ST. XAVIER’S COLLEGE

**(Affiliated to Tribhuvan University)**

Maitighar, Kathmandu



**Database Management System Assignment #13**

**Submitted by:**

Bikash Paneru  
013BSCCSIT012

**Submitted to:**

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

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

**GRANT and REVOKE authorizations**

The SQL GRANT statement lets you grant explicit privileges to authorization IDs. The REVOKE statement lets you take them away. Only a privilege that has been explicitly granted can be revoked.

Granting privileges is very flexible. For example, consider table privileges. You can grant all the privileges on a table to an ID. Alternatively, you can grant separate, specific privileges that allow that ID to retrieve data from the table, insert rows, delete rows, or update specific columns. By granting or not granting those privileges on views of the table, you can effectively determine exactly what action an ID can or cannot take on the table.

You can use the GRANT statement to assign privileges as follows:

* Grant privileges to a single ID or to several IDs in one statement.
* Grant a specific privilege on one object in a single statement, grant a list of privileges, or grant privileges over a list of objects.
* Grant ALL, for all the privileges of accessing a single table or for all privileges that are associated with a specific package.

The same ID that grants a privilege can revoke it by issuing the REVOKE statement. If two or more grantors grant the same privilege to an ID, executing a single REVOKE statement does not remove the privilege for that ID. To remove the privilege, each ID that explicitly granted the privilege must explicitly revoke it.

**Data encryption**

Databases can be treasure troves of sensitive information. They can contain customers' personal data, confidential competitive information, and intellectual property. Lost or stolen data, especially customer data, can result in brand damage, competitive disadvantage, and serious fines—even lawsuits. Many of today’s privacy mandates require protecting data at rest, and the database is an obvious place where data accumulates and is potentially accessible to range of business systems and users. Organizations can choose to encrypt data at the application level, the database level, or the storage level. Encryption at the lowest of these levels, the storage level—on the disk or tape—guards against risk in the case where storage media are lost, but it does little to protect against malicious insiders or systems infected by malware . Application-level encryption on the other hand represents the other extreme by providing the highest degree of control, but it may not always be a viable approach.

A DBMS can use encryption to protect information in certain situations where the normal security mechanisms of the DBMS are not adequate.

**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:

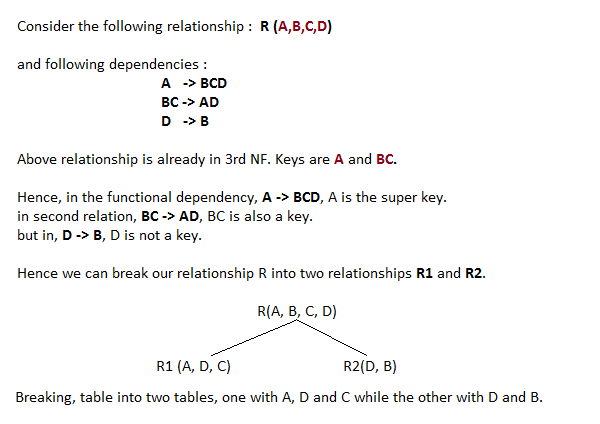
\vdash X \rightarrow \varnothing  
X \rightarrow Y \vdash XZ \rightarrow YZ  
X \rightarrow Y, Y \rightarrow Z \vdash X \rightarrow Z.

These three rules are a sound 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 into BCNF**

**Boyce and Codd Normal Form** is a higher version of the Third Normal form. This form deals with certain type of anamoly that is not handled by 3NF. A 3NF table which does not have multiple overlapping candidate keys is said to be in BCNF. For a table to be in BCNF, following conditions must be satisfied:

* R must be in 3rd Normal Form
* And, for each functional dependency ( X -> Y ), X should be a super Key.



