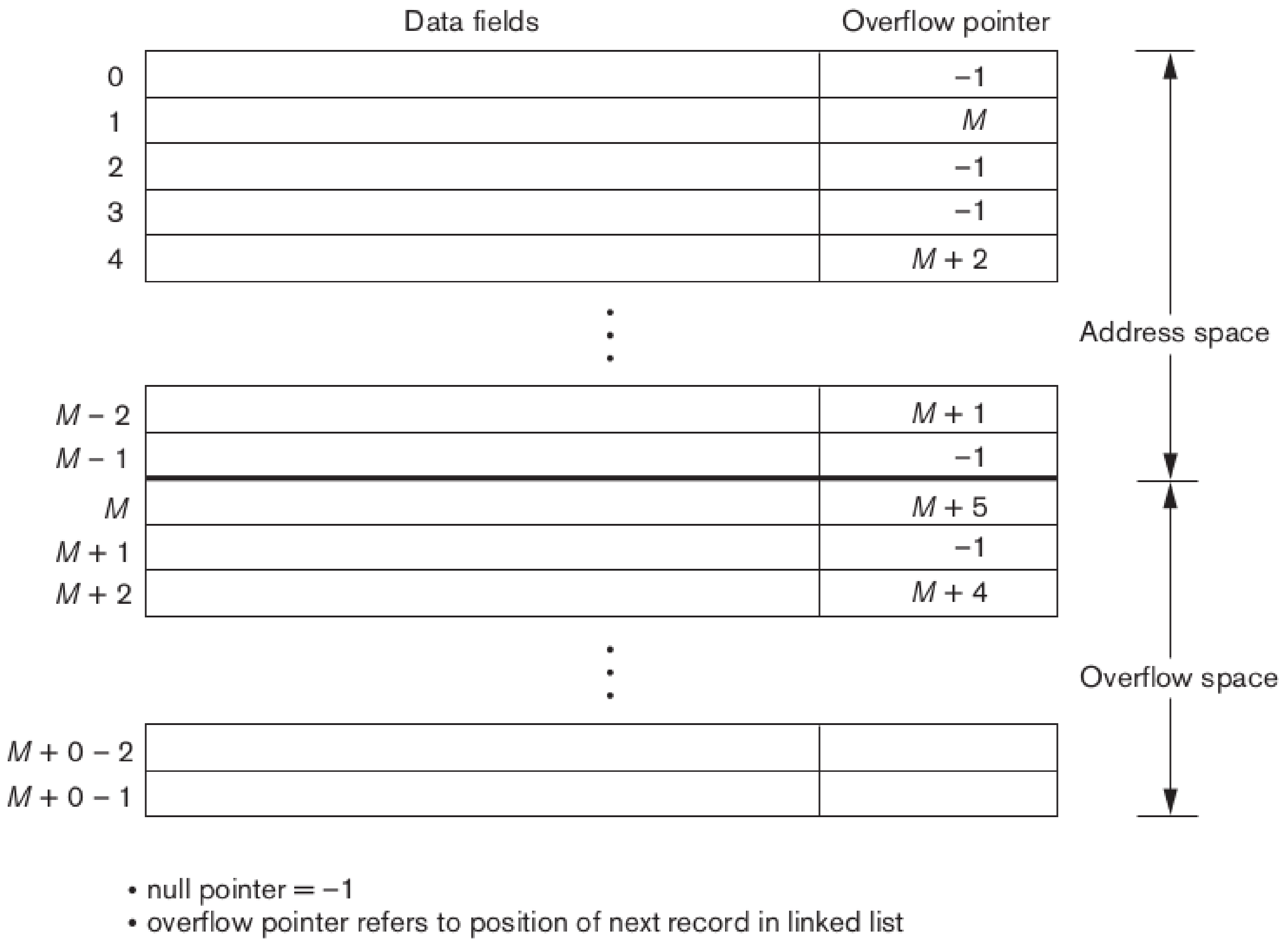
Array of *M* positions for use in internal hashing.

Non integer hash field values can be transformed into integers before the mod function is applied. For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation-for example, by multiplying those code values.

Other hashing functions can be used. One technique, called folding, involves applying an arithmetic function such as *addition* or a logical function such as *exclusive* or to different portions of the hash field value to calculate the hash address. Another technique involves picking some digits of the hash field value-for example, the third, fifth, and eighth digits-to form the hash address. to The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses, because the hash field space-the number of possible values a hash field can take-is usually much larger than the address space-the number of available addresses for records. The hashing function maps the hash field space to the address space.

A collision occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied. The process of finding another position is called collision resolution. There are numerous methods for collision resolution, including the following:

• *Open addressing:* Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
• *Chaining:* For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address is thus maintained.



Collision Resolution by Chaining

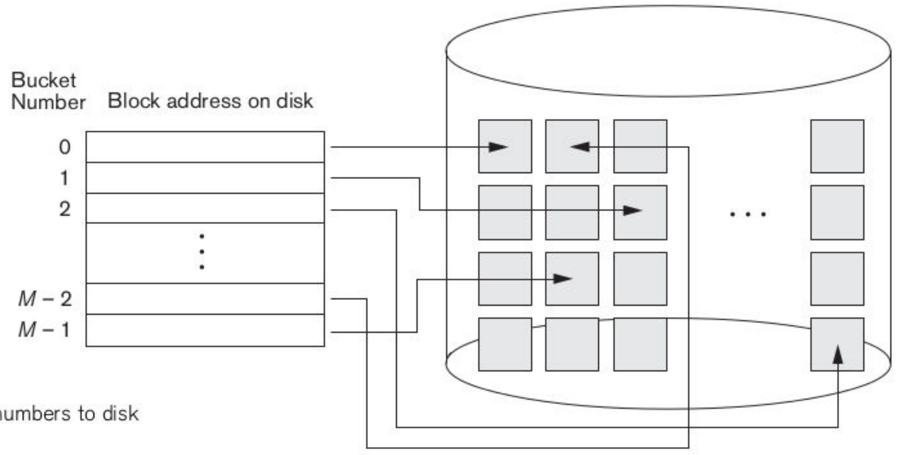
- *Multiple hashing:* The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. The goal of a good hashing function is to distribute the records uniformly over the address space so as to minimize collisions while not leaving many unused locations.

External Hashing for Disk Files:

Hashing for disk files is called external hashing. To suit the characteristics of disk storage, the target address space is made of buckets, each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous blocks. The hashing function maps a key into a relative bucket number, rather than assign an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address.

The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems. However, we must make provisions for the case where a bucket is filled to capacity and a new record being inserted hashes to that bucket. We can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket. The pointers in the linked list should be record pointers, which include both a block address and a relative record position within the block.

