Content

Serial and serializable plans

- If the transactions are processed one after another, then we talk about *serial plan*.
- A serial plan is often written as a tuple ordered according to the order of each transaction; for example, if A is processed before B then we write the serial plan as (A, B).
- How we recognize the correctly processed concurrent transactions?
- We have to define some 'measure of correctness': We use the *serializability*.

Serial and serializable plan

Equivalent plans

Two plans for the same transactions are equivalent if they return the same results.

Serializable plan

Transaction plan of two transactions is correct if and only if the plan is serializable: the plan is equivalent to any serial plan.

Serial plan, example (A, B)

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transakce A	**Time**	**Transaction B**
`READ` acc_1	t_1	-
$suma = 30$		
`READ` acc_2	t_2	-
$suma = 50$		
`READ` acc_3	t_3	-
$suma = 100$		
-	t_4	`READ` acc_3
-	t_5	`WRITE` $acc_3 = 60$
-	t_6	`READ` acc_1
-	t_7	`WRITE` $acc_1 = 20$
-	t_8	`COMMIT`
		$acc_1 = 20$
		$acc_2 = 20$
		$acc_3 = 60$

Serial plan, example (B, A)

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transaction A	**Time**	**Transaction B**
-	t_1	`READ` acc_3
-	t_2	`WRITE` $acc_3 = 60$
-	t_3	`READ` acc_1
-	t_4	`WRITE` $acc_1 = 20$
-	t_5	`COMMIT`
		$acc_1 = 20$
		$acc_2 = 20$
		$acc_3 = 60$
`READ` acc_1	t_6	-
$suma = 20$		
`READ` acc_2	t_7	-
$suma = 40$		
`READ` acc_3	t_8	-
$suma = 100$		

Remark: (A, B) a (B, A) can lead to a different results.

Problem of inconsistent analysis

The plan presented during the description of inconsistent analysis is not serializable (that means that the result is not equivalent to (A, B) nor (B, A)).

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transaction A	**tIME**	**Transaction B**
READ acc_1	t_1	-
$suma = 30$		
READ acc_2	t_2	-
$suma = 50$		
-	t_3	READ acc_3
-	t_4	WRITE $acc_3 = 60$
-	t_5	READ acc_1
-	t_6	WRITE $acc_1 = 20$
-	t_7	COMMIT
READ acc_3	t_8	-
$suma = 110$ ne 100		

We therefore see an example that serializability provides a measurement of concurrency transactions correctness.

Serializable plan and two-phase locking protocol

- Two-phase locking guarantees that the plan is always serializable.
- For example, locking protocol solving the problem of uncommitted dependency is equivalent with plan (B, A).

Transaction A	**Time**	**Transaction B**
-	t_1	`WRITE` t (X lock on t received)
`READ` t (S lock on t requested)	t_2	-
wait	t_3	`COMMIT/ROLLBACK` (X lock on t released)
opakuj: `READ` t (S lock on t received)	t_4	

Serializable plan and two-phase locking protocol

A locking protocol dealing with the problem of lost update finishes with deadlock, which is detected and if the transaction A is canceled, then the plan is equivalent with plan (B, A).

Transaction A	**Time**	**Transaction B**
`READ` t	t_1	-
(S lock on t received)		
-	t_2	`READ` t
		(S lock on t received)
`WRITE` t	t_3	-
(X lock on t requested)		
wait	t_4	`WRITE` t
		(X lock on t requested)
wait		wait
wait		wait

Theorem about the two-phase locking protocol

Authors of the following paper:

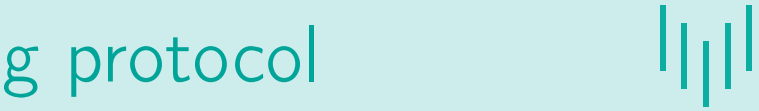
Kapali P. Eswaran, Jim Gray, Raymond A. Lorie ,and Irving L. Traiger: *The Notions of Consistency and Predicate Locks in a Database System*. In Commun. ACM 19(11), 1976, pages 624-633.

proof the following theorem:

Theorem about the two-phase locking protocol

If all transactions keep strict two-phase locking then all concurrent plans are serializable.

