ST. XAVIER’S COLLEGE

**(Affiliated to Tribhuvan University)**

Maitighar, Kathmandu



**Database Management System**

**Assignment#13**

**SUBMITTED BY:**

Prashraya Hada

013BSCCSIT027

**SUBMITTED TO:**

|  |  |
| --- | --- |
| **Sanjay Kumar Yadav**  Lecturer  St. Xavier’s College |  |

**Date of Submission:** November 3, 2015

**GRANT and REVOKE authorizations**

To be able to grant and revoke privileges and roles to and from users and roles, several prerequisites must be met.

The following table lists the prerequisites that a user must meet to grant privileges and roles to another user (or role).

| **Table 1: Prerequisites for Granting Privileges** | |
| --- | --- |
| **To grant...** | **The granting user needs...** |
| A system privilege | The system/object privilege being granted and be authorized to grant it to other users and roles |
| An object privilege on an object that exists only in runtime |
| An object privilege on an activated object created in the repository, such as a calculation view | The object privilege EXECUTE on the procedure GRANT\_PRIVILEGE\_ON\_ACTIVATED\_CONTENT |
| An object privilege on schema containing activated objects created in the repository, such as a calculation view | The object privilege EXECUTE on the procedure GRANT\_SCHEMA\_PRIVILEGE\_ON\_ACTIVATED\_CONTENT |
| A package privilege | The package privilege being granted and be authorized to grant it to other users and roles |
| An analytic privilege | The object privilege EXECUTE on the procedure GRANT\_ACTIVATED\_ANALYTICAL\_PRIVILEGE |
| An application privilege | The object privilege EXECUTE on the procedure GRANT\_APPLICATION\_PRIVILEGE |
| A role created in runtime | Either:   * The role being granted and be authorized to grant it to other users and roles, or * The system privilege ROLE ADMIN |
| A role created in the repository | The object privilege EXECUTE on the procedure GRANT\_ACTIVATED\_ROLE |

| **Table 2: Prerequisites for Revoking Privileges** | |
| --- | --- |
| **To revoke ...** | **The granting user needs...** |
| A system privilege | To be the user who granted the privilege |
| An object privilege on an object that exists only in runtime |
| An object privilege on an activated object created in the repository, such as a calculation view | The object privilege EXECUTE on the procedure REVOKE\_PRIVILEGE\_ON\_ACTIVATED\_CONTENT |
| An object privilege on schema containing activated objects created in the repository, such as a calculation view | The object privilege EXECUTE on the procedure REVOKE\_SCHEMA\_PRIVILEGE\_ON\_ACTIVATED\_CONTENT |
| A package privilege | The user who granted the privilege |
| An analytic privilege | The object privilege EXECUTE on the procedure REVOKE\_ACTIVATED\_ANALYTICAL\_PRIVILEGE |
| An application privilege | The object privilege EXECUTE on the procedure REVOKE\_APPLICATION\_PRIVILEGE |
| A role created in runtime | To be the user who granted the role |
| A role created in the repository | The object privilege EXECUTE on the procedure REVOKE\_ACTIVATED\_ROLE |

**Data Encryption**

Encrypting sensitive data in databases has clearly gone beyond optional, and is now a firm requirement. Whether an organization is looking to secure intellectual property, comply with privacy or regulatory mandates, or simply guard the organization’s brand against the damage associated with data breaches, database encryption represents a vital imperative.

By providing database encryption for sensitive data in databases, organizations can establish a strong line of defense that can help secure sensitive assets against a range of threats. However, while the reasons to adopt database encryption are clear, that doesn’t mean the effort is simple. In fact, for many organizations, database encryption has presented a range of obstacles, including degraded database performance, laborious revisions to application code, and complex and time consuming key management efforts.

**Transitivity, reflexivity and augmentation properties of fds**

Given that X, Y, and Z are sets of attributes in a relation R, one can derive several properties of functional dependencies. Among the most important are the following, usually called Armstrong's axioms:

* Reflexivity: If Y is a subset of X, then X → Y
* Augmentation: If X → Y, then XZ → YZ
* Transitivity: If X → Y and Y → Z, then X → Z

"Reflexivity" can be weakened to just X \rightarrow \varnothing, i.e. it is an actual axiom, where the other two are proper inference rules, more precisely giving rise to the following rules of syntactic consequence:

|- X ->ф

X -> Y |- XZ -> YZ

X ->Y, Y ->Z |-X ->Z.

These three rules are a sound and complete axiomatization of functional dependencies. This axiomatization is sometimes described as finite because the number of inference rules is finite, with the caveat that the axiom and rules of inference are all schemata, meaning that the X, Y and Z range over all ground terms (attribute sets).

**BCNF and decomposition of BCNF**

For relation R to be in BCNF, all the functional dependencies (FDs) that hold in R need to satisfy property that the determinants X are all superkeys of R. i.e. if X->Y holds in R, then X must be a superkey of R to be in BCNF.

In your case, it can be shown that the only candidate key (minimal superkey) is ACE. Thus both FDs: A->B and C->D are violating BCNF as both A and C are not superkeys or R.

**Decompose R into BCNF form:**

If R is not in BCNF, we decompose R into a set of relations S that are in BCNF.

This can be accomplished with a very simple algorithm:

Initialize S = {R}

While S has a relation R' that is not in BCNF do:

Pick a FD: X->Y that holds in R' and violates BCNF

Add the relation XY to S

Update R' = R'-Y

Return S

In your case the iterative steps are as follows:

S = {ABCDE} // Intialization S = {R}

S = {ACDE, AB} // Pick FD: A->B which violates BCNF

S = {ACE, AB, CD} // Pick FD: C->D which violates BCNF

// Return S as all relations are in BCNF

Thus R(A,B,C,D,E) is decomposed into a set of relations: R1(A,C,E), R2(A,B) and R3(C,D) that satisfies BCNF.

**Characterizing Schedules based on Recoverability**

When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history). A schedule (or history) S of n transactions T1, T2, …, Tn:

It is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of T1 in S must appear in the same order in which they occur in T1.

Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in S.

**Characterizing Schedules based on Serializability:**

DBMS must control concurrent execution of transactions to ensure read consistency, i.e., to avoid dirty reads etc. A (possibly concurrent) schedule S is serializable if it is equivalent to a serial schedule S0, i.e., S has the same result database state as S0.

**TRANSACTION SUPPORT IN SQL**

The definition of an SQL-transaction is that it is a logical unit of work and is guaranteed to be atomic. A single SQL statement is always considered to be atomic—either it completes execution without error or it fails and leaves the database unchanged.

The isolation level option is specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE. The default isolation level is SERIALIZABLE, although some systems use as READ COMMITTED their default. The use of the term SERIALIZABLE here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms, and it is thus not identical to the way serializability. If a transaction executes at a lower isolation level than SERIALIZABLE, then one or more of the following three violations may occur:

1. Dirty read: A transaction may read the update of a transaction , which has not yet committed. If fails and is aborted, then would have read a value that does not exist and is incorrect.

2. Non-repeatable read: A transaction may read a given value from a table. If another transaction later updates that value and reads that value again, will see a different value.

3. Phantoms: A transaction may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction inserts a new row that also satisfies the WHERE-clause condition used in, into the table used by. If is repeated, then will see a phantom, a row that previously did not exist.