**Unit – 7 Database Security and indexing**

**Authorization & Authentication:**

A security problem common to computer systems is that of preventing unauthorized persons from accessing the system itself, either to obtain information or to make malicious changes in a portion of the database. The security mechanism of a DBMS must include provisions for restricting access to the database system as a whole. This function, called **access control**, is handled by creating user accounts and passwords to control the login process by the DBMS.

Whenever a person or a group of persons needs to access a database system, the individual or group must first apply for a user account. The DBA will then create a new **account number** and **password** for the user if there is a legitimate need to access the database. The user must **log in** to the DBMS by entering the account number and password whenever database access is needed. The DBMS checks that the account number and password are valid; if they are, the user is permitted to use the DBMS and to access the database. Application programs can also be considered users and are required to log in to the database

It is straightforward to keep track of database users and their accounts and passwords by creating an encrypted table or file with two fields: AccountNumber and Password. This table can easily be maintained by the DBMS. Whenever a new account is created, a new record is inserted into the table. When an account is canceled, the corresponding record must be deleted from the table.

In SQL the following types of privileges can be granted on each individual relation *R*:

■ **SELECT (retrieval or read) privilege on *R.*** Gives the account retrieval privilege. In SQL this gives the account the privilege to use the SELECT statement to retrieve tuples from *R*.

■ **Modification privileges on *R.*** This gives the account the capability to modify the tuples of *R*. In SQL this includes three privileges: UPDATE, DELETE, and INSERT. These correspond to the three SQL commands (see Section 4.4) for modifying a table *R*. Additionally, both the INSERT and UPDATE privileges can specify that only certain attributes of *R* can be modified by the account.

■ **References privilege on *R.*** This gives the account the capability to *reference* (or refer to) a relation *R* when specifying integrity constraints. This privilege can also be restricted to specific attributes of *R*.

**An Example to Illustrate Granting and Revoking of Privileges**

Suppose that the DBA creates four accounts—A1, A2, A3, and A4—and wants only A1 to be able to create base relations. To do this, the DBA must issue the following GRANT command in SQL:

**GRANT** CREATETAB **TO** A1;

**GRANT** INSERT, DELETE **ON** EMPLOYEE, DEPARTMENT **TO** A2;

**REVOKE** SELECT **ON** EMPLOYEE **FROM** A3;

**Type of Security Breaches :**

The different types of Computer breaches can be broadly classified into three categories: **accidental** or **unintentional errors, intentional errors,** and **natural disasters.**

1. **Accidental or Unintentional Errors:** Accidents in the workplace can cause security breaches; it is often said that the cost of unintentional exceeds the cost of computer crime. Accidental security violations can be reduced by education and training, and by establishing and communicating security policies and procedures.

Sometimes hardware components accidentally fail- the memory, keyboard, Terminal emulators, LAN connection cards, network cabling etc, causing a security breach. Cables, modems, fax machines, and cellular phones are particularly prone to failure. Countless cables installed underground or on building roofs link and support corporate networks; since they are often forgotten during development of security plans. However, cables can be accidentally cut or destroyed during building repairs and maintenance, so measures should be taken to protect them.

1. **Intentional Errors :** intentional damage to systems is also common it is widely believed that is personnel are the greatest threat to system security, because they have intimate knowledge of the workings and the critical elements of a system. Intentional errors may take the form of illicit entry to a system, access to valuable proprietary corporate data, stealing passwords in order to snoop around, reading E-mail or destroying important files, regardless of the motives behind such acts, they are serious violations. They may go undetected for days weeks or even months, particularly if the violator is good at what be or she does let us look at some types of intentional security violations. Let looks some types of intentional security violations.

**(I) Cracking Passwords :** Cracking, or Decoding, passwords, which are the ’ heel of computer security, is one of the most common security violations, since hackers and crackers can easily guess passwords or capture them as they travel over the network. If a hacker or a cracker is hacked password of a system administrator or system controller, the he or she can access the vital important information.

**(II) Breaking into Computer hardware :** Another type of intentional security breach is breaking into and damaging computer hardware. Modems, fax machines, and cellular phones are particularly vulnerable to this type of damage.