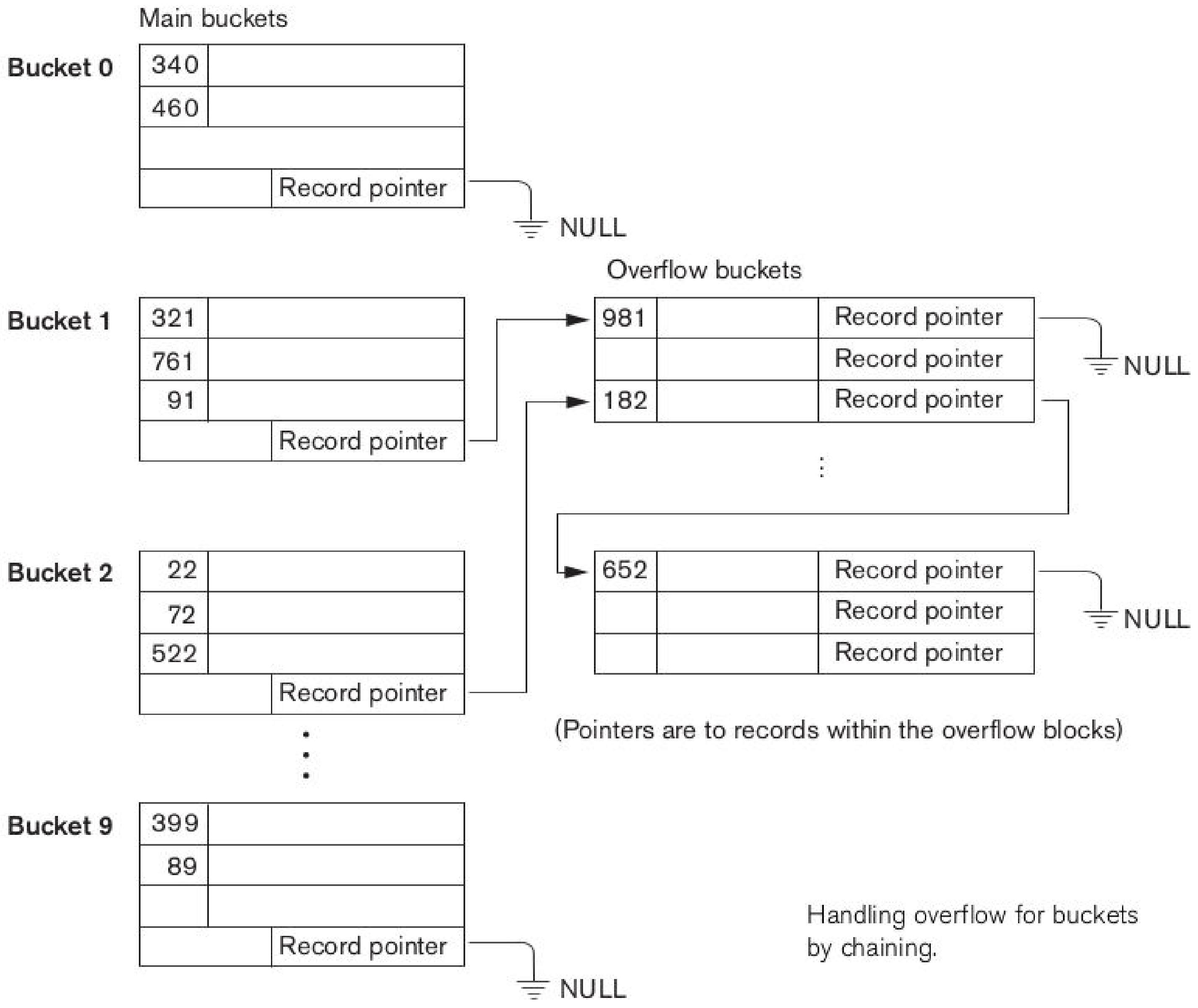


Matching bucket numbers to disk block addresses.

The hashing scheme described is called static hashing because a fixed number of buckets M is allocated. This can be a serious drawback for dynamic files. Suppose that we allocate M buckets for the address space and let m be the maximum number of records that can fit in one bucket then, at most (m * M) records will fit in the allocated space. If number of records turns out to be substantially fewer than (m * M), we are left with a lot of unused space. On the other hand, if the number of records increases to substantially more than (m * M), numerous collisions will result and retrieval will be slowed down because of the long lists of overflow records.

In either case, we may have to change the number of blocks M allocated and then use a new hashing function (based on the new value of M) to redistribute the records. These reorganizations can be quite time consuming for large files.

When using external hashing, searching for a record given a value of some field other than the hash field is as expensive as in the case of an unordered file. Record deletion can be implemented by removing the record from its bucket. If the bucket has an overflow chain, we can move one of the overflow records into the bucket to replace the deleted record. If the record to be deleted is already in overflow, we simply remove it from the linked list.



Handling overflow for buckets by chaining.

Hashing Techniques That Allow Dynamic File Expansion:

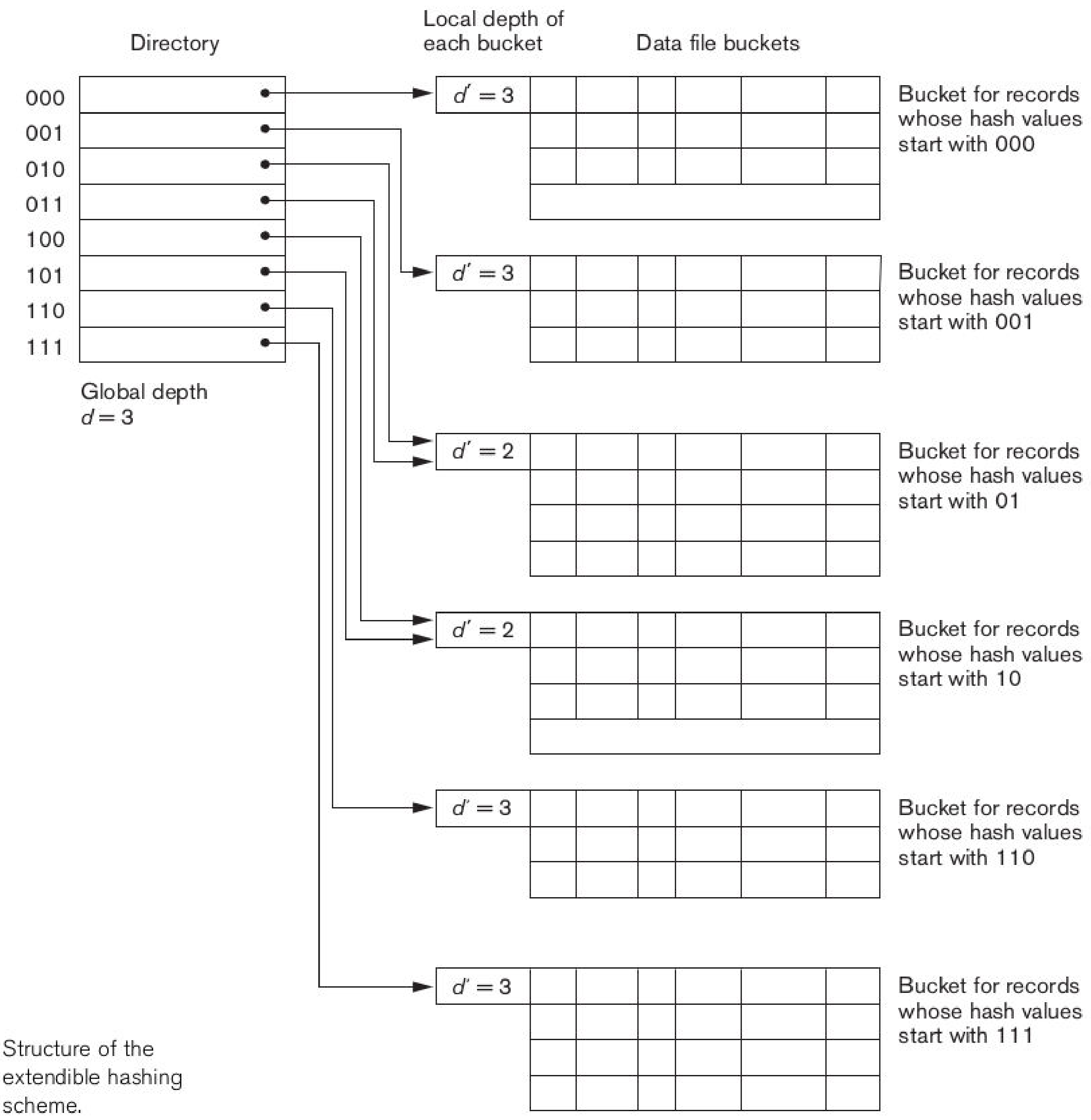
A major drawback of the *static* hashing scheme just discussed is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. The techniques used in dynamic hashing are:

1. Extendible Hashing
2. Dynamic Hashing
3. Linear Hashing

These hashing schemes take advantage of the fact that the result of applying a hashing function is a nonnegative integer and hence can be represented as a binary number. The access structure is built on the binary representation of the hashing function result, which is a string of bits. We call this the hash value of a record. Records are distributed among buckets based on the values of the *leading bits* in their hash values.

Extendible Hashing: In extendible hashing, a type of directory-an array of 2^d bucket addresses-is maintained, where d is called the global depth of the directory. The integer value corresponding to the first (high-order) d bits of a hash value is used as an index to the array to determine a directory entry, and the address in that entry determines the bucket in which the corresponding records are stored. However, there does not have to be a distinct bucket for each of the 2^d directory locations. Several directory locations with the same first d' bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket. A local depth d'-stored with each bucket-specifies the number of bits on which the bucket contents are based.

Bucket splitting: suppose that a new inserted record causes overflow in the bucket whose hash values start with 01-the third bucket, the records will be distributed between two buckets: the first contains all records whose hash values start with 010, and the second all those whose hash values start with 011. Now the two directory locations for 010 and 011 point to the two new distinct buckets. Before the split, they pointed to the same bucket. The local depth d' of the two new buckets is 3, which is one more than the local depth of the old bucket.


Directory
Local depth of each bucket
Data file buckets
000
001
010
011
100
101
110
111
Global depth d = 3
d' = 3
Bucket for records whose hash values start with 000
d' = 3
Bucket for records whose hash values start with 001
d' = 2
Bucket for records whose hash values start with 01
d' = 2
Bucket for records whose hash values start with 10
d' = 3
Bucket for records whose hash values start with 110
d' = 3
Bucket for records whose hash values start with 111
Structure of the extendible hashing scheme.

Overflow: If a bucket that overflows and is split used to have a local depth d' equal to the global depth d of the directory, then the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. For example, if the bucket for records whose hash values start with 111 overflows, the two new buckets need a directory with global depth d = 4, because the two buckets are now labeled 1110 and 1111, and hence their local depths are both 4. The directory size is hence doubled, and each of the other original locations in the directory is also split into two locations, both of which have the same pointer value as did the original location.