Theorem about the two-phase locking protocol

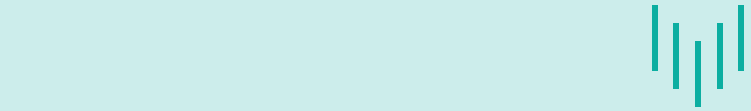


Let us introduce a simplified version of the previous two-phase protocol:

1. The transaction has to request a lock on a database object before it starts to perform some operations with this object.
2. After release of the first lock the transaction can not request any more locks.

Transactions keeping this protocol work in two phases: in the first phase they request locks and in the second phase they release the locks. The second phase can be simply processed by a COMMIT or ROLLBACK operation, where all locks are released.

Transaction Processing Performance



- Implementation of transaction processing (mainly the lock protocol) significantly influence the performance of RDBMS.
- Due to this fact there is still a lot of effort in this area. New locking protocols with higher throughput are developed (measured by a number of transactions per second).
- You can find the list of the best systems at the Transaction Processing Performance Council web page[1].
- RDBMS vendors then use these results to highlight the advantages of their systems.

[1] `http://www.tpc.org/`

Transaction isolation levels in SQL

- Serializability guarantees the isolation of transactions in the meaning of ACID.
- If a plan of transactions is serializable then there are not any concurrency issues.
- The isolation of transactions has an overhead – the performance is lower, i.e. throughput (the number of operations per second) is lower.
- Therefore, SQL specifications and DBMS provides some levels of transaction isolation.

Transaction isolation level in SQL

In SQL, there are 4 isolation levels (sorted from the lowest one to the highest one):

- READ UNCOMMITTED (RU)
- READ COMMITTED (RC)
- REPEATABLE READ (RR)
- SERIALIZABLE (SR)

Transaction isolation level in SQL

- A higher level means a higher isolation of transactions, however the throughput is lower and vice versa.
- As a result:
  - In the case of SERIALIZABLE, we get the maximum isolation of concurrent transactions.
  - On the contrary, lower isolations mean some concurrency issues.

READ COMMITTED 1/2

The READ COMMITTED level, for example, enables the following behaviour of the locking protocol:

1. We get a record t from database.
2. The lock S on t is retrieved.
3. If update is not required (and the lock X is not obtained) then
4. The lock S can be released before the transaction is finished.

When another transaction processes updates t and COMMIT and the first transaction processes repeatable read of t then it gets different values of t. Consequently, **non-repeatable read** appears and the result is an incorrect status of the database.

READ COMMITTED 2/2

- This plan does not comply two-phases locking: the lock is released before the transaction is finished.
- In the case of SR and RR levels, all locks are released in the time of the transaction's end. As a result, there are no concurrency issues.

Concurrency issues for transaction isolation levels

In the following table, we see concurrency issues appear for individual levels of transaction isolation:

Transaction isolation level	**Dirty read**	**Non-repeatable read**	**Phantom Read**
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

Phantom Read 1/3

Let us have the table `Student`:

login	name	class
joh001	John	1
geo002	George	3

When the level RR (or lower) is set the phantom occurrence can appear.

Phantom Read 2/3

Let us have the following plan:

Transaction A	Time	Transaction B
`SELECT * from Student`	t_1	-
`  WHERE class BETWEEN 1 AND 2`		
-	t_2	`INSERT INTO student`
		`VALUES('mar006',`
		`'Mark',2)`
	t_3	`COMMIT`
`SELECT * from student`	t_4	-
`  WHERE class BETWEEN 1 AND 2`		
`COMMIT`	t_5	-

Phantom Read 3/3

- In this case (RR and lower), a dbms returns different results for both queries of A in t_1 and t_4: the range of records of the table `Student` is not locked (records with values of the attribute `class` ≤ 1 and ≥ 2).
- In the case of SR, both queries return the same results.

