**ST. XAVIER’S COLLEGE**

**(Affiliated to Tribhuvan University)**

Maitighar, Kathmandu



**DATABASE MANAGEMENT SYSTEM**

**THEORY ASSIGNMENT #12**

**Submitted by:**

Rojesh Tamrakar

013BSCCSIT032

**Submitted to:**

|  |  |
| --- | --- |
| **Er. Sanjay Kumar Yadav** |  |

Lecturer

Department of Computer Science

Date of submission: November 5th, 2015

1. **Grant And Revoke Authorizations**

**GRANT PRIVILEGES ON TABLE**

You can grant users various privileges to tables. These permissions can be any combination of SELECT, INSERT, UPDATE, DELETE, ALTER, or ALL.

**Syntax:**

The syntax for granting privileges on a table in SQL Server is:

GRANT privileges ON object TO user;

**Privileges**

The privileges to assign. It can be any of the following values:

|  |  |
| --- | --- |
| Privilege | Description |
| SELECT | Ability to perform SELECT statements on the table. |
| INSERT | Ability to perform INSERT statements on the table. |
| UPDATE | Ability to perform UPDATE statements on the table. |
| DELETE | Ability to perform DELETE statements on the table. |
| REFERENCES | Ability to create a constraint that refers to the table. |
| ALTER | Ability to perform ALTER TABLE statements to change the table definition. |

**Object**

The name of the database object that you are granting permissions for. In the case of granting privileges on a table, this would be the table name.

**User**

The name of the user that will be granted these privileges.

Example

Let's look at some examples of how to grant privileges on tables in SQL Server.

For example, if you wanted to grant SELECT, INSERT, UPDATE, and DELETE privileges on a table called *employees* to a user name*smithj*, you would run the following GRANT statement:

GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO smithj;

**REVOKE PRIVILEGES ON TABLE**

Once you have granted privileges, you may need to revoke some or all of these privileges. To do this, you can run a revoke command. You can revoke any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, or ALL.

**Syntax:**

The syntax for revoking privileges on a table in SQL Server is:

REVOKE privileges ON object FROM user;

**Privileges**

It is the privileges to assign. It can be any of the following values:

|  |  |
| --- | --- |
| Privilege | Description |
| SELECT | Ability to perform SELECT statements on the table. |
| INSERT | Ability to perform INSERT statements on the table. |
| UPDATE | Ability to perform UPDATE statements on the table. |
| DELETE | Ability to perform DELETE statements on the table. |
| REFERENCES | Ability to create a constraint that refers to the table. |
| ALTER | Ability to perform ALTER TABLE statements to change the table definition. |

**Object**

The name of the database’s object that you are revoking privileges for. In the case of revoking privileges on a table, this would be the table name.

**User**

The name of the user that will have these privileges revoked.

Example

Let's look at some examples of how to revoke privileges on tables in SQL Server.

For example, if you wanted to revoke DELETE privileges on a table called *employees* from a user named *anderson*, you would run the following REVOKE statement:

REVOKE DELETE ON employees FROM anderson;

1. **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.

## Vormetric Transparent Encryption

## Vormetric Application Encryption

## Key Management for Oracle and Microsoft SQL Server Database Encryption

**Data Encryption** helps to save data from following attacks:

* **Virtual attack**
* **Physical attack**
* **Power**
* **Flexibility**
* **Transparency**

1. **Transivity, 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](https://en.wikipedia.org/wiki/Armstrong%27s_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](https://en.wikipedia.org/wiki/Axiom), where the other two are proper [inference rules](https://en.wikipedia.org/wiki/Inference_rules), more precisely giving rise to the following rules of syntactic consequence:

\vdash X \rightarrow \varnothing  
X \rightarrow Y \vdash XZ \rightarrow YZ  
X \rightarrow Y, Y \rightarrow Z \vdash X \rightarrow Z.

These three rules are a [sound](https://en.wikipedia.org/wiki/Soundness) and [complete](https://en.wikipedia.org/wiki/Completeness_(logic)) 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](https://en.wikipedia.org/wiki/Schema_(logic)), meaning that the *X*, *Y* and *Z* range over all ground terms (attribute sets).