The main advantage of extendible hashing that makes it attractive is that the performance of the file does not degrade as the file grows, as opposed to static external hashing where collisions increase and the corresponding chaining causes additional accesses. In addition, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated dynamically as needed. Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets.

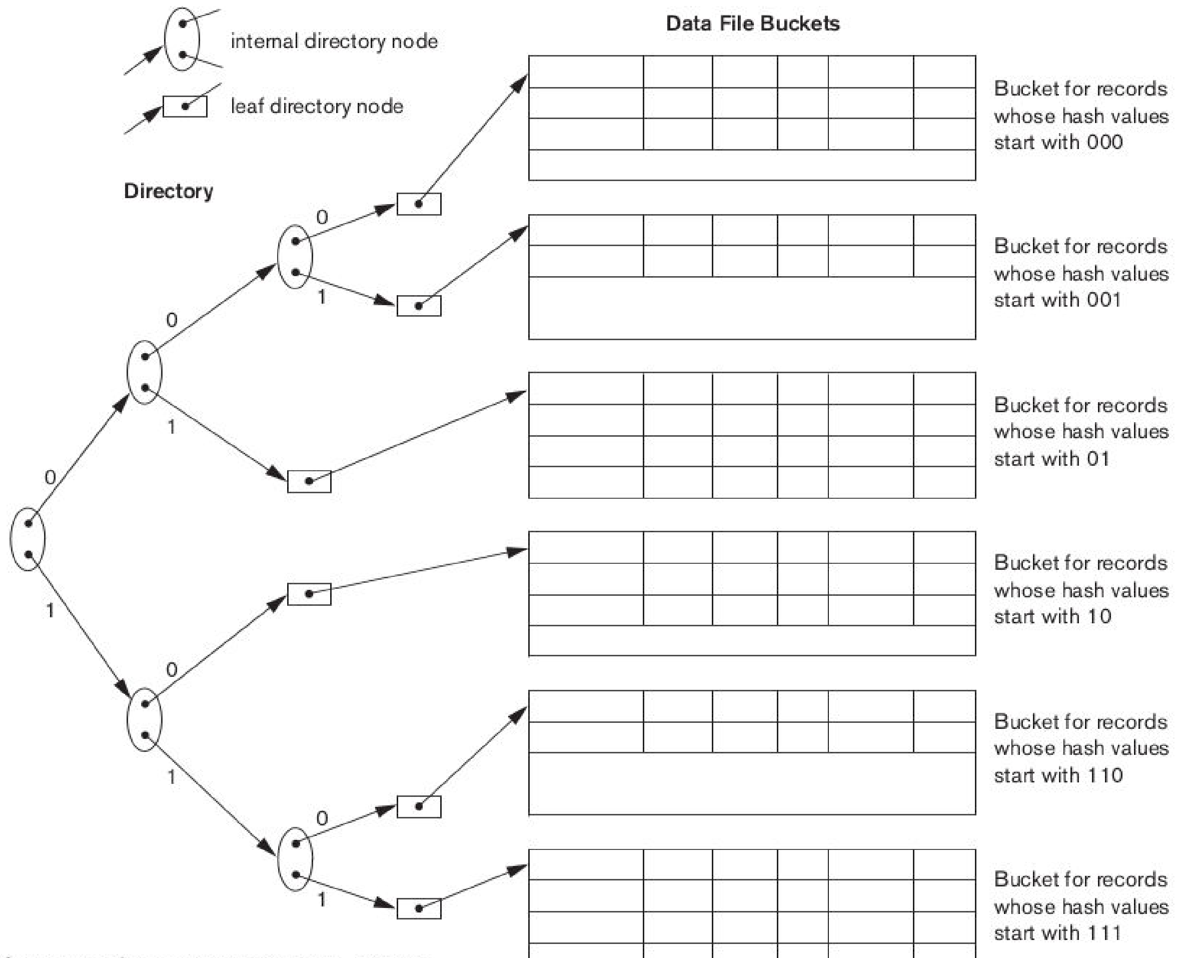
A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing. This performance penalty is considered minor.

Dynamic Hashing:

A precursor to extendible hashing was dynamic hashing, in which the addresses of the buckets were either the n- higher order bits or n-1 higher order bits, depending n the total number of keys belonging to the respective bucket. The eventual storage of records in buckets for dynamic hashing is somewhat similar to extendible hashing. The major difference is in the organization of the directory. whereas extendible hashing uses the notion of global depth (higher order d bits) for the flat directory and then combines adjacent collapsible buckets into a bucket of local depth d-1, dynamic hashing maintains a tree structured directory with two types of nodes.

- Internal nodes that have two pointers - the left pointer corresponds to the 0 bit (in the hashed address) and a right pointer corresponding to the 1 bit.
- Leaf nodes - these hold a pointer to the actual bucket with records.

An example of the dynamic hashing appears as shown in the figure.


internal directory node
leaf directory node
Data File Buckets
Directory
0
1
0
1
0
1
0
1
0
1
Bucket for records whose hash values start with 000
Bucket for records whose hash values start with 001
Bucket for records whose hash values start with 01
Bucket for records whose hash values start with 10
Bucket for records whose hash values start with 110
Bucket for records whose hash values start with 111

Structure of the dynamic hashing scheme.

Four buckets are shown ("000","001","110" and "111") with higher order 3 bit addresses (corresponding to the global depth of 3), and two buckets ("01" and "10") are shown with higher order 2 bit addresses (corresponding to the local depth of 2). The latter two are the result of collapsing the "010" and "011" into "01" and collapsing "100" and "101" into "10". Note that the directory nodes are used implicitly to determine local and global depths of buckets in dynamic hashing.

The search for a record given the hashed address involves traversing the directory tree which leads to the bucket holding that record.

Linear Hashing:

The idea behind linear hashing is to allow a hash file to expand and shrink its number of buckets dynamically *without* needing a directory. Suppose that the file starts with M buckets numbered 0, 1, ... , M - 1 and uses the mod hash function *h(K)* = K mod M; this hash function is called the initial hash function h_i .Overflow because of collisions is still needed and can be handled by maintaining individual overflow chains for each bucket. However, when a collision leads to an overflow record in *any* file bucket, the *first* bucket in the file-bucket 0-is split into two buckets: the original bucket 0 and a new bucket M at the end of the file. The records originally in bucket 0 are distributed between the two buckets based on a different hashing function $h_{i+1}(K)$ = K mod 2M. A key property of the two hash functions h_i and h_{i+1} is that any records that hashed to bucket 0 based on *hi* will hash to either bucket 0 or bucket M based on h_{i+1}; this is necessary for linear hashing to work.

As further collisions lead to overflow records, additional buckets are split in the *linear* order 1, 2, 3, ... If enough overflows occur, all the original file buckets 0, 1, ... ,M - 1 will have been split, so the file now has 2M instead of M buckets, and all buckets use the hash function h_{i+1} . Hence, the records in overflow are eventually redistributed into regular buckets, using the function h_{i+1} via a *delayed split* of their buckets. There is no directory; only a value n-which is initially set to 0 and is incremented by 1 whenever a split occurs-is needed to determine which buckets have been split. To retrieve a record with hash key value *K,* first apply the function *hi* to *K;* if $h_i(K) < n$, then apply the function h_{i+1} on *K* because the bucket is already split. Initially, n = 0, indicating that the function h_i applies to all buckets; n grows linearly as buckets are split.

Splitting can be controlled by monitoring the file load factor instead of by splitting whenever an overflow occurs. In general, the file load factor *1*can be defined as $l= r/(bfr * N)$, where r is the current number of file records, *bfr* is the maximum number of records that can fit in a bucket, and N is the current number of file buckets. Buckets that have been split can also be recombined if the load of the file falls below a certain threshold. Blocks are combined linearly, and N is decremented appropriately. The file load can be used to trigger both splits and combinations; in this manner the file load can be kept within a desired range.

Single Level Order Indexes:

An ordered index access structure is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book. We can search an index to find a list of addresses page numbers in this case-and use these addresses to locate a term in the textbook by *searching* the specified pages. Without index, searching for a phrase in a text book by shifting through whole textbook is like linear search.

For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an indexing field (or indexing attribute). The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are *ordered* so that we can do a binary search on the index. The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient.

There are several types of ordered indexes.

A **primary index** is specified on the *ordering key field* of an ordered file of records. An ordering key field is used to *physically order* the file records on disk, and every record has a *unique value* for that field. Primary index works for the records having a key field.

A **clustering index** is used if numerous records in the file can have the same value for the ordering field. A file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, *but not both.*

A **secondary index**, can be specified on any *non-ordering* field of a file. A file can have several secondary indexes in addition to its primary access method.

PRIMARY INDEX:

A primary index is an ordered file whose records are of fixed length with two fields. The first field is of the same data type as the ordering key field-called the primary key-of the data file, and the second field is a pointer to a disk block (a block address). There is one index entry (or index record) in the index file for each *block* in the data file. Each index entry has the value of the primary key field for the *first* record in a block and a pointer to that block as its two field values. We will refer to the two field values of index entry i as < K(i),P(i) >.