Millions of modems all over the country give employees and other legal entities access to a company’s computer system. Unfortunately, modems are also highly vulnerable to security breaches.

1. **Software Viruses :** There are two primary sources of intentional software security breaches computer viruses and bad programs. The greatest threat to software is a computer virus, a program that causes a computer system to behave in unexpected and undesirable ways. Some signs of a virus include unexplainable loss of free memory, unusually log times for program loading or execution, changes in file or program size, point routines that stop working, computers “freezing up “ strange beeps or computer reboots in the middle of a process and corrupted files.
   1. **Natural Disasters:** The third type of security breach is caused by natural disasters, such as earthquakes, tornadoes, and floods. These can be so divesting that companies, and even entire cites take a long time to recover from them in 1992 alone insures in the U.S. handled records $18 billion in natural-disaster claims.

* **Types of Security**
  + Legal and ethical issues
  + Policy issues
  + System-related issues
  + The need to identify multiple security levels

■ Various legal and ethical issues regarding the right to access certain information—

for example, some information may be deemed to be private and cannot

be accessed legally by unauthorized organizations or persons. In the

United States, there are numerous laws governing privacy of information.

■ Policy issues at the governmental, institutional, or corporate level as to what

kinds of information should not be made publicly available—for example,

credit ratings and personal medical records.

■ System-related issues such as the *system levels* at which various security functions

should be enforced—for example, whether a security function should

be handled at the physical hardware level, the operating system level, or the

DBMS level.

■ The need in some organizations to identify multiple *security levels* and to

categorize the data and users based on these classifications—for example,

top secret, secret, confidential, and unclassified. The security policy of the

organization with respect to permitting access to various classifications of

data must be enforced.

**Level of Database Security**

In a multiuser database system, the DBMS must provide techniques to enable certain users or user groups to access selected portions of a database without gaining access to the rest of the database. This is particularly important when a large integrated database is to be used by many different users within the same organization. For example, sensitive information such as employee salaries or performance reviews should be kept confidential from most of the database system’s users. A

DBMS typically includes a **database security and authorization subsystem** that is responsible for ensuring the security of portions of a database against unauthorized access. It is now customary to refer to two types of database security mechanisms:

■ **Discretionary security mechanisms.** These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).

■ **Mandatory security mechanisms.** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization. For example, a typical security policy is to permit users at a certain classification (or clearance) level to see only the data items classified at the user’s own (or lower) classification level. An extension of this is *role-based security,* which enforces policies and privileges based on the concept of organizational roles.

**Types of Discretionary Privileges**

In SQL2 and later versions,3 the concept of an **authorization identifier** is used to refer, roughly speaking, to a user account (or group of user accounts). For simplicity, we will use the words *user* or *account* interchangeably in place of *authorization identifier.* The DBMS must provide selective access to each relation in the database based on specific accounts. Operations may also be controlled; thus, having an account does not necessarily entitle the account holder to all the functionality provided by the DBMS. Informally, there are two levels for assigning privileges to use

the database system:

■ **The account level.** At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database. The privileges at the **account level** apply to the capabilities provided to the account itself and can include the CREATE SCHEMA or CREATE TABLE privilege, to create a schema or base relation; the CREATE VIEW privilege; the ALTER privilege, to apply schema changes such as adding or removing attributes from relations; the DROP

privilege, to delete relations or views.

■ **The relation (or table) level.** At this level, the DBA can control the privilege to access each individual relation or view in the database. Privileges at the relation level specify for each user the individual relations on which each type of command can be applied. Some privileges also refer to individual columns (attributes) of relations. SQL2 commands provide privileges at the *relation and attribute level only.* Although this is quite general, it makes it difficult to create accounts with limited privileges. In SQL the following types of privileges can be granted on each individual relation *R*:

■ **SELECT (retrieval or read) privilege on *R.*** Gives the account retrieval privilege. In SQL this gives the account the privilege to use the SELECT statement to retrieve tuples from *R*.

■ **Modification privileges on *R.*** This gives the account the capability to modify the tuples of *R*. In SQL this includes three privileges: UPDATE, DELETE, and INSERT.

