Co to są transakcje?

Transakcja to zbiór operacji (u nas - instrukcji języka SQL), które mogą być wykonane jedynie wszystkie lub żadna.

Nazwa takiego ciągu instrukcji pochodzi od operacji bankowych - przelew musi **jednocześnie** zabrać z jednego konta i dodać na drugie. W przypadku niepowodzenia żadna z tych operacji nie powinna mieć miejsca. Jeśli zajdzie tylko jedna skutki mogłyby być katastrofalne.

Transakcje opisuje zasada **ACID** - atomowość (**A**tomicity), spójność (**C**onsistency), izolacja (**I**solation) i trwałość (**D**urability):

- Atomowość - transakcja może być albo wykonana w całości albo w całości niewykonana.
- Spójność - stan bazy danych zawsze przedstawia stan przed lub po transakcji. Zapytania składane systemowi w czasie wykonywania transakcji muszą pokazywać sytuację przez transakcją, nie sytuację przejściową.
- Izolacja - transakcja dzieje się niezależnie od innych wykonywanych operacji, w tym od innych transakcji.
- Trwałość - w przypadku awarii systemu bazodanowego, np. w wyniku odcięcia elektryczności, transakcja będzie albo wykonana w całości albo wcale nie wykonana.

Obsługa transakcji jest bardzo ważna w bazach danych.

Systemy plików z journallingiem zmieniają dane systemu plików w sposób transakcyjny. Gwarantuje to że system plików jest stabilny nawet po awarii systemu operacyjnego.

Jednak sam zapis do plików nie jest wykonywany transakcyjnie, było by to zbyt kosztowne rozwiązanie. Istnieją jednak metody transakcyjnego zapisu danych do systemu plików - najprostsza to (na Uniksach):

- zapisujemy plik tymczasowy w którym znajdują się nowe dane. W przypadku krachu w tej fazie mamy stary plik nienaruszony.
- kasuje się poprzedni plik. Operacja jest atomowa. W przypadku krachu przed skasowaniem mamy oba pliki, w przypadku krachu po skasowaniu ale przed następną fazą mamy nowe dane, choć w złym pliku (należy je później odzyskać kończąc operacje).
- zmieniamy nazwę pliku. Operacja jest atomowa. Po tej operacji transakcja została dokończona.

Sposób niekoniecznie działa po NFS.

Przebieg transakcji

Przykład - koszyk zakupów w sklepie internetowym po złożeniu zamówienia

Jest to zwykle czas na wystawianie faktury (wpisy do tabeli np. faktura), która bazuje na zawartości koszyka (tabela koszyk_pozycje). Jeśli w momencie wpisu kolejnych zamówionych pozycji do tabeli faktura lub usuwania zawartości koszyka po złożeniu zamówienia (tabela koszyk_pozycje) wystąpi przerwanie komunikacji, przestanie działać system, itp. wówczas nastąpić mogą nieprzewidywalne przekłamania w zawartości wspomnianych tabel. Problem owych przekłamań rozwiązuje mechanizm transakcji, które możemy zapisać poniższym pseudo-kodem:

```
START_TRANSAKCJA;
  INSERT INTO faktura (...) VALUES (...);
```

```
$faktura_id = mysql_inserted_id();
foreach (koszyk_pozycje){
  INSERT INTO faktura_pozycje (...,faktura,..) VALUES (...,$faktura_id,... );
}
DELETE FROM koszyk_pozycje WHERE koszyk_id = ?
POTWIERDZ_TRANSAKCJA;
```

Transakcje rozpoczynają się poleceniem BEGIN (lub BEGIN WORK (lub, jak później zobaczymy, poleceniem START TRANSACTION)) i kończa słowem COMMIT.