To create a primary index on the ordered file introduced , we use the NAME field as primary key, because that is the ordering key field of the file (assuming that each value of NAME is unique). Each entry in the index has a NAME value and a pointer. The first three index entries are as follows:

<K(l) = (Aaron,Ed), P(l) = address of block 1>
<K(2) = (Adams.john), P(2) = address of block 2>
<K(3) = (Alexander,Ed), P(3) = address of block 3>
Below figure illustrates this primary index.

The total number of entries in the index is the same as the *number of* disk *blocks* in the ordered data file. The first record in each block of the data file is called the anchor record of the block, or simply the block anchor.

Indexes can also be characterized as dense or sparse.
A dense index has an index entry for *every search key value* (and hence every record) in the data file.
A sparse (or non-dense) index on the other hand, has index entries for only some of the search values.

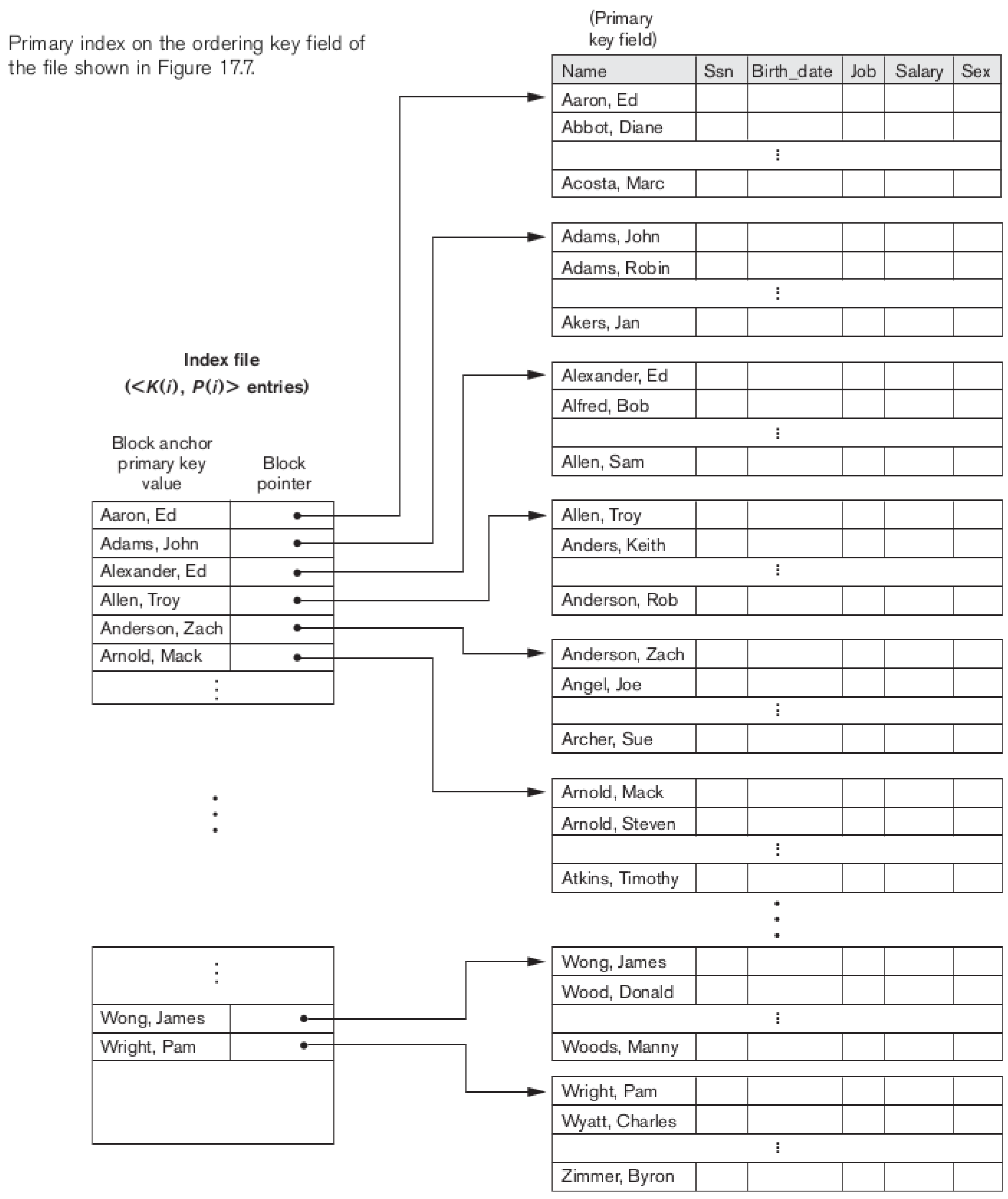
A primary index is hence a non-dense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value. The index file for a primary index needs substantially fewer blocks than does the data file, for two reasons.

1. There are *fewer* index *entries* than there are records in the data file.
2. Each index entry is typically *smaller in size* than a data record because it has only two fields; consequently, more index entries than data records can fit in one block. Hence requires fewer block accesses than searching on the data file.

The binary search for an ordered data file required log $_2$ b block accesses. where b is the no. of blocks in the data file. But if the primary index file contains b, blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record: a total of log $_2$ b_i +1 accesses.

A major problem with a primary index-as with any ordered file-is insertion and deletion of records. With a primary index, the problem is compounded because, if we attempt to insert a record in its correct position in the data file, we have to not only move records to make space for the new record but also change some index entries, since moving records will change the anchor records of some blocks.

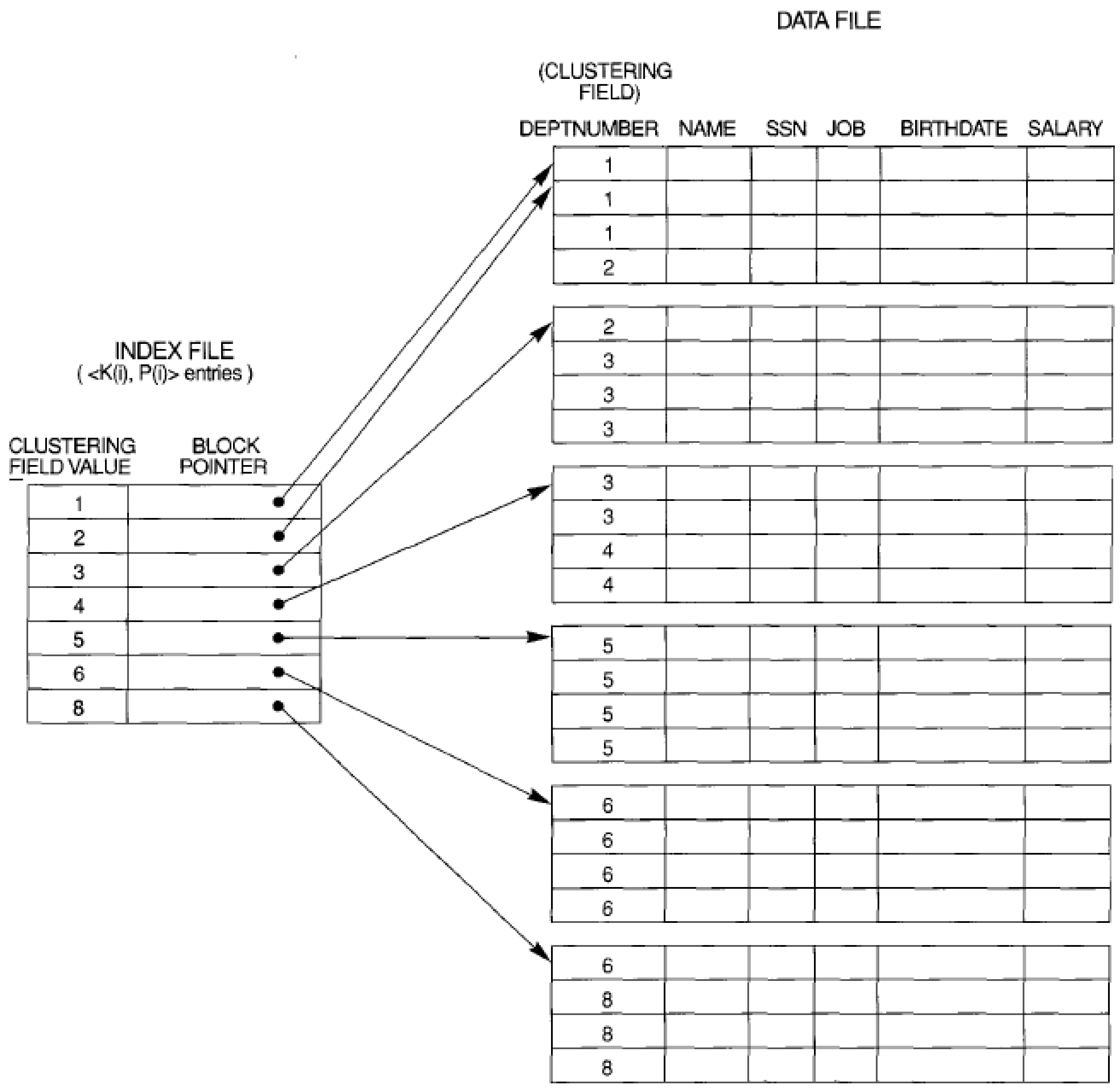
Solution for this is using an unordered overflow file or using a linked list of overflow records for each block in the data file.