■ **References privilege on *R.*** This gives the account the capability to *reference* (or refer to) a relation *R* when specifying integrity constraints. This privilege can also be restricted to specific attributes of *R*.

**Revoking of Privileges**

In some cases it is desirable to grant a privilege to a user temporarily. For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for **revoking** privileges is needed. In SQL a REVOKE command is included for the purpose of canceling privileges.

**An Example to Illustrate Granting and Revoking of Privileges**

Suppose that the DBA creates four accounts—A1, A2, A3, and A4—and wants only A1 to be able to create base relations. To do this, the DBA must issue the following GRANT command in SQL:

**GRANT** CREATETAB **TO** A1;

The CREATETAB (create table) privilege gives account A1 the capability to create new database tables (base relations) and is hence an *account privilege.*

Next, suppose that account A1 wants to grant to account A2 the privilege to insert and delete tuples in both of these relations. However, A1 does not want A2 to be able to propagate these privileges to additional accounts. A1 can issue the following command:

**GRANT** INSERT, DELETE **ON** EMPLOYEE, DEPARTMENT **TO** A2;

Next, suppose that A1 wants to allow account A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts. A1 can issue the following command:

**GRANT** SELECT **ON** EMPLOYEE, DEPARTMENT **TO** A3 **WITH GRANT OPTION**;

Now suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 then can issue this command:

**REVOKE** SELECT **ON** EMPLOYEE **FROM** A3;

**Mandatory Access Control and Role-Based Access Control for Multilevel Security**

The discretionary access control technique of granting and revoking privileges on relations has traditionally been the main security mechanism for relational database systems. This is an all-or-nothing method: A user either has or does not have a certain privilege. In many applications, an *additional security policy* is needed that classifies data and users based on security classes. This approach, known as **mandatory access control (MAC)**, would typically be *combined* with the discretionary access control mechanisms. It is important to note that most commercial DBMSs currently provide mechanisms only for discretionary access control. However, the need for multilevel security exists in government, military, and intelligence applications, as well as in many industrial and corporate applications. Some DBMS vendors—for example, Oracle—have released special versionsof their RDBMSs that incorporate mandatory access control for government use.

Typical **security classes** are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest. Other more complex security classification schemes exist, in which the security classes are organized in a lattice. For simplicity, we will use the system with four security classification levels, where TS ≥S ≥C ≥U, to illustrate our discussion. The commonly used model for multilevel security, known as the *Bell-LaPadula model*, classifies each **subject** (user, account, program) and **object** (relation, tuple, column, view, operation) into one of

the security classifications TS, S, C, or U.We will refer to the **clearance** (classification) of a subject *S* as **class(*S*)** and to the **classification** of an object *O* as **class(*O*)**. Two restrictions are enforced on data access based on the subject/object classifications:

**1.** A subject *S* is not allowed read access to an object *O* unless class(*S*)>= class(*O*). This is known as the **simple security property**.

**2.** A subject *S* is not allowed to write an object *O* unless class(*S*) <= class(*O*). This is known as the **star property** (or \*-property).

**Role-Based Access Control**

Role-based access control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprise-wide systems. Its basic notion is that privileges and other permissions are associated with organizational **roles**, rather than individual users. Individual users are then assigned to appropriate roles. Roles can be created using the CREATE ROLE and DESTROY ROLE commands. The GRANT and REVOKE commands can then be used to assign and revoke privileges from roles, as well as for individual users when needed. For example, a company may have roles such as sales account manager, purchasing agent, mailroom clerk, department manager, and so on. Multiple individuals can be assigned to each role. Security privileges that are common to a role are granted to the role name, and any individual assigned to this role would automatically have those privileges granted.

RBAC can be used with traditional discretionary and mandatory access controls; it ensures that only authorized users in their specified roles are given access to certain data or resources. Users create sessions during which they may activate a subset of roles to which they belong. Each session can be assigned to several roles, but it maps to one user or a single subject only. Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles.