Przykład

```
mysql> CREATE TABLE tab (f INT) TYPE=InnoDB;
```

Rozpocznijmy transackję, w której wstawimy nowy rekord do tablicy `tab`:

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tab(f) VALUES (1);
Query OK, 1 row affected (0.01 sec)
```

Zobaczmy, co znajduje się w tablicy `tab`

```
mysql> SELECT * FROM tab;
+---+
| f |
+---+
| 1 |
+---+
1 row in set (0.00 sec)
```

Nastepnie wykonajmy ROLLBACK

```
mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 sec)
```

Zobaczmy jeszcze co znajduje się w tablicy tab

```
mysql> SELECT * FROM tab;
Empty set (0.00 sec)
```

Bez komendy COMMIT wstawienie nowego rekordu nie było permanentne i zostało cofnięte poleceniemROLLBACK. Należy zauważyć, że nowy rekord byłby widoczny w tablicy w czasie wykonywania transakcji z poziomu tej samej sesji (z poziomu innej sesji nie był widoczny).

Spójne SELECTy

Spójrzmy na proces transakcji z poziomu dwóch różnych sesji.

Sesja 1

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tab (f) VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM tab;
+---+
| f |
+---+
| 1 |
+---+
1 row in set (0.00 sec)
```

Sesja 2

```
mysql> SELECT * FROM tab;
Empty set (0.00 sec)
```

A zatem wykonując to samo polecenie SELECT z poziomu różnych sesji (jednej, w czasie której wykonujemy transkację i drugiej, w czasie której polecenia sa wykonywane "na zewnątrz" transkacji) dostaniemy dwa różne rezultaty.

Dopiero po wykonaniu COMMIT w sesji pierwszej, wynik będzie taki sam z poziomu obu sesji.

Sesja 1

```
mysql> COMMIT; Query OK, 0 rows affected (0.00 sec)
```

Sesja 2

```
mysql> SELECT * FROM tab;
+---+
| f |
+---+
| 1 |
+---+
1 row in set (0.00 sec)
```

Taką właściwość nazywa się **spójnym czytaniem** lub **spójnym SELECT**. Każdy wykonany `SELECT` zwraca dane aktualne do ostatnio ZAKOŃCZONEJ transakcji.

SELECTy FOR UPDATE

Może się zdarzyć, że będziemy chcieli przeczytać rekord, w celu zmiany wartości niektórych z jego pól, mając jednocześnie pewność, że nikt inny nie będzie chciał w tym samym czasie wykonać tego samego. Na przykład dwóch użytkowników w czasie dwóch różnych sesji czytają ten sam rekord, w celu wstawienia następnego rekordu w którym pewna wartośc w pewnym polu będzie zwiększoną inkrementalnie wartością z pola przeczytanego właśnie rekordu, albo wartością maksymalną w tym polu (bierzącą wartością maksymlaną).

Sesja 1

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT MAX(f) FROM tab;
+--------+
| MAX(f) |
+--------+
|      3 |
+--------+
1 row in set (0.00 sec)

mysql> INSERT INTO tab(f) VALUES (4);
Query OK, 1 row affected (0.00 sec)
```

Sesja 2

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT MAX(f) FROM tab;
+--------+
| MAX(f) |
+--------+
|      3 |
+--------+
1 row in set (0.00 sec)

mysql> INSERT INTO tab(f) VALUES (4);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Sesja 1

```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    4 |
+------+
5 rows in set (0.00 sec)
```

W wyniku takich działań powstały dwa rekordy z wartością 4, podczas gdy chcieliśmy mieć jeden rekord z wartością 4 i jeden z wartością 5.

Aby zabezpieczyć się przed taką sytuacją musimy ograniczyć dostęp do rekordów tablicy. Można to zrobić za pomocą zamknięcia dostępu do tablicy do czasu, aż transakcja nie zostanie zakończona. Służy do tego klauzula`FOR UPDATE` dodawana do polecenia `SELECT`. Jest to więc specjalny `SELECT` wykonywany z myślą o tym, aby chwilę później wykonać `UPDATE`.

W przykładzie poniżej najpierw usuwamy błędne rekordy.

```
mysql> DELETE FROM tab WHERE f=4;
Query OK, 2 rows affected (0.00 sec)

mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT MAX(f) FROM tab FOR UPDATE;
+--------+
| MAX(f) |
+--------+
|      3 |
+--------+
1 row in set (0.00 sec)

mysql> INSERT INTO tab(f) VALUES (4);
```

```
Query OK, 1 row affected (0.00 sec)
```

Sesja 2

```
mysql> SELECT MAX(f) FROM tab FOR UPDATE;
```

Nie ma żadnych wyników. MySQL czeka, aż aktywna transakcja się zakończy i dopiero wówczas zwróci dane, które będą aktualne po zakończeniu transakcji w sesji 1.

Sesja 1

```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Dopierow w tym momencie wyniki są zwracane do sesji 2. Należy jeszcze dodać, że jeśli blokowanie trwało zbyt długo, wówczas MySQL zwróci informację, że został przekroczony czas oczekiwania.

