DATABASE AUDIT

**Database auditing** involves observing a [database](https://en.wikipedia.org/wiki/Database) so as to be aware of the actions of database [users](https://en.wikipedia.org/wiki/User_(computing)). Database administrators and consultants often set up auditing for security purposes, for example, to ensure that those without the permission to access information do not access it.

**Auditing** is the monitoring and recording of selected user database actions. It can be based on individual actions, such as the type of SQL statement executed, or on combinations of factors that can include user name, application, time, and so on. Security policies can trigger auditing when specified elements in an Oracle database are accessed or altered, including the contents within a specified object.

Auditing is typically used to:

* Enable future accountability for current actions taken in a particular schema, table, or row, or affecting specific content
* Deter users (or others) from inappropriate actions based on that accountability
* Investigate suspicious activity

For example, if some user is deleting data from tables, then the security administrator might decide to audit all connections to the database and all successful and unsuccessful deletions of rows from all tables in the database.

* Notify an auditor that an unauthorized user is manipulating or deleting data and that the user has more privileges than expected which can lead to reassessing user authorizations
* Monitor and gather data about specific database activities

For example, the database administrator can gather statistics about which tables are being updated, how many logical I/Os are performed, or how many concurrent users connect at peak times.

* Detect problems with an authorization or access control implementation

For example, you can create audit policies that you expect will never generate an audit record because the data is protected in other ways. However, if these policies do generate audit records, then you will know the other security controls are not properly implemented.

This chapter describes the types of auditing available in Oracle systems, in the following sections:

* [Auditing Types and Records](http://docs.oracle.com/cd/B19306_01/network.102/b14266/auditing.htm#i1007052)
* [Statement Auditing](http://docs.oracle.com/cd/B19306_01/network.102/b14266/auditing.htm#i1006917)
* [Privilege Auditing](http://docs.oracle.com/cd/B19306_01/network.102/b14266/auditing.htm#i1007417)
* [Schema Object Auditing](http://docs.oracle.com/cd/B19306_01/network.102/b14266/auditing.htm#i1006926)
* [Fine-Grained Auditing](http://docs.oracle.com/cd/B19306_01/network.102/b14266/auditing.htm#i1009205)
* [Focusing Statement, Privilege, and Schema Object Auditing](http://docs.oracle.com/cd/B19306_01/network.102/b14266/auditing.htm#i1009530)
* [Auditing in a Multitier Environment](http://docs.oracle.com/cd/B19306_01/network.102/b14266/auditing.htm#i1006750)

*Auditing* an instance of SQL Server or a SQL Server database involves tracking and logging events that occur on the system. The *SQL Server Audit*object collects a single instance of server- or database-level actions and groups of actions to monitor. The audit is at the SQL Server instance level. You can have multiple audits per SQL Server instance. The *Database-Level Audit Specification* object belongs to an audit. You can create one database audit specification per SQL Server database per audit.

====================================================================================

A **Multi-Value Dependency (MVD)** occurs when two or more independent multi valued facts about the same attribute occur within the same table. It means that if in a relation R having A, Band C as attributes, B and Care multi-value facts about A, which is represented as A ààB and A ààC ,then multi value dependency exist only if B and C are independent on each other.

A **partial functional dependency** is a functional dependency where the determinant consists of key attributes, but not the entire primary key, and the determined consist~ of non-key attributes.

----------------------------------------------

In [databases](https://en.wikipedia.org/wiki/Database) and [transaction processing](https://en.wikipedia.org/wiki/Transaction_processing), **two-phase locking** (**2PL**) is a [concurrency control](https://en.wikipedia.org/wiki/Concurrency_control) method that guarantees [serializability](https://en.wikipedia.org/wiki/Serializability).[[1]](https://en.wikipedia.org/wiki/Two-phase_locking#cite_note-Bern1987-1)[[2]](https://en.wikipedia.org/wiki/Two-phase_locking#cite_note-Weikum2001-2) It is also the name of the resulting set of [database transaction](https://en.wikipedia.org/wiki/Database_transaction) [schedules](https://en.wikipedia.org/wiki/Schedule_(computer_science)) (histories). The protocol utilizes [locks](https://en.wikipedia.org/wiki/Lock_(computer_science)), applied by a transaction to data, which may block (interpreted as signals to stop) other transactions from accessing the same data during the transaction's life.