The **role hierarchy** in RBAC is a natural way to organize roles to reflect the organization’s lines of authority and responsibility. By convention, junior roles at the bottom are connected to progressively senior roles as one moves up the hierarchy. The hierarchic diagrams are partial orders, so they are reflexive, transitive, and anti symmetric. In other words, if a user has one role, the user automatically has roles lower in the hierarchy. Defining a role hierarchy involves choosing the type of hierarchy and the roles, and then implementing the hierarchy by granting roles to other roles. Role hierarchy can be implemented in the following manner:

**GRANT ROLE** full\_time **TO** employee\_type1

**GRANT ROLE** intern **TO** employee\_type2

The above are examples of granting the roles *full\_time* and *intern* to two types of employees.

**Concept of indexing :**

It is used to determine an efficient file organization for each base relation. For example, if we want to retrieve student records in alphabetical order of name, sorting the file by student name is a good file organization. However, if we want to retrieve all students whose marks is in a certain range, a file ordered by student name would not be a good file organization. Some file organizations are efficient for bulk loading data into the database but inefficient for retrieve and other activities.

The objective of this selection is to choose an optimal file organization for each relation.

Types of File Organization

In order to make effective selection of file organizations and indexes, here we present the details different types of file Organization. These are:

• Hash File Organization

• Indexed Sequential Access Methods (ISAM) File Organization

• B+- tree File Organization

• Cluster File Organization

**Difference between data file and index file:**

|  |  |
| --- | --- |
| **Data files** | **Index File** |
| A **Data file** is a [computer file](https://en.wikipedia.org/wiki/Computer_file) which stores data to be used by a computer [application](https://en.wikipedia.org/wiki/Application_software) or [system](https://en.wikipedia.org/wiki/System_software). | An **indexed file** is a [computer file](https://en.wikipedia.org/wiki/Computer_file) with an [index](https://en.wikipedia.org/wiki/Lookup_table) that stores the records based on its file [key](https://en.wikipedia.org/wiki/Unique_key). |
| Data is stored without any order. | Data is stored on any order it can be ascending order or descending order. |
| Data can not be ordered on any field. | Records can be order on multiple fields |
| Because of data is not order so searching takes more time. | Because data is already ordered so searching becomes easy and fast. |
| Here, data structure is in word format like as paragraph. | Here, data is stored in tabular format. |
| Data files are two typed open file and close file . | Index can be clustered and nonclustered. |
| **Example:** text file, image file | **Mdb, accdb files etc.** |

Type of indexes :

* + 1. Primary index
    2. Secondary index

1. 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 types as the ordering key field of the data file, and the second field is a pointer to a disk block - a block address. The ordering key field is called the **primary key of the data** file 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 other block as its two field values. We will refer to the two field values of index entry i as K (i), P (i).

|  |
| --- |
|  |

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 **nondense**) **index**, on the other hand, has index entries for only some of the search values. A sparse index has fewer entries than the number of records in the file. Thus, **a primary index is a nondense (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 (or every record).

The index file for a primary index occupies a much smaller space than does the data file, for two reasons. First, there are *fewer index entries* than there are records in the data file. Second, 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.

**Clustering Indexes**

If file records 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** and the data file is called a **clustered file.**We can create a different type of index, called a **clustering index**, to speed up retrieval of all the 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 disk block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, and it contains the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. Figure 18.2 shows an example. Notice that 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 nondense index because it has an entry for every distinct value of the indexing field, which is a nonkey by definition and hence has duplicate values rather than a unique value for every record in the file.

|  |
| --- |
|  |

**Secondary Indexes**

A secondary index provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed. The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a nonkey field with duplicate values. The index is again 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. Many secondary

indexes (and hence, indexing fields) can be created for the same file—each represents an additional means of accessing that file based on some specific field.

First we consider a secondary index access structure on a key (unique) field that has a distinct value for every record. Such a field is sometimes called a secondary key; in the relational model, this would correspond to any UNIQUE key attribute or to the primary key attribute of a table. In this case there is one index entry for each record in the data file, which contains the value of the field 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**.

|  |
| --- |
|  |

**Multilevel Indexes**

The indexing schemes we have described thus far involve an ordered index file. A

binary search is applied to the index to locate pointers to a disk block or to a record