Sesja 2

```
mysql> SELECT MAX(f) FROM tab FOR UPDATE;
+--------+
| MAX(f) |
+--------+
|      4 |
+--------+
1 row in set (4.20 sec)

mysql> INSERT INTO tab(f) VALUES(5);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
+------+
5 rows in set (0.00 sec)
```

SELECTy w trybie wspólnym

Kolejnym typem ograniczenia dostępu do danych jest tzw. `LOCK IN SHARE MODE`. Taki sposób ograniczania danych zapewnia dostęp do najświeższych danych (wprowadzanych w czasie transakcji) z zewnątrz transakcji. Takie udostępnianie danych blokuje wszystkie zmiany danych (polecenia `UPDATE` i `DELETE`) i, jeśli ostatnie zmiany nie były jeszcze potwierdzone poleceniem `COMMIT`, powoduje oczekiwanie na wynik zapytania dopóty, dopóki nie nastąpi potwierdzenie transakcji w sesji, która rozpoczęła tą trasnakcję.

Przykład.

Sesja 1

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT MAX(f) FROM tab LOCK IN SHARE MODE;
+--------+
| MAX(f) |
+--------+
|      5 |
+--------+
1 row in set (0.00 sec)
```

W tym czasie użytkownik w innej sesji próbuje wykonać `UPDATE`

Sesja 2

```
mysql> UPDATE tab SET f = 55 WHERE f=5;
```

Jednak polecenie oczekuje dopóty dopóki nie nastąpi zakończenie transakcji w sesji 1.

Sesja 1

```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Sesja 2

```
mysql> UPDATE tab SET f = 55 WHERE f=5;
Query OK, 0 rows affected (6.95 sec)
Rows matched: 0 Changed: 0 Warnings: 0

mysql> UPDATE tab SET f = 55 WHERE f=5;
Query OK, 1 row affected (43.30 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> SELECT * FROM tab;
+------+
```

```
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
+------+
5 rows in set (0.00 sec)
```

Nie wszystko można cofnąć

Niektóre polecenia SQL nie mogą być cofnięte, pomimo tego, że wykonywane będą w transakcji. Należą do nich, generalnie, wszystkie polecenia języka DDL, czyli takie, za pomoca których tworzymy lub usuwamy bazy danych, albo tworzymy lub usuwamy tablice w obrębie bazy. Nie mogą być też cofane polecenia, które definiują w obrębie transakcji tzw. procedury przypisane do tablic (tzw. stored procedures).

Należy tak zaprojektować transakcje, aby nie wykonywać w jej obrębie takich poleceń. Jeśli w obrębie transakcji użyjemy polecenia, za pomocą którego utworzymy tablicę, a następnie użyjemy polecenia, które nie zostanie poprawnie wykonane z powodu jakiegoś błędu, wówczas trzeba się liczyć z tym, że po wykonaniu poleceniaROLLBACK nie wszystkie efekty wykonania różnych poleceń zostaną cofnięte.

Wyrażenia, które wywołują automatycznie COMMIT

Niektóre polecenia automatycznie kończą transakcję pomimo tego, że nie wykonamy explicite polecenia COMMIT.

ALTER TABLE	BEGIN	CREATE INDEX
DROP DATABASE	DROP INDEX	DROP TABLE
LOAD MASTER DATA	LOCK TABLES	RENAME TABLE
SET AUTOCOMMIT=1	START TRANSACTION	TRUNCATE TABLE

Polecenie UNLOCK TABLES kończy transakcję ze skutkiem COMMIT nawet jeśli jakies tablice są w danym momencie zablokowane.

```
mysql> SAVEPOINT indetyfikator
mysql> ROLLBACK TO SAVEPOINT indetyfikator
```

Wyrażenie SAVEPOINT ustawia pewne miejsce w transakcji o nazwie indetyfikator. Jeśli jakaś transakcja ma już oznaczone w taki sam sposób (za pomocą tego samego inentyfikatora) miejsce, wówczas to miejsce jest zamazywane przez nowe miejsce.