Non-repeatable Read 1/2

- In the case of RC (and lower), *non-repeatable read* can appear.
- In this case, `SELECT` requires a **shared lock** on a record, however two-phases locking protocol is not complied and locks can be released before the transaction is finished.
- **Write locks** are however released at the end of the transaction.
- In the case of SR and RR, the behaviour is not enabled, in the case of RC and RU, it is enabled and this issue can appear.

Non-repeatable Read 2/2

Transaction A	Time	Transaction B
`SELECT * from Student` `  WHERE id='joh001'`	t_1	-

-	t_2	`UPDATE Student SET class=1`
		`  WHERE id='joh001'`
		`COMMIT`

`SELECT * from Student` `  WHERE id='joh001'`	t_3	-
`COMMIT`	t_4	

Dirty Read

In the case of RU, *dirty read* can appear, i.e. a transaction can read uncommitted updates of another transaction, see:

Transaction A	Time	Transaction B
-	t_2	`UPDATE Student SET class=1 WHERE id='joh001'`
`SELECT * from Student WHERE id='joh001'`	t_3	-
`COMMIT`	t_4	
	t_5	`COMMIT/ROLLBACK`

In this case, even *write locks* can be released before the end of a transaction.

START TRANSACTION command 1/3

```
START TRANSACTION <optional parameters>;
```

Where <*optional parameters*> are the following:

- *access mode*,
- *isolation level*,
- *diagnostics area size*.

START TRANSACTION command 2/3

- The **isolation level** is set using `ISOLATION LEVEL` <*level*>, where <*level*> is `READ UNCOMMITED`, `READ COMMITED`, `REPEATABLE READ` or `SERIALIZABLE`.
- The **access mode** can be `READ ONLY` or `READ WRITE`. The default value is `READ WRITE`. In the case of the `READ UNCOMMITED` level, the access mode `READ ONLY` is set. It means, if `READ WRITE` is set, the isolation level can not be `READ UNCOMMITED`.

START TRANSACTION command 3/3

- The **diagnostics area size** is set as an integer using `DIAGNOSTICS SIZE`, it means the number of exceptions stored on a stack.

Example

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT last_name, salary FROM employees
  WHERE last_name IN ('Pondělí','Středa','Pátek');

UPDATE employees SET salary = 9900
  WHERE last_name = 'Pondělí';