CLUSTERING INDEX:

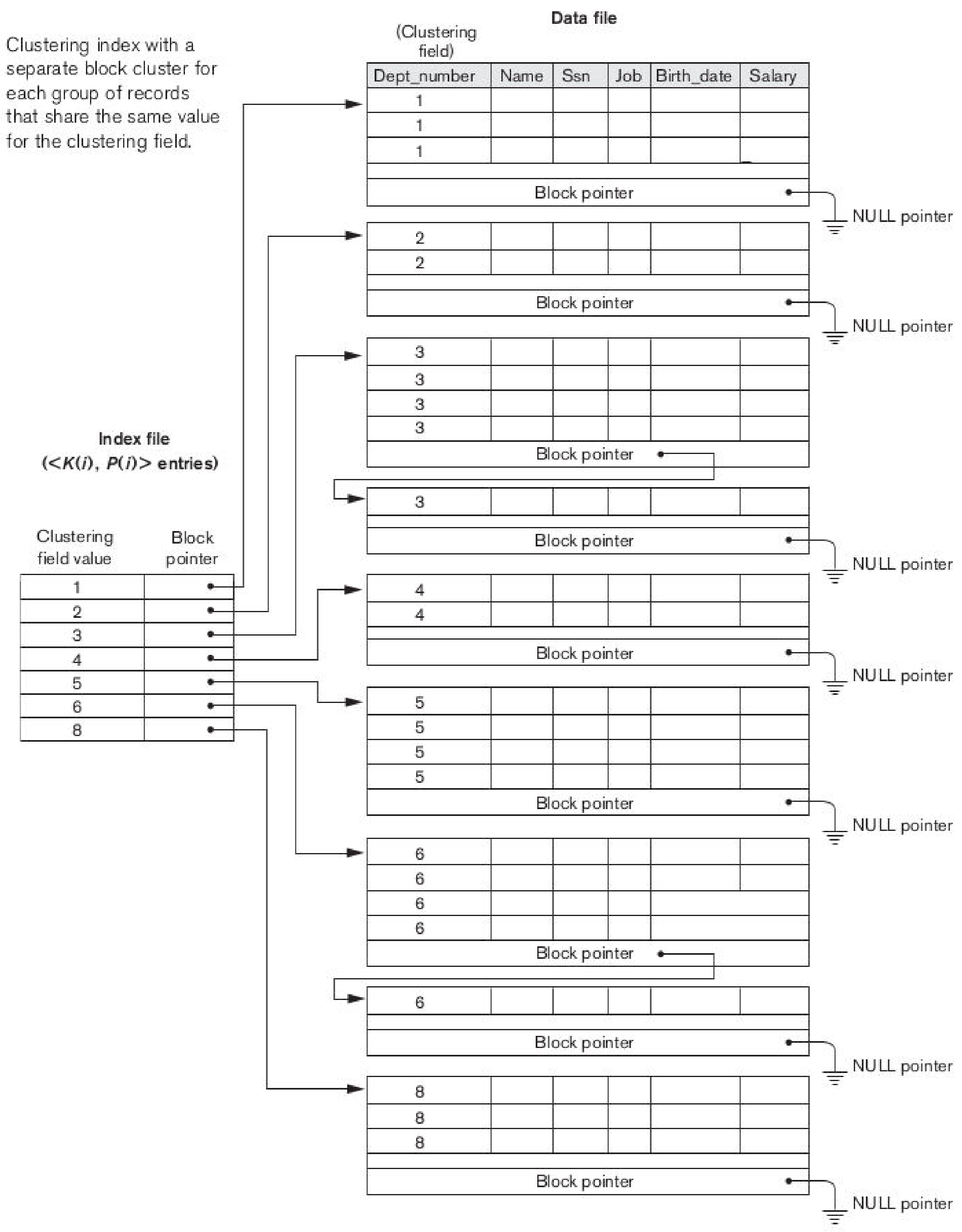
If records of a file are physically ordered on a non-key field-which *does not* have a distinct value for each record-that field is called the clustering field. We can create a different type of index, called a clustering index, to speed up retrieval of records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a *distinct value* for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, containing the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field.



Record insertion and deletion still cause problems, because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for *each value* of the clustering field; all records with that value are placed in the block (or block cluster). This makes insertion and deletion relatively straightforward.

A clustering index is another example of a *non-dense* index, because it has an entry for every *distinct value* of the indexing field which is a non-key by definition and hence has duplicate values rather than for every record in the file.



SECONDARY INDEXES:

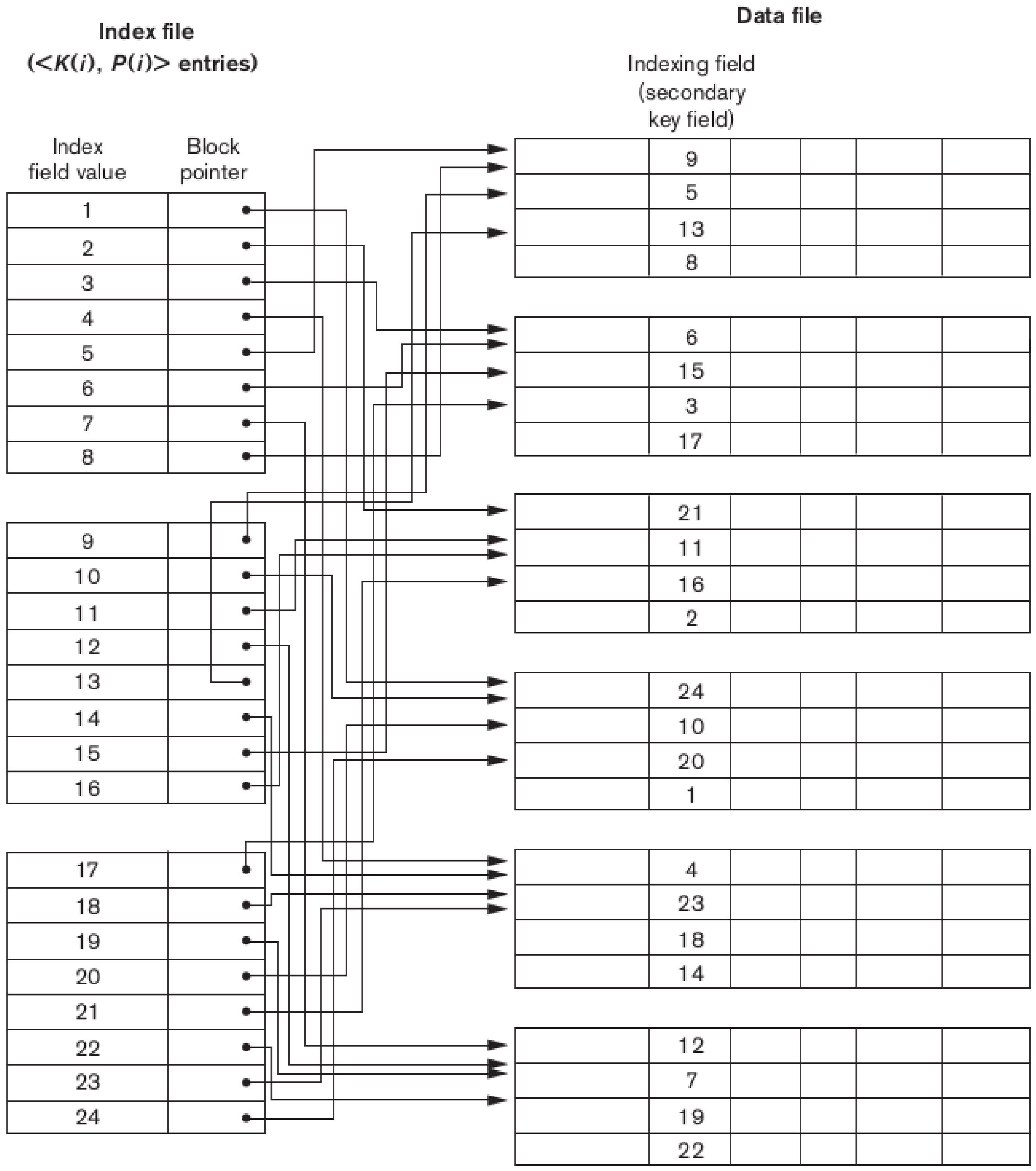
A secondary index provides a secondary means of accessing a file for which some primary access already exists. The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non key with duplicate values. The index is an ordered file with two fields. The first field is of the same data type as some *non ordering field* of the data file that is an indexing field. The second field is either a *block* pointer or a *record* pointer. There can be *many* secondary indexes (and hence, indexing fields) for the same file.