Wyrażenie `ROLLBACK TO SAVEPOINT` cofa transakcję do punktu oznaczonego przez `indetyfikator`. Zmiany, które zaszły w rekordach po miejscu oznaczonym indetyfikatorem, są cofane poleceniem `ROLLBACK`, natomiast te, które były wykonane przed identyfikatorem, nie są cofane. Identyfikatory, które zostały ustawione po identyfikatorze, do którego odwołaliśmy się w poleceniu `ROLLBACK TO SAVEPOINT` są usuwane.

Jesli wyrażenie `ROLLBACK TO SAVEPOINT` zwraca błąd

```
ERROR 1181: Got error 153 during ROLLBACK
```

to oznacza to, że nie istnieje miejsce oznaczone przez identyfikator, do którego się odnosiliśmy.

Jeśli użyjemy zwykłego COMMIT lub ROLLBACK, wówczas wszystkie identyfikatory miejsc zostaną usunięte.

Poziom izolacji transakcji

Poziom izolacji transakcji wpływa bezpośrednio na zachowanie się transakcji. Zmiana poziomu izolacji może prowadzić do zupełnie różnych wyników poleceń SQL.

Poziom izolacji transakcji oznacza jak "szczelnie" jest zaizolowana tranasakcja i jakiego rodzaju izolacja jest skojarzona z zapytaniami wewnątrz transakcji. Można wybrać jeden z czterech poziomów izolacji (wymienionych poniżej w kolejności rosnącej szczelności izolacji).

1. `READ UNCOMMITTED`
   Ustawienie takiego poziomu transakcji powoduje dopuszczenie tzw. "dirty reads", tzn. że niepotwierdzone poleceniem `COMMIT` efekty poleceń z jednej transakcji są widoczne z poziomu drugiej transakcji.
2. `READ COMMITTED`
   Potwierdzone poleceniem `COMMIT` zmiany danych w tablicach są widoczne z poziomu innych transakcji. Oznacza to, że identyczne polecenia w obrębie tej samej transakcji mogą zwrócić zupełnie inne wyniki. W niektórych systemach baz danych jest to domyślny sposób izolacji transakcji.
3. `REPEATABLE READ`
   Jest to domyślny sposób izolacji transakcji dla tablic typu InnoDB. W obrębie transakcji wszystkie zapytania są spójne.
4. `SERIALIZABLE`
   Jeśli w obrębie jednej transakcji wykonujemy właśnie polecenie `SELECT` wówczas z poziomu dowolnej innej transakcji nie możemy wykonać zmiany danych, które są właśnie wybierane poleceniem `SELECT`. Inaczej mówiąc zapytania w obrębie tranakcji są wykonywane tak, jakby automatycznie była do nich dołączana klauzula `LOCK IN SHARE MODE`.

Tablice InnoDB wspierają wszystkie cztery poziomy izolacji transakcji. Przy przenoszeniu kodów SQL na inny system baz danych, należy mieć świadomość, że nie szystkie wymienione wyżej poziomy izolacji są wspierane przez inne systemy baz danych, a co więcej, w niektórych z nich domyślnym poziomem izolacji jest zupełnie inny poziom niż w MySQL.

- **SQL SERVER** - domyślnie `READ COMMITTED`, poza tym, nie ma żadnych innych poziomów izolacji.
- **Oracle** - domyślnie `READ COMMITTED`, poza tym można wybrać też `SERIALIZABLE` i niestandardowy`READ ONLY`.

- **DB2** - domyślnie `REPEATABLE READ`, poza tym można wybrać też `UNCOMMITTED READ` oraz inne niestandardowe poziomy izolacji.
- **PostgreSQL** - domyślnie `REPEATABLE READ`, poza tym można też wybrać `SERIALIZABLE`.

Przykłady,

Zakładamy, że mamy następującą tablicę `tab` z danymi:

```
mysql> CREATE TABLE tab (f INT) TYPE = InnoDB;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tab values (1),(2),(3),(4),(55);
Query OK, 5 rows affected (0.00 sec)
```

Na początek sprawdźmy jaki poziom izolacji transakcji obowiązuje w danej chwili (jeśli tego nie zmieniliśmy my lub administrator to domyslnym poziomem izolacji transakcji w MySQL jest `REPEATABLE READ`).

```
mysql> SELECT @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set (0.00 sec)
```

Repeatable Read

Zobaczmy, czy polecenie `INSERT` wykonane w obrębie jednej transakcji i potwierdzone następnie poleceniem`COMMIT` jest widoczne z poziomu drugiej transakcji.