By the 2PL protocol locks are applied and removed in two phases:

1. Expanding phase: locks are acquired and no locks are released.
2. Shrinking phase: locks are released and no locks are acquired.

Two types of locks are utilized by the basic protocol: *Shared* and *Exclusive* locks. Refinements of the basic protocol may utilize more lock types. Using locks that block processes, 2PL may be subject to [deadlocks](https://en.wikipedia.org/wiki/Deadlock) that result from the mutual blocking of two or more transactions.

------------------------------------------

**TWO PHASE LOCKING**

A transaction is said to follow the **two-phase locking protocol**if *all*locking operations (read\_lock, write\_lock) precede the*first*unlock operation in the transaction. Such a transaction can be divided into two phases:

Phase 1: Growing Phase

i)  transaction may obtain locks

ii)  transaction may not release locks

Phase 2: Shrinking Phase

i)  transaction may release locks

ii)  transaction may not obtain locks

If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a read\_lock(*X*) operation that downgrades an already held write lock on *X*can appear only in the shrinking phase.

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock). Two-phase locking *does not* ensure freedom from deadlocks.

Types of 2PL:

* **Basic 2PL** – described above.
* **Conservative 2PL** (Static 2PL) - requires a transaction to lock all the items it accesses *before the transaction begins execution,*by **predeclaring**its *read-set*and *write-set.*
* Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
* **Rigorous two-phase locking** is even stricter: here *all*locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.  However, in the absence of extra information (e.g., ordering of  access to data), two-phase locking is needed for conflict serializability in the following sense:

    Given a transaction *T*i that does not follow two-phase locking, we can find a transaction *Tj* that uses two-phase locking, and a schedule for *Ti* and *Tj* that is not conflict serializable.

**Automatic Acquisition of Locks**

A transaction *T*i issues the standard read/write instruction, without explicit locking calls.

The operation **read**(*D*) is processed as:

**if** *Ti* has a lock on *D*

**then**

                                read(*D*)

**else begin**

                                   if necessary wait until no other

                                       transaction has a **lock-X** on *D*

                                   grant *Ti* a **lock-S** on *D*;

                                   read(*D*)

**end**

**write***(D)* is processed as:

**if***Ti* has a  **lock-X** on *D*

**then**

          write(*D*)

**else begin**

            if necessary wait until no other trans. has any lock on *D*,

            if *Ti* has a **lock-S** on *D*

**then**

**upgrade** lock on *D*  to **lock-X**

**else**

                    grant *Ti* a **lock-X** on *D*

                write(*D*)

**end**;

All locks are released after commit or abort

=============================================

|  |  |
| --- | --- |
| **Serial Schedule** | **Non-Serial Schedule** |
| A serial schedule is a sequence of operation by a set of concurrent transaction that preserves the order of operations in each of the individual transactions. | A non-serial schedule is a schedule where the operations of a group of concurrent transactions are interleaved. |
| Transactions are performed in serial order. | Transactions are performed in non-serial order, but result should be same as serial. |
| No interference between transactions | Concurrency problem can arise here. |
| It does not matter which transaction is executed first, as long as every transaction is executed in its entirely from the beginning to end. | The problem we have seen earlier lost update, uncommitted data, inconsistent analysis is arise if scheduling is not proper. |
| A serial schedule gives the benefits of concurrent execution without any problem | In this schedule there is no any benefit of concurrent execution. |
| Serial schedule that does interleaved the actions of different transactions. | Where non-serial schedule has no only fix actions of any transaction. |
| EXAMPLE:  If some transaction T is long, the other transaction must wait for T to complete all its operations. | EXAMPLE:  In this schedule the execution of other transaction goes on without waiting the completion of T. |
| If we consider transaction to be independent serial schedule is correct based on (property ACID) above assumption is valid. | The objective behind serializability is to find the non-serial schedule that allows transactions to execute concurrently without interfering one another. |

P

Schedules

• Schedule

– A sequence of reads/writes by a group of

transactions.

• Serial Schedule

– A schedule where transactions are executed

consecutively.

• Non-serial Schedule

– A schedule where the operations of a transaction are

interleaved.