1. **BCNF & Decomposition into BCNF**

**BCNF:**

We say a relation R is in BCNF if whenever X → Y is a nontrivial FD that holds in R, X is a superkey. Remember: nontrivial means Y is not contained in X. Remember, a superkey is any superset of a key (not necessarily a proper superset)

Example:

Drinkers(name, addr, beersLiked, manf, favBeer)

FD’s: name → addr favBeer, beersLiked → manf

Only key is {name, beersLiked}

In each FD, the left side is not a superkey

Any one of these FD’s shows Drinkers is not in BCNF

**Decomposition into BCNF:**

The relation R with FD’s F § Look among the given FD’s for a BCNF violation X → Y

If any FD following from F violates BCNF, then there will surely be an FD in F itself that violates BCNF

Compute X+.

Not all attributes, or else X is a superkey.

1. **Characterizing Schedules based on Recoverability**

**Transaction schedule or history**:

* 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.

Schedules classified on recoverability:

**Recoverable schedule**:

* One where no transaction needs to be rolled back.
* A schedule S is recoverable if no transaction T in S commits until all transactions T’ that have written an item that T reads have committed.

**Cascadeless schedule**:

* One where every transaction reads only the items that are written by committed transactions.

**Schedules requiring cascaded rollback**:

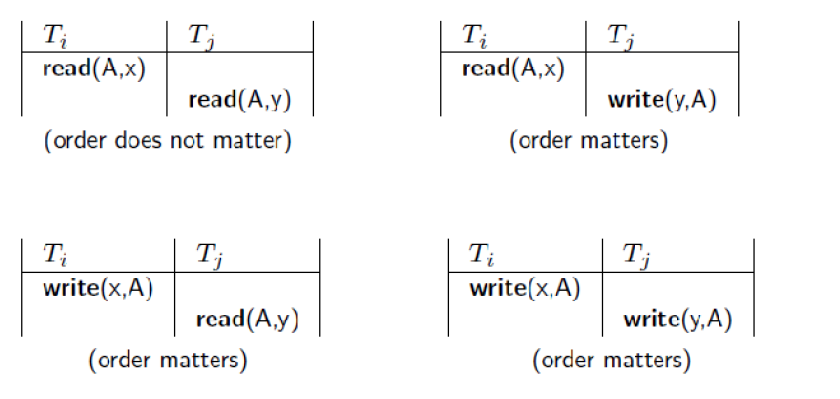
* A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.

1. **Characterizing Schedules based on Serializability**

**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.

**How to ensure serializability of concurrent transactions?**

Conflicts between operations of two transactions:



A schedule S is serializable with regard to the above conflicts iff S can be transformed into a serial schedule S' by a series of swaps of non-conflicting operations. Checks for serializability are based on precedence graph that describes dependencies among  
concurrent transactions; if the graph has no cycle, and then the transactions are serializable. - they can be executed concurrently without affecting each other’s transaction result.

1. **Transactions supports in SQL**

For this is a specific intended run (with specific parameters, e.g., with transaction identification, at least) of a computer program (or programs) that accesses a database (or databases). Such a program is written with the assumption that it is running in isolation from other executing programs, i.e., when running, its accessed data (after the access) are not changed by other running programs. Without this assumption the transaction's results are unpredictable and can be wrong. The same transaction can be executed in different situations, e.g., in different times and locations, in parallel with different programs. A live transaction (i.e., exists in a computing environment with already allocated computing resources; to distinguish from a transaction request, waiting to get execution resources) can be in one of three states, or phases:

* **Running** - Its program(s) is (are) executing.
* **Ready**- Program's execution has ended, and it is waiting to be Ended (Completed).
* **Ended (or Completed) –** It is either Committed or Aborted (Rolled-back), depending whether the execution is considered a success or not, respectively . When committed, all its recoverable (i.e., with states that can be controlled for this purpose), durable resources (typically database data) are put in their final states, states after running.

A failure in transaction's computing environment before ending typically results in its abort. However, a transaction may be aborted also for other reasons as well. Upon being ended (completed), transaction's allocated computing resources are released and the transaction disappears from the computing environment. However, the effects of a committed transaction remain in the database, while the effects of an aborted (rolled-back) transaction disappear from the database. The concept of atomic transaction ("all or nothing" semantics) was designed to exactly achieve this behavior, in order to control correctness in complex faulty systems.