**ST. XAVIER’S COLLEGE**

**(Affiliated to Tribhuvan University)**

**Maitighar, Kathmandu**

****

**Database Management System**

**Theory Assignment #13**

**SUBMITTED BY:**

**Siddhant Rimal**

**013BSCCSIT039**

**SUBMITTED TO**

|  |  |
| --- | --- |
| **Er. Sanjay Kr. Yadav**  **( Lecturer )** |  |
| **Department of Computer Science** | |

**Date of Submission: November 5th 2015**

13.1. GRANT and REVOKE authorizations

### GRANT authorization

### The grant statement is used to confer authorization

### grant <privilege list>

### on <relation name or view name> to <user list>

### <user list> is:

### a user­id

### public, which allows all valid users the privilege granted

### A role

### Granting a privilege on a view does not imply granting any privileges on the underlying relations. The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

### Privileges:

### select: allows read access to relation,or the ability to query using the view

### Example: grant users U1, U2, and U3 select authorization on the branch relation:

### grant select on branch to U1, U2, U3

### insert: the ability to insert tuples

### update: the ability to update using the SQL update statement

### delete: the ability to delete tuples.

### all privileges: used as a short form for all the allowable privileges

### REVOKE authorization

### The revoke statement is used to revoke authorization.

### revoke <privilege list>

### on <relation name or view name> from <user list>

### Example:

### revoke select on branch from U1, U2, U3

### <privilege­list> may be all to revoke all privileges the revokee may hold.

### If <revokee­list> includes public, all users lose the privilege except those granted it explicitly.

### If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

### All privileges that depend on the privilege being revoked are also revoked.

13.2. Data Encryption

### Data Encryption Standard (DES) substitutes characters and rearranges their order on the basis of an encryption key which is provided to authorize users via a secure mechanism. Scheme is no more secure than the key transmission mechanism since the key has to be shared.

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**

13.3. Transivity, Reflexivity and Augumentation properties of FDs

### Transitivity, Refleibity and Augumentaation properties of functional dependencies can be found out using Armstrong’s axioms.

### We can find all of F+ by applying Armstrong’s Axioms as follows: ● if β ⊆ α, then α → β (reflexivity) ● if α → β, then γ α → γ β (augmentation) ● if α → β, and β → γ, then α → γ (transitivity)

### Example of some members of F+

### ● A → H

### by transitivity from A → B and B → H

### ● AG → I

### by augmenting A → C with G, to get AG → CG and then transitivity with CG → I

### ● CG → HI

### by augmenting CG → I to infer CG → CGI, and augmenting of CG → H to infer CGI → HI, and then transitivity

13.4. BCNF & decomposition into BCNF

**13.4.1. Boyce-Codd Normal Form (BCNF)**

A relation schema R is in BCNF with respect to a set F of  
functional dependencies if for all functional dependencies in F+ of the form

α → β

### where α ⊆ R and β ⊆ R, at least one of the following holds:

### α → β is trivial (i.e., β ⊆ α)

### α is a superkey for R

### Example schema not in BCNF:

### bor\_loan = ( customer\_id, loan\_number, amount )

### because loan\_number → amount holds on bor\_loan but loan\_number is not a superkey

### 13.4.2. Decomposition into BCNF

### Suppose we have a schema R and a non­trivial dependency α →β causes a violation of BCNF.

### We decompose R into:

### (α U β )

### ( R ­ ( β ­ α ) )

### In our example,

### α = loan\_number

### β = amount

### and bor\_loan is replaced by

### (α U β ) = ( loan\_number, amount )

### ( R ­ ( β ­ α ) ) = ( customer\_id, loan\_number )

13.5. 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.

**Strict Schedules**:

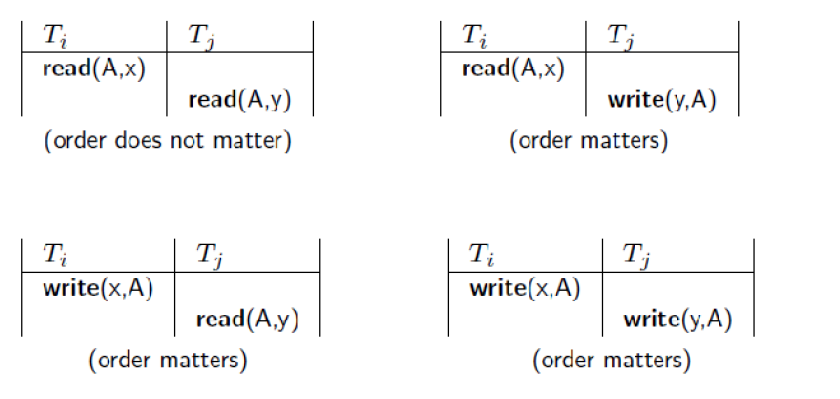
A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

13.6. 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.

13.7. Transaction 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.

REFERENCES

### Database System Concepts, 5th Edition, July 2005 , Silberschatz, Korth and Sudarshan, Chapter-4 : Advanced SQL

### Database System Concepts, 5th Edition, July 2005 , Silberschatz, Korth and Sudarshan, Chapter-7 : Relational Database Design

### Reference: Database System Concepts, 5th Edition, July 2005 , Silberschatz, Korth and Sudarshan, Chapter-8 : Application design and development

### Transaction Processing Concepts, Internet, Available: [www.cs.ucdavis.edu/~ludaesch/ECS-165A-FQ13/handouts/9-transactions.pdf](http://www.cs.ucdavis.edu/~ludaesch/ECS-165A-FQ13/handouts/9-transactions.pdf)

### “Database Encryption for Safeguarding Sensitive Data”, Internet, Available: <http://www.vormetric.com/uk/data-security-solutions/use-cases/database-encryption>