We first consider a secondary index access structure on a key field that has a *distinct value* for every record. Such a field is sometimes called a secondary key. In this case there is one index entry for *each record* in the data file, which contains the value of the secondary key *for* the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is dense.

We again refer to the two field values of index entry i as <K(i), P(i)>. The entries are ordered by value of K(i), so we can perform a binary search. Because the records of the data file are *not* physically ordered by values of the secondary key field, we *cannot* use block anchors. P(i) in the index entries are *block pointers,* not record pointers. Once the appropriate block is transferred to main memory, a search for the desired record within the block can be carried out.

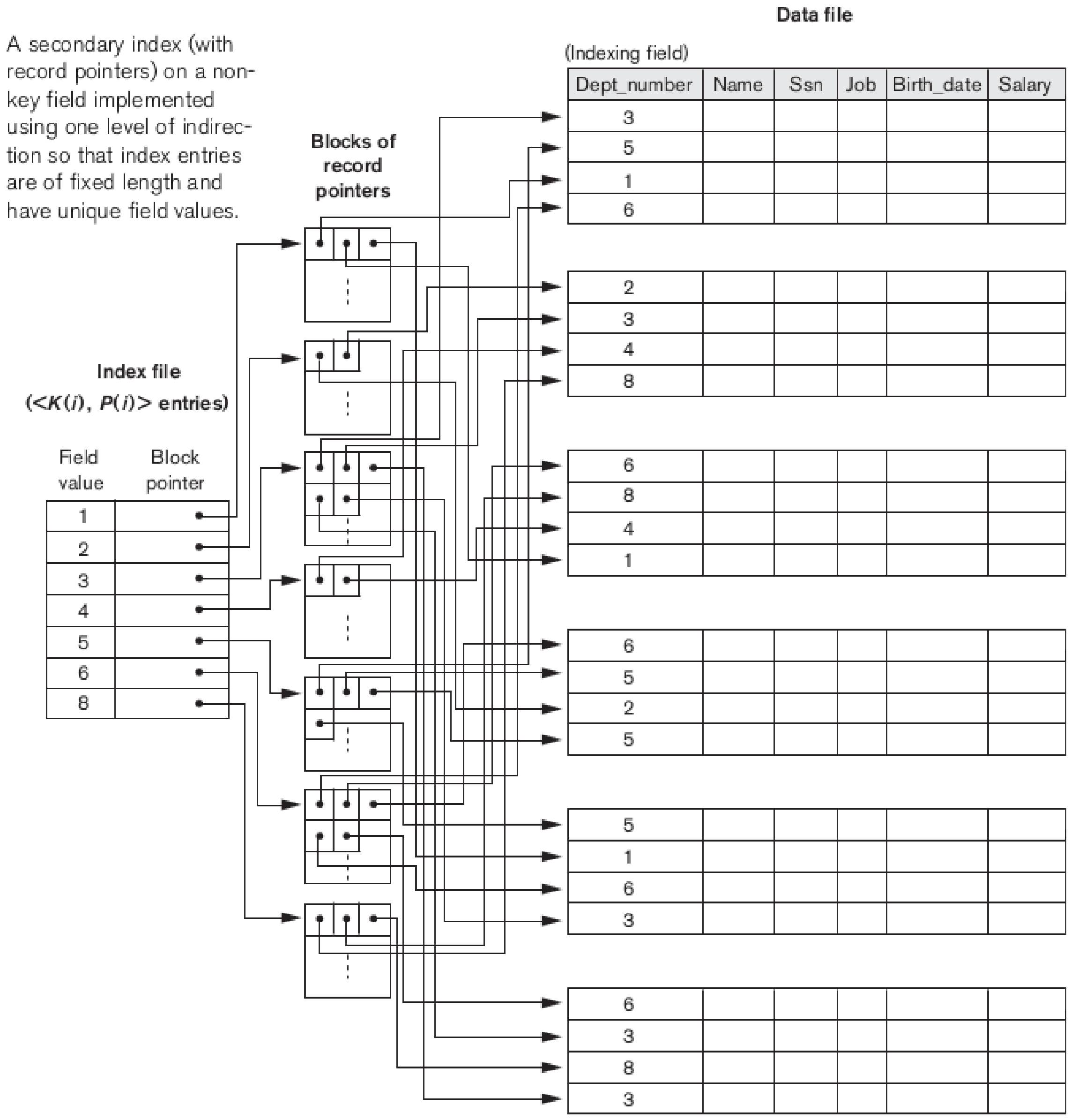
A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, the *improvement* in search time for an arbitrary record is much greater for a secondary index than for a primary index.

A dense secondary index (with block pointers) on a nonordering key field of a file.



We can also create a secondary index on a *non key field* of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:

1. Option 1 is to include several index entries with the same K(i) value-one for each record. This would be a dense index.
2. Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer. <K(i),<P(i,1),P(i,2).........P(i,k)>
3. Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each *index field value* but to create an extra level of indirection to handle the multiple pointers. In this non dense scheme, the pointer P(i) in index entry <K(i), P(i)> points to a *block of record pointers;* each record pointer in that block points to one of the data file records with value K(i) for the indexing field. If some value K(i) occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used.


Data file
A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.
(Indexing field)
Dept_number
Name
Ssn
Job
Birth_date
Salary
Blocks of record pointers
Index file (<K(i), P(i)> entries)
Field value
Block pointer
1
2
3
4
5
6
8
3
5
1
6
2
3
4
8
6
8
4
1
6
5
2
5
5
1
6
3
6
3
8
3

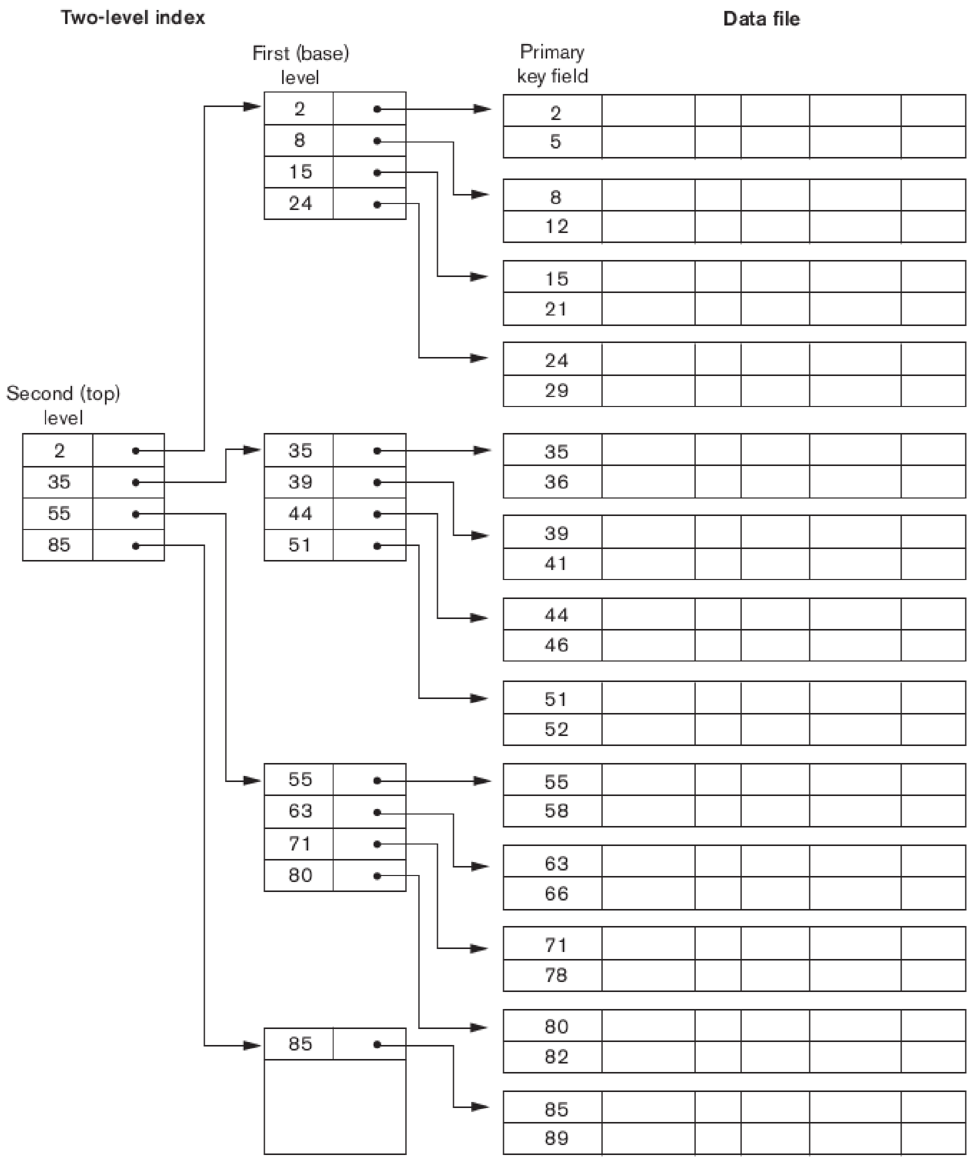
MULTI LEVEL INDEXES:

The idea behind a multilevel index is to reduce the part of the index that we continue to search by bfr_i, the blocking factor for the index, which is larger than 2. Hence, the search space is reduced much faster. The value bfr_i is called the fan-out of the multilevel index, and we will refer to it by the symbol f_0, Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses, which is a smaller number than for binary search if the fan-out is larger than 2.