Sesja 1

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
+------+
5 rows in set (0.00 sec)
```

Sesja 2

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tab VALUES(6);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

Pamiętamy, że nie ma znaczenia dla transakcji w sesji 2, że polecenie SELECT zostało wykonane po poleceniu COMMIT. W obrębie transakcji nowy rekord jest natychmiast "widzialny" (równie dobrze moglibyśmy wykonać SELECT przed wykonaniem polecenia COMMIT).

Sesja 1

```
mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
+------+
5 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
```

```
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

To właśnie jest idea blokowania typu **Repeatable Read**. Wkonane polecenie SELECT zwraca wynik, który charakteryzuje się spójnością, a nowe rekordy dodane do tablicy z poziomu innej transakcji nie są od razu widoczne. Aby były widoczne należy bezwzględnie zakończyć transakcję.

Uncommitted Read

Zobaczmy jak zachowują się transakcje w trybie **Uncommitted Read**. Musimy w tym celu zmienić poziom izolacji transakcji z domyślnego na Uncommitted Read właśnie. Aby to uczynić musimy mieć przywilej `SUPER`.

```
mysql> SET GLOBAL TRANSACTION ISOLATION LEVEL
       READ UNCOMMITTED;
Query OK, 0 rows affected (0.00 sec)
```

Ponownie użyjemy dwóch (nowych!) sesji.

Sesja 1

```
mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

Sesja 2

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO tab VALUES (7),(8);
Query OK, 1 row affected (0.06 sec)
```

Sesja 1

```
mysql> SELECT * FROM tab;
+------+
| f    |
+------+
```

```
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
|    7 |
|    8 |
+------+
8 rows in set (0.00 sec)
```

To właśnie jest tzw. "dirty read" - nowe rekordy nie zostały jeszcze nawet potwierdzone w drugiej transakcji a już są widoczne z poziomu pierwszej transakcji.

Sesja 2

```
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
```

Sesja 1

```
mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

Taki poziom izolacji jest niebezpieczny i właściwie łamie zasady ACID. Używa się takiego trybu pracy transakcji w przypadku, kiedy nie interesuje nas spójność danych, a jedynie dostęp do najświeższych danych z poziomu dowolnej transakcji.

Committed Read

Ponownie trzeba zmienić poziom izolacji i uruchomić dwie nowe (!) sesje.

```
mysql> SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)
```

Sesja 1

```
mysql>BEGIN;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

Sesja 2

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t VALUES (7),(8);
Query OK, 1 row affected (0.05 sec)
```

Sesja 1

```
mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
+------+
6 rows in set (0.00 sec)
```

Sesja 2

```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Sesja 1

```
mysql> SELECT * FROM tab;
+------+
| f    |
+------+
|    1 |
```

```
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
|    7 |
|    8 |
+------+
8 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Istotną różnicą jest to, że niepotwierdzone poleceniem COMMIT polecenie INSERT nie wpłynęło na aktualny stan bazy danych widziany z poziomu drugiej transakcji. Dopiero po potwierdzeniu transakcji (po jej zakończeniu) widoczne sa zmiany w tablicach. Jest też różnica pomiędzy tym poziomem izolacji (READ COMMITTED) a omówionym pierwszym domyślnym poziomem izolacji (REPEATABLE READ). W trybie READ COMMITTED zmiany w tablicach widoczne są już wówczas, gdy transakcja, w której zostały wykonane potwierdzi je poleceniem COMMIT, nawet wówczas gdy w danej transakcji nie wykonano jeszcze COMMIT. W trybie REPEATABLE READ, zmiany są widoczne dopiero wówczas, gdy w obu transakcjach wykonane zostana polecenia potwierdzenia (COMMIT).

Serializable

```
mysql> SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
Query OK, 0 rows affected (0.00 sec)
```

Tryb SERIALIZABLE posuwa się o krok dalej niż tryb REPEATABLE READ. W trybie SERIALIZABLE wszystkie zwyczajne polecenia SELECT są traktowane jakby były wykonywane z klauzulą LOCK IN SHARE MODE.

Sesja 1

```
mysql> BEGIN;
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT * FROM tab
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
|    7 |
|    8 |
+------+
8 rows in set (0.00 sec)
```

Sesja 2