**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.

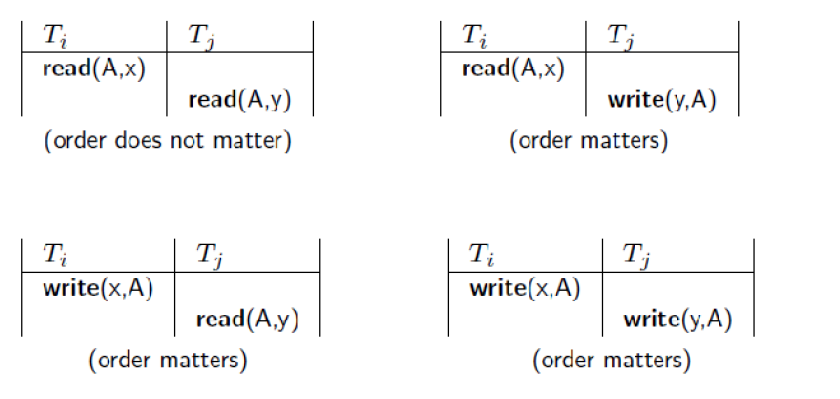
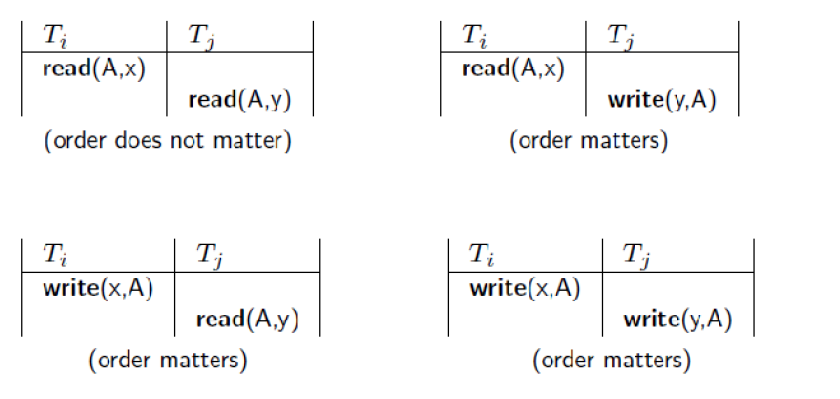
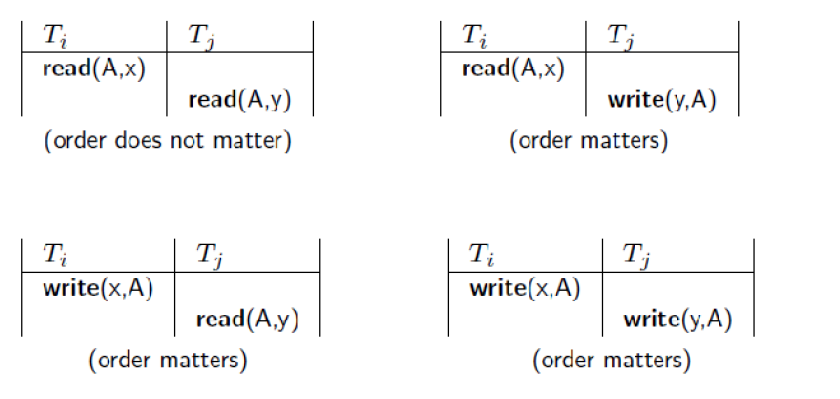
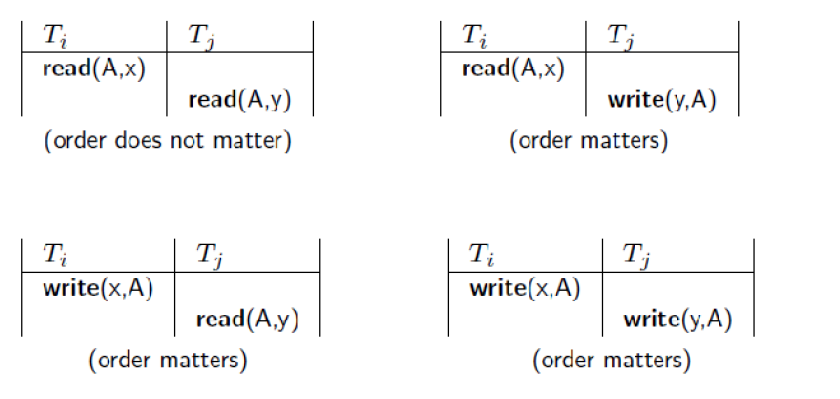
**Characterizing schedules based on Serializability**

**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?**

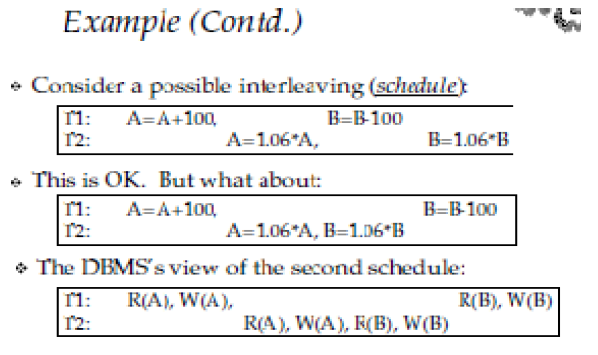
Conflicts between operations of two transactions:



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.

**Atomicity of Transactions**

A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.  
A very important property guaranteed by the DBMS for all transactions is that they are atomic.  
That is, a user can think of a Xact as always executing all its actions in one step, or not executing  
any actions at all.  
\_ DBMS logs all actions so that it can undo the actions of aborted transactions.  
Example  
Consider two transactions (Xacts):  
T1: BEGIN A=A+100, B=B-100 END  
T2: BEGIN A=1.06\*A, B=1.06\*B END  
Intuitively, the first transaction is transferring $100 from B’s account to A’s account. The  
second is crediting both accounts with a 6% interest payment.  
There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted  
together. However, the net effect must be equivalent to these two transactions running serially in  
some order.



**Transactions support in SQL**

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program.

There are following commands used to control transactions:

* **COMMIT:** to save the changes.
* **ROLLBACK:** to rollback the changes.
* **SAVEPOINT:** creates points within groups of transactions in which to ROLLBACK
* **SET TRANSACTION:** Places a name on a transaction.

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE only. They can not be used while creating tables or dropping them because these operations are automatically commited in the database.