A multilevel index considers the index file, which we will now refer to as the first (or base) level of a multilevel index, as an *ordered file* with a *distinct value* for each K(i). Hence we can create a primary index for the first level; this index to the first level is called the second level of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for *each block* of the first level. The blocking factor bfr_i for the second level-and for all subsequent levels-is the same as that for the first-level index, because all index entries are the same size; each has one field value and one block address. If the first level has r_1 entries, and the blocking factor-which is also the fan-out-for the index is $bfr_i = f_0$, then the first level needs $ceil((r_1/f_0))$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.

We can repeat this process for the second level. The third level, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is $r3 = ceil((r_2/f_0))$. Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block. We can repeat the preceding process until all the entries of some index level t fit in a single block. This block at the t^{th} level is called the top index level. Each level reduces the number of entries at the previous level by a factor of f_0-the index fan-out-so we can use the formula $1 <= (r_1/((f_0)^t))$ to calculate t. Hence, a multilevel index with r_1 first-level entries will have approximately t levels, where $t = CEIL((\log_{f_0}(r_1)))$.

A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



DYNAMIC MULTILEVEL INDEXES USING B-TREES AND B+-TREES

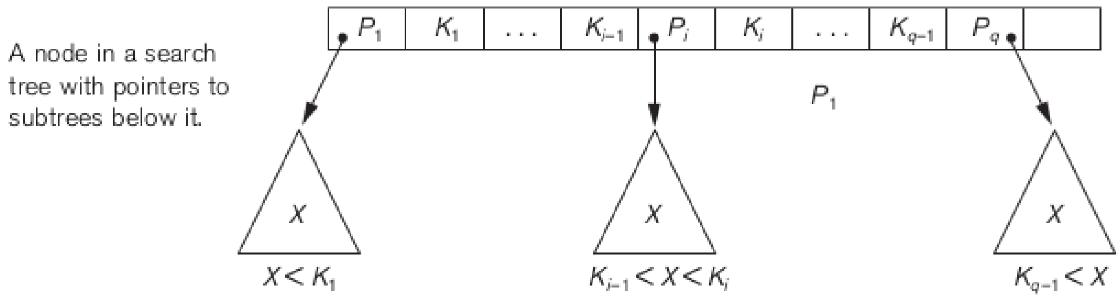
Search Trees and B-Trees:

A search tree is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields. A search tree is slightly different from a multilevel index. A search tree of order p is a tree such that each node contains *at most* p - 1 search values and p pointers in the order

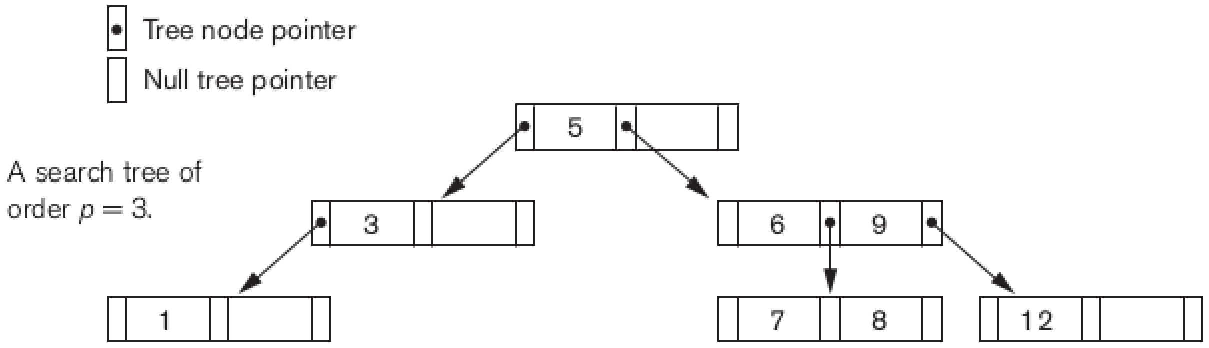
$<P_1, K_1, P_2, K_2 \ldots, P_{q-1}, K_{q-1}, P_q>$, where q <= p; each Pi

is a pointer to a child node (or a null pointer); and each K_i, is a search value from some ordered set of values. All search values are assumed to be unique. Two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < K_3 < \ldots < K_{q-1}$.
2. For all values X in the sub tree pointed at by P_i we have Ki-1 < X < K, for 1 < i < q; X < K_i, for i = 1; and Ki- 1 < X for i = q



We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the search field (which is the same as the index field if a multilevel index guides the search). Each key value in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.



Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is balanced, meaning that all of its leaf nodes are at the same level.

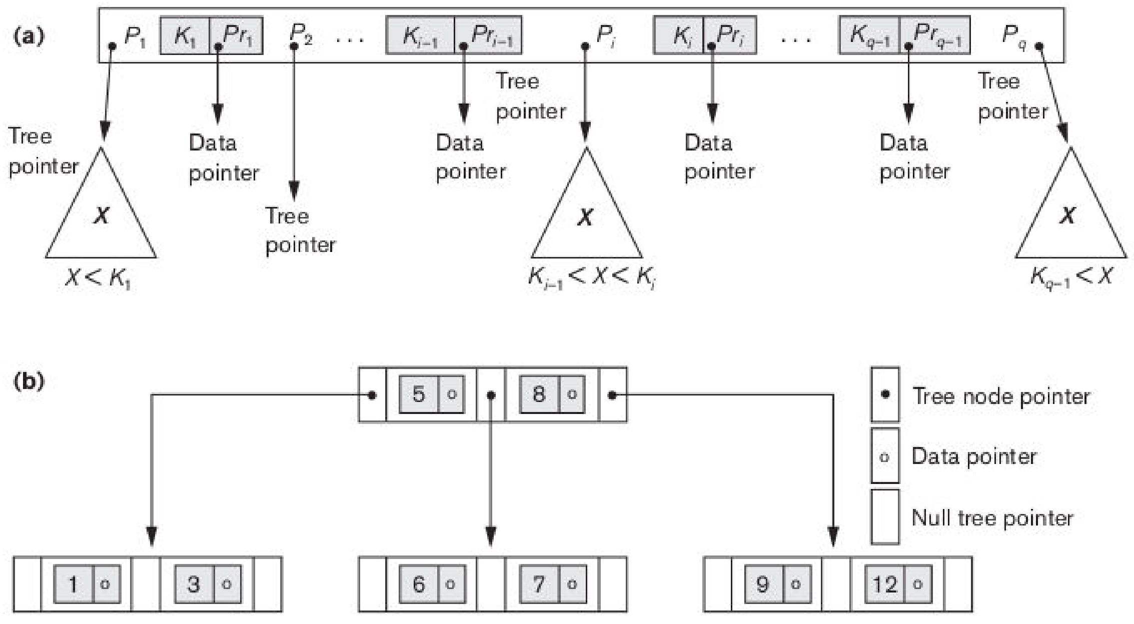
Keeping a search tree balanced is important because it guarantees that no nodes will beat very high levels and hence require many block accesses during a tree search. Keeping the tree balanced yields a uniform search speed regardless of the value of the search key. Another problem with search trees is that record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

B-Trees:

The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive.

More formally, a B-tree of order p, when used as an access structure on a *key field* to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree is of the form
   $<P_1,<K_1,Pr_1>,P_2,<K_2,Pr_2>..........,<K_{q-1},Pr_{q-1}>,P_q>$
   where q<=p. Each Pi is a **tree pointer** - a pointer to another node in the B-tree. Each Prj is a **data pointer** - a pointer to the record whose search key field value is equal to K, (or to the data file block containing that record).
2. Within each node, $K_1 < K_2 < ... < K_{q-1}$
3. For all search key field values X in the sub tree pointed at by P_i (the ith sub tree, see Figure), we have: $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for i=1 and $K_{i-1} < X$ for $i = q$.
4. Each node has at most p tree pointers.
5. Each node, except the root and leaf nodes, has at least Ceil(p/2) tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.
6. A node with q tree pointers, q <= p, has q - 1 search key field values (and hence has q - 1 data pointers).
7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P, are null.