```
mysql> BEGIN;
Query OK, 0 rows affected (0.06 sec)

mysql> UPDATE tab SET f=88 WHERE f=8;
```

Z powodu wykonania polecenia SELECT w sesji 1 polecenie UPDATE wykonywane w sesji 2 czeka aż po poleceniu SELECT (w sesji 1) nie zostanie wykonane polecenie COMMIT (tak, jak przy zwykłym LOCK IN SHARE MODE). Dopiero, kiedy w sesji 1 wykonane zostanie polecenie COMMIT kończące transakcję, wówczas zostanie wykonanie polecenie UPDATE w sesji 2.

Sesja 1

```
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
```

Sesja 2

```
Query OK, 1 rows affected (4.23 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM tab
+------+
| f    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|   55 |
|    6 |
|    7 |
|   88 |
+------+
8 rows in set (0.00 sec)
```

Konkluzje

Tryb REPEATABLE READ jest domyślnym poziomem izolacji transakcji w MySQL i nie powinniśmy tego raczej zmieniać. Jak widzieliśmy, pomiędzy różnymi trybami izolacji transakcji są pewne subtelności i jeśli ich nie poznamy a bedziemy używać, wówczas może się zdarzyć że będziemy kiedyś oczekiwać długie godziny, zanim nasze polecenia wykonywane w bazie danych odniosą trwały skutek.

Składnia polecenia `START TRANSACTION`

```
START TRANSACTION [WITH CONSISTENT SNAPSHOT]
```

Domyślnie, MySQL pracuje z włączoną opcją AUTOCOMMIT (zresztą nie tylko MySQL). Aby więc możliwe było wykonywanie transakcji należy tą opcję wyłączyć.

```
mysql> SET AUTOCOMMIT = 0;
```

Należy jednak pamiętać, że tak wyłączona opcja oznacza, że każda operacja na bazie danych nie zostanie trwale zapisana, dopóty, dopóki nie wykonamy polecenia `COMMIT`. W praktyce więc transakcja rozpoczyna się nie wykoananiem polecenia `BEGIN` ale poleceniem `SET AUTOCOMMIT = 0;` i podobnie, transakcja nie powinna się kończyć poleceniem `COMMIT` ale dodatkowo należy jeszcze włączyć z powrotem opcję AUTOCOMMIT: `SET AUTOCOMMIT = 1;`. Dopiero wtedy będziemy mieli wykonaną nasza transakcję i będziemy mogli wykonywać normalne zmiany w bazie danych bez używania transakcji.

Inną możliwością jest rozpoczęcie transakcji poleceniem `START TRANSACTION`. W takim wypadku tryb AUTOCOMMIT zostaje zawieszony automatycznie na czas wykonywania ciągu operacji i uruchomiony ponownie w momencie wykonania polecenia `COMMIT` lub `ROLLBACK`

Przykład:

```
mysql> START TRANSACTION;
mysql> SELECT @A:=SUM(pensja) FROM tab1 WHERE type=1;
mysql> UPDATE tab2 SET suma=@A WHERE type=1;
mysql> COMMIT;
```

Poleceń `BEGIN` i `BEGIN WORK` można używać zamiast polecenia `START TRANSACTION` w celu rozpoczęcia transakcji. Polecenie `START TRANSACTION` zostało dodane w wersji 4.0.11 MySQLa.

Od wersji 4.1.8 można rozpocząć transakcję w następujący sposób

```
mysql> START TRANSACTION WITH CONSISTENT SNAPSHOT;
```

Klauzula `WITH CONSISTENT SNAPSHOT` rozpoczyna uruchamianie pewnych procedur (tzw. stored engines) związanych na stałe z tablicami, które musza być typu `InnoDB` Efekt jest taki sam, jak po uruchomieniu `START TRANSACTION` a następnie polecenia `SELECT`.

W czasie wykonywania transakcji należy uzywać tablic transakcyjnych. Można jednak używać wewnątrz transakcji tablic nietransakcyjnych, jednak wówczas należy się liczyć z tym, że w momencie dokonania zmian w takich tablicach (UPDATE, INSERT, REPLACE) i następnie polecenia `ROLLBACK` zmiany w takich tablicach zostaną zapisane, pomimo wykonania `ROLLBACK`. Jedynie zmiany w tablicach transakcyjnych zostaną cofniete.

Składnia polecenia `SET TRANSACTION`

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ |
SERIALIZABLE }
```

To polecenie ustawia (SET) poziom izolacji transakcji dla następującej po tym poleceniu transakcji, albo globalnie (dla wszystkich transakcji), albo dla bierzącej, właśnie wykonywanej transakcji.