(or records) in the file having a specific index field value. A binary search requires

approximately (log2bi) block accesses for an index with bi blocks because each step

of the algorithm reduces the part of the index file that we continue to search by a

factor of 2. This is why we take the log function to the base 2. The idea behind a

multilevel index is to reduce the part of the index that we continue to search by bfri,

the blocking factor for the index, which is larger than 2. Hence, the search space is

reduced much faster. The value bfri is called the **fan-out** of the multilevel index, and

we will refer to it by the symbol fo. Whereas we divide the record search space into

two halves at each step during a binary search, we divide it n-ways (where n = the

fan-out) at each search step using the multilevel index. Searching a multilevel index

requires approximately (logfobi) block accesses, which is a substantially smaller

number than for a binary search if the fan-out is larger than 2. In most cases, the

fan-out is much larger than 2.

|  |
| --- |
|  |

In a full indexing scheme, the address of every record is maintained in the index. For a small file, this index would be small and can be processed very efficiently in main memory.

For a large file, the index's size would pose problems. It is possible to create a hierarchy of indexes with the lowest level index pointing to the records, while the higher-level indexes point to the indexes below them (**figure 18.5**). The higher-level indices are small and can be moved to main memory, allowing the search to be localized to one of the larger lower level indices.

**Dense and Sparse Indices :**

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 **nondense**) **index**, on the other hand, has index entries for only some of the

search values. A sparse index has fewer entries than the number of records in the

file. Thus, a primary index is a nondense (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 (or every record)..

The index file for a primary index occupies a much smaller space than does the data

file, for two reasons. First, there are *fewer index entries* than there are records in the

data file. Second, 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.

**Indexing Evaluation :**

A binary search on the index file requires fewer block accesses than a binary search on the data file. The binary search for an ordered data file required log2*b* block accesses. But if the primary index file contains only *bi* 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 **log2*bi* + 1** accesses.

A record whose primary key value is *K* lies in the block whose address is *P*(*i*), where

*K*(*i*) ≤ *K* < *K*(*i* + 1). The *i*th block in the data file contains all such records because

of the physical ordering of the file records on the primary key field. To retrieve a

record, given the value *K* of its primary key field, we do a binary search on the index

file to find the appropriate index entry *i*, and then retrieve the data file block whose

address is *P*(*i*). Example 1 illustrates the saving in block accesses that is attainable

when a primary index is used to search for a record.

**Example 1.** Suppose that we have an ordered file with *r* = 30,000 records stored on

a disk with block size *B* = 1024 bytes. File records are of fixed size and are

unspanned, with record length *R* = 100 bytes. The blocking factor for the file would

be *bfr* = ⎣(*B*/*R*)⎦ = ⎣(1024/100)⎦ = 10 records per block. The number of blocks

needed for the file is *b* = ⎡(*r*/*bfr*)⎤ = ⎡(30000/10)⎤ = 3000 blocks. A binary search on the data file would need approximately ⎡log2*b*⎤= ⎡(log23000)⎤ = 12 block accesses.

Now suppose that the ordering key field of the file is *V* = 9 bytes long, a block pointer is *P* = 6 bytes long, and we have constructed a primary index for the file. The size of each index entry is *Ri* = (9 + 6) = 15 bytes, so the blocking factor for the index is *bfri* = ⎣(B/R*i*)⎦ = ⎣(1024/15)⎦ = 68 entries per block. The total number of index entries *ri* is equal to the number of blocks in the data file, which is 3000. The number of index blocks is hence *bi* = ⎡(*ri*/*bfri*)⎤ = ⎡(3000/68)⎤ = 45 blocks. To perform a binary search on the index file would need ⎡(log2*bi*)⎤ = ⎡(log245)⎤ = 6 block accesses. To search for a record using the index, we need one additional block access to the data file for a total of 6 + 1 = 7 block accesses—an improvement over binary search on the data file, which required 12 disk block accesses.

**Clustering Indexes**

If file records are physically ordered on a nonkey field—which *does not* have a distinct value for each record that field is called the **clustering field** and the data file is called a **clustered file.** We can create a different type of index, called a **clustering index**, to speed up retrieval of all the 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 disk block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, and it contains the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. Figure 18.2 shows an example. Notice that record insertion and deletion still cause problems because the data

records are physically ordered.

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

|  |
| --- |
|  |