COMMIT;
```

Multiversioning

- *Locking* is called the pessimistic concurrency control approach: we assume that concurrent transactions affect each other.
- *Multiversioning* is another approach – an optimist approach: we assume that concurrent transactions do not affect each other.
- In locking, a system handles only single copy of data and locks are assigned.
- In multiversioning, a system handles more data copies with individual visibility for individual transactions.

Multiversioning, Example

Transactions read/update records of a table `Products(id, name, count)`.

t_1	Transaction A reads the record (48501, 'Axis - steel', 45). There is one copy in a database - it is labelled c_1.
t_2	Transaction B updates the record, the second copy is created (48501, 'Axis - steel', 44), it is lablled c_2. When B wants to read this record, it gets c_2.
t_3	Transaction C wants to read this record. Transaction B does not execute COMMIT, therefore C gets c_1.
t_4	Transaction B executes COMMIT, the system handle only one copy.
t_5	Transaction E wants to read this record, it gets the copy.

Locking, Example

The same example: Transactions read/update records of a table `Products(id, name, count)`.

t_1	Transaction A reads the record (48501, 'Axis - steel', 45), an S lock is assigned to the transaction.
t_2	Transaction B wants to update the record, in the case of SERIALIZABLE or REPEATABLE READ isolation level, the transaction is paused. If the isolation level of A is lower, the S lock can be released before the end. As a result, the Transaction B continues, the X lock is assigned and B updates the record: (48501, 'Axis - steel', 44).
t_3	Transaction C wants to read the record. When B handles the X lock, C is paused.
t_4	...

Example, Comparison

Observation 1:

- Multiversioning enables to processes a higher number of concurrent transactions.
- Locking can often cause waiting of concurrent transactions.

Observation 2:

- Plans are different but serializable.

Multiversioning, Pros and Cons

- Cons: the higher memory overhead for data copies handling.
- In the case of a higher number of read operations: multiversioning can be more efficient.
- In the case of a higher number of update operations of the same records, the overhead of multiversioning is higher.
- As a result, DBMSs often use both techniques (Oracle, SQL Server).

Oracle, Concurrency Control

- Oracle utilizes locking as well as multiversioning. Multiversioning does not enable dirty read (READ UNCOMMITTED is not therefore supported).

- Oracle supports two isolation levels of SQL and one own isolation level:
  - READ COMMITTED
  - READ ONLY
  - SERIALIZABLE

READ COMMITTED

- Each query of an executed transaction sees data committed before the starting of the query (not transaction).
- This isolation level is appropriate in the case of a low number of concurrency conflicts.
- However, non-repeatable read can appear (together with phantoms), a transaction can see committed updates of other transactions among queries of the transaction.

SERIALIZABLE 1/2

- Each query of a transaction sees updates committed with other transactions before the starting of the transaction (not query as in the case of READ COMMITTED).
- It is appropriate:
  - for short transactions updating less records,
  - when the probability, that two concurrent transactions update the same records, is low,
  - when long transactions use mainly read operations.

Notice: Evidently, the main issue is transactions executing more updates of the same records.

SERIALIZABLE 2/2

- Oracle executes a SERIALIZABLE transaction updating a record only when an update of a record executed with another transaction was committed before the SERIALIZABLE transaction has been started.
- In the opposite case, Oracle reports an error[2]: when a transaction tries to update records updated with another transaction in the case that the updates have been committed after the start of the first transaction.

[2] ORA-08177: Cannot serialize access for this transaction

READ-ONLY

- Similar to SERIALIZABLE however a transaction can not update records[3].
- This isolation level is appropriate for generating of reports when we need data the same during the complete time of a transaction.

[3] The SYS user can update the records.

SQL Server, Concurrency Control

- Transaction isolation levels:
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE
  - SNAPSHOT - similar to SERIALIZABLE, but ... (see a next slide).

SNAPSHOT vs SERIALIZABLE

- SERIALIZABLE: concurrency issues do not appear however transactions can block each other (since locks are utilized):
  - Operations of a transaction can not read data updated but uncommitted with other transactions.
  - Other transactions can not update data read with the current transaction until it will not finish.
  - Other transactions also can not insert records with values to appear in the range of reads of the current transaction until it will not finish.

SNAPSHOT vs SERIALIZABLE

- SNAPSHOT:
  - Similar to SERIALIZABLE from the SQL specification point of view: a transaction sees data committed before the transaction starts.
  - Locks are not used, multiversioning is used: a transaction reading data does not block other transactions, similarly a transaction writing data does not block reading transactions.
  - To enable SNAPSHOT, it is necessary to set `ALLOW_SNAPSHOT_ISOLATION` to `ON`:

    ```
    ALTER DATABASE <DB Name>
    SET ALLOW_SNAPSHOT_ISOLATION ON
    ```

    `READ_COMMITTED_SNAPSHOT` sets default isolation level to READ COMMITTED (since multiversioning is used).

References

- SQL Server:
  - https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql
  - https://www.sqlservercentral.com/articles/isolation-levels-in-sql-server
- Oracle:
  - https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-concurrency-and-consistency.html
  - https://blogs.oracle.com/oraclemagazine/on-transaction-isolation-levels