Domyślnie polecenia `SET TRANSACTION` ustawia poziom izolacji dla następującej po tym poleceniu transakcji (jeszcze nie rozpoczetej). Opcja `GLOBAL` ustawia poziom izolacji globalnie, dla wszystkich połączeń z baza danych ustanowionych od mementu wykonania tego polecenia, przy czym istniejace połączenia nie są zmieniane.

Aby móc wykonać polecenie `SET TRANSACTION GLOBAL` trzeba mieć przywileje administratora (przywilej`SUPER`). Opcja `SESSION` użyta w składni `SET TRANSACTION` powoduje ustawienie domyślnego poziomu izolacji transakcji dla wszystkich transakcji wykonywanych w ramach bierzącej sesji. Domyślnym poziomem jest`REPEATABLE READ`.

Składnia poleceń `LOCK TABLES` i `UNLOCK TABLES`

```
LOCK TABLES
  tablica [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
  [, tablica [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}]
...
UNLOCK TABLES
```

Polecenie `LOCK TABLES` blokuje dostęp do danych w tablicy. Jeśli jakaś tablica, do której dostęp zamierzamy zablokować, jest już zablokowana, wówczas próba jest powtarzana aż do osiągnięcia celu (czyli aż do zablokowania tablicy).

Polecenie `UNLOCK TABLES` zwalania dostęp do tablic, do których dostęp był zabroniony poprzednio wykonanym poleceniem `LOCK TABLES`. Wystarczy wydać jedno polecenie `LOCK TABLES` aby odblokować dostęp do wszystkich zablokowanych wcześniej tablic (w danym wątku (danej sesji)). Dostęp do tablic odblokowauje też zrestartowanie serwera baz danych.

Głównym powodem do używania polecenia `LOCK TABLES` jest symulowanie transakcji.

Polecenie `LOCK TABLES` nie jest bezpieczne w przypadku transakcji

- Automatycznie kończy rozpoczetą transakcję. Z drugiej strony rozpoczęcie transakcji (na przykład poleceniem`START TRANSACTION` automatycznie wywołuje `UNLOCK TABLES`.
- Prawidłowym sposobem użycia `LOCK TABLES` z tablicami transakcyjnymi, jest wykonanie `AUTOCOMMIT = 0` i nie wywoływanie `UNLOCK TABLES` dopóty, dopóki nie wykonamy `COMMIT`. Kiedy wykonujemy`LOCK TABLES` InnoDB wewnętrznie wykonuje blokowanie dostępu do tablic na swój sposób, jednocześnie serwer MySQL wykonuje blokowanie dostępu do tablic. Jednak aby zwolnic dostęp do tablic nietransakcyjnych trzeba wykonać polecenie `UNLOCK TABLES`, podczas gdy InnoDB zwalnia dostęp do tablic w momencie najbliższego `COMMIT`. Nie powinniśmy mieć ustawionej na 1 zmiennej `AUTOCOMMIT`, gdyż w takim wypadku InnoDB zwalnia dostep do tablic automatycznie po wykonaniu jakiejkolwiek komendy (gdyż wówczas każde polecenie jest jakby transakcja, i wykonywane jest po nim `COMMIT`).
- `ROLLBACK` nie zwalnia automatycznie dostępu do zablokowanych tablic nietransakcyjnych.

Aby wykonywać polecenie `LOCK TABLES` trzeba mieć przywileje `LOCK TABLES` i `SELECT`.

Kiedy w czasie jakejś sesji (jakiegoś wątku) nastąpi zablokowanie dostępu do czytania danych z tablicy, wówczas osoba z innego wątku nie może czytać danych z zablokowanej tablicy. Może to zrobić jedynie osoba która wykonała blokade tablicy. Jeśli w jakiejś sesji nastapi zablokowanie dostępu do pisania do tablicy, wówczas może w niej pisać jedynie osoba, która wykonała blokadę; inne osoby nie mogą dokonywac zmian w tablicy.