B-tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

B+Trees:

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B+**-tree. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B+-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a non key search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

The leaf nodes of the W-tree are usually linked together to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B+-tree correspond to the other levels of a multilevel index. Some search field values from the leaf nodes are *repeated* in the internal nodes of the B+ tree to guide the search.

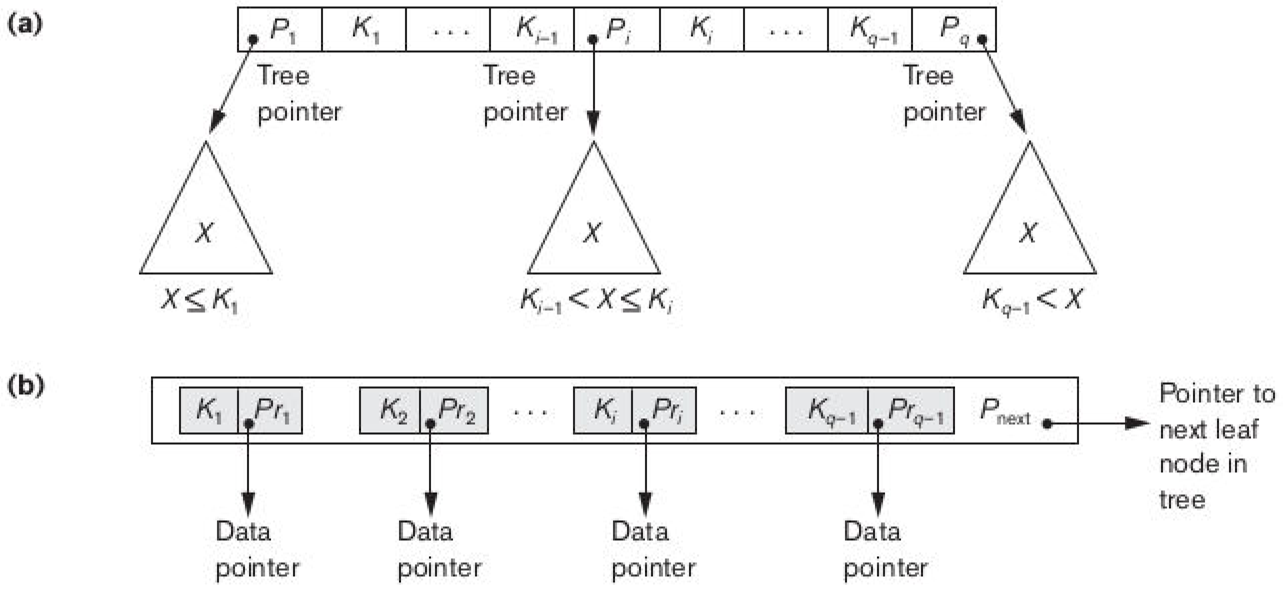
The structure of the *internal nodes* of a B+ tree of order p is as follows:

1. Each internal node is of the form $<P_1,K_1,P_2,K_2,......,P_{q-1},K_{q-1},P_q>$ where q <= P and each P_i is a tree pointer.
2. Within each internal node, $K_1 < K_2 < ...... < K_{q-1}$
3. For all search field values X in the sub tree pointed at by P_i, we have $K_{i-1} < X <= K_i$, for 1 < i < q; X <= K, for i = 1; and $K_{i-1} < X$ for i = q
4. Each internal node has at most p tree pointers.
5. Each internal node, except the root, has at least Ceil(p/2) tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with q pointers, q <= p, has q - 1 search field values.

The structure of the *leaf nodes* of a B^+-tree of order p is as follows:

1. Each leaf node is of the form $<<K_1,Pr_1>,<K_2,Pr_2>,....<K_{q-1},Pr_{q-1}>,P_{next}>$
   where q <= p, each Pr_i, is a data pointer, and P_{next} points to the next *leaf node* of the B+-tree.
2. Within each leaf node, $K_1 < K_2 < ... < K_{q-1}$, q <= p.
3. Each Pr, is a data pointer that points to the record whose search field value is K, or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K, if the search field is not a key).
4. Each leaf node has at least Ceil(p/2)) values.
5. All leaf nodes are at the same level.

The pointers in internal nodes are *tree pointers* to blocks that are tree nodes, whereas the pointers in leaf nodes are *data pointers* to the data file records or blocks-except for the P_{next} pointer, which is a tree pointer to the next leaf node. By starting at the leftmost leaf node, it is possible to traverse leaf nodes as a linked list, using the P_{next} pointers. This provides ordered access to the data records on the indexing field.



The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values.
(b) Leaf node of a B^+-tree with $q - 1$ search values and $q - 1$ data pointers.

Comparison between B-trees and B+ trees:

Because entries in the *internal nodes* of a B+-tree include search values and tree pointers without any data pointers, more entries can be packed into an internal node of a B+-tree than for a similar B-tree. Thus, for the same block (node) size, the order p will be larger for the B+-tree than for the B-tree. This can lead to fewer B+-tree levels, improving search time. Because the structures for internal and for leaf nodes of a B+-tree are different, the order p can be different. We will use p to denote the order for *internal nodes* and P_{leaf} to denote the order for *leaf nodes,* which we define as being the maximum number of data pointers in a leaf node.

Indexes on Multiple Keys:

If a certain combination of attributes is used very frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes. For example: List the employees whose deptno is 4 and age is 59. To find this there are alternative search strategies. They are:

1. If deptno has index but not age, then select the records having deptno 4 and then filtering based on age.
2. If age had index but not deptno, then select records having age 59 and filter the records based on deptno.
3. If both have indexes, then an intersection of these sets of records or pointers yields those records that satisfy both conditions, those records that satisfy both conditions, or the blocks in which records satisfying both conditions are located.

All of these alternatives eventually give the correct result. However, if the set of records that meet each condition (deptno = 4 or age = 59) individually are large, yet only a few records satisfy the combined condition, then none of the above is a very efficient technique for the given search request.

Ordered Index on Multiple Attributes:

In general, if an index is created on attributes $<A_1, A_2, \ldots , A_n>$, the search key values are tuples with n values: $<v_1, v_2, \ldots\ldots , v_n>$.

A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of department keys for department number 3 precede those for department 4. Thus <3, n> precedes <4, m> for any values of m and n. The ascending key order for keys with Dna = 4 would be <4, 18>, <4, 19>, <4,20>, and so on. Lexicographic ordering works similarly to ordering of character strings.

Partitioned Hashing:

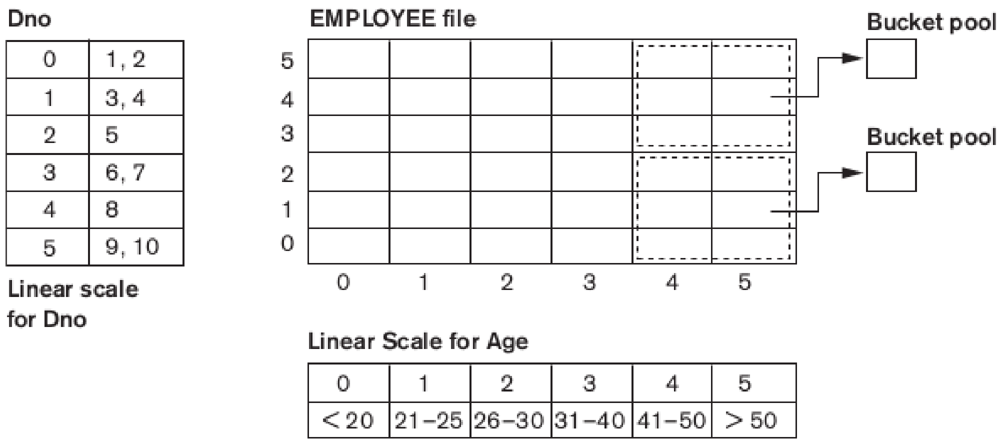
Partitioned hashing is an extension of static external hashing that allows access on multiple keys. It is suitable only for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of n components, the hash function is designed to produce a result with n separate hash addresses. The bucket address is a concatenation of these n addresses. It is then possible to search for the required composite search key by looking up the appropriate buckets that match the parts of the address in which we are interested.

An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

Grid Files:

Another alternative is to organize the EMPLOYEE file as a grid file. we can construct a grid array with one linear scale (or dimension) for each of the search attributes.

The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for DNO has DNO = 1, 2 combined as one value a on the scale, while DNO = 5 corresponds to the value 2 on that scale. Similarly, AGE is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age. The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored.



0	1, 2
1	3, 4
2	5
3	6, 7
4	8
5	9, 10

0	1	2	3	4	5
< 20	21–25	26–30	31–40	41–50	> 50

Example of a grid array on Dno and Age attributes.

Thus our request for DNO = 4 and AGE = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket. This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. Grid files perform well in terms of reduction in time for multiple key access. However, they represent a space overhead in terms of the grid array structure.

Moreover, with dynamic files, a frequent reorganization of the file adds to the maintenance cost.