Kiedy blokujemy dostęp do danych, musimy zablokować wszystkie tablice, z których zamierzamy korzystać w czasie blokady. W czasie blokady nie możemy korzystać z tablic, które nie były zablokowane. Nie możemy też używać zablokowanych tablic wielokrotnie w tym samym zapytaniu, powinniśmy w takim wypadku stosować aliasy. W takim wypadku należy uzyskać blokadę dla każdego aliasu używanego w zapytaniu.

Przykład

wykonujemy najpierw blokadę tablicy i aliasu

```
mysql> LOCK TABLE t WRITE, t AS t1 WRITE;
```

Następnie, w poleceniu INSERT wstawiamy dane, uzyskane najpierw poleceniem SELECT z tablicy. Ale w tym wypadku akurat tablica, z której uzyskujemy dane wyrażeniem SELECT jest tą samą tablicą, do której wstawiamy dane. W jednym wyrażeniu więc posługujemy się tą samą tablicą.

```
mysql> LOCK TABLE t WRITE, t AS t1 WRITE;
mysql> INSERT INTO t SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
```

Baza zwróciła błąd, gdyż po zablokowaniu chcieliśmy uzyskać dane przy pomocy tej samej nazwy tablicy.

W takim wypadku musimy się posłużyć aliasem, dopiero to odniesie skutek.

```
mysql> INSERT INTO t SELECT * FROM t AS t1;
```

Kolejny problem

Jeśli nasze zapytania odnoszą się do tablic i używają aliasów, wówczas musimy zablokować tablicę używając dokładnie tych samych aliasów. W takim wypadku nie zadziała zablokowanie dostępu do tablicy bez użycia aliasu.

```
mysql> LOCK TABLE t READ;
mysql> SELECT * FROM t AS myalias;
ERROR 1100: Table 'myalias' was not locked with LOCK TABLES
```

Analogicznie, jeśli blokujemy dostęp do tablicy używając aliasu, wówczas musimy odnosić się do tej tablicy w czasie zapytań jedynie poprzez alias.

```
mysql> LOCK TABLE t AS myalias READ;
mysql> SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
```

Zadziała dopiero następujące polecenie

```
mysql> SELECT * FROM t AS myalias;
```

Normalnie, nie powinniśmy blokować dostępu do tablic, ponieważ wszystkie operacje `UPDATE` są atomowe. Jednak jest kilka sytuacji, w których chcielibyśmy używać `LOCK TABLES`.

- Pierwsza taką sytuacją jest kiedy używamy tablic nietransakcyjnych `MyISAM`. Wstawianie rekordów, uaktualnianie ich i usuwanie jest szybsze w przypadku, kiedy tablice są zablokowane. Jest to wewnętrzne zachowanie MySQL.
- Jeśli używamy procedur (stored procedures), które nie wspierają transakcji. W takim wypadku musimy być pewni, że nie będzie żadnego polecenia pomiędzy `SELECT` i `UPDATE`

```
mysql> LOCK TABLES trans READ, klient WRITE;
mysql> SELECT SUM(value) FROM trans WHERE klient_id=jakis_id;
mysql> UPDATE klient
    SET wartość=suma_z_poprzedniego_wyrazenia
    WHERE klient_id=jakis_id;
mysql> UNLOCK TABLES;
```

- Bez użycia `LOCK TABLES` jest możliwe że w innym wątku (innej sesji) zostanie wprowadzony nowy rekord do tablicy `trans` pomiędzy wykonaniem `SELECT` i `UPDATE`.

Blokady `WRITE` normalnie mają wyższy priorytet niż blokady `READ`. dla zapewnienia wykonania zmaian w tablicy tak szybko jak to tylko możliwe. Oznacza to, że jeśli tylko tablica otrzyma w jednym wątku polecenie zablokowania do czytania (`READ`) a następnie z innego wątku polecenie zablokowania do pisania (`WRITE`), wówczas blokada`READ` czeka aż blokada `WRITE` zakończy działanie i zwolni tablicę. Można więc użyć opcji `LOW_PRIORITY WRITE` aby pozwolić na czytanie z tablicy z innych wątków, w czasie kiedy jest ona zablokowana do pisania. Opcja ta pozwala więc na wykonanie `LOCK TABLE tab READ;` z innego wątku w momencie kiedy tablica jest jeszcze zablokowana poleceniem `LOCK TABLE tab WRITE` z innego wątku.