# Living in a cluster jungle

or How to correctly query the DB in cluster environments

by Slava Kruglov

# About me

- Senior software engineer
- Entrepreneur (shardik.com)
- Speaker
- Trainer/Mentor



# Why?

- backend applications need a persistent storage
- common use-case: read – modify – persist
- clear how to synchronize in a single node
- clouds and microservices are everywhere
- clouds allow to scale up and down easily
- multiple nodes add race conditions -> unexpected data bugs

# What is it about?

- how an application works with a database
- how database configuration may impact the results
- recommendations on synchronizing in cluster
- common pitfalls in concurrent environments

# Databases

# Databases

- an organized collection of structured information
- relational database:
  - a database structured to recognize relations between stored items of information
- examples: PostgreSQL, Oracle Database, MySQL, MSSQL Server
- other databases like Clickhouse are not covered in the talk

| | id | created_date | last_modified_date | external_id |
|---|---|---|---|---|
| 1 | 2 | 2023-02-27 12:24:55.190514 | 2023-02-27 12:24:55.190515 | c5f86a52-418f-4300-929f-aaf28993b055 |

# Transactions

# Transactions

- a unit of work, performed within a database management system (or similar system) against a database, that is treated in a **coherent** and **reliable** way ***independent*** of other transactions
- example: a bank transfer

# Transactions

- a unit of work, performed within a database management system (or similar system) against a database, that is treated in a coherent and reliable way independent of other transactions

- usually a unit of work is a set of SQL queries (one or multiple)
- we want either all queries to succeed or all to abort: atomicity
- successful execution **commits** the transaction, unsuccessful **rollbacks** the transaction

# Transactions

- a unit of work, performed within a database management system (or similar system) against a database, that is treated in a coherent and reliable way independent of other transactions

- coherent means that the updated state is visible to other processes when a transaction commits
- also, the result of a transaction must be consistent

# Transactions

- a unit of work, performed within a database management system (or similar system) against a database, that is treated in a coherent and reliable way independent of other transactions

- reliable means the database will be able to recover even in case of a system failure in the middle of a transaction

# Transactions

- a unit of work, performed within a database management system (or similar system) against a database, that is treated in a coherent and reliable way ***independent*** of other transactions

- independent means there is a certain level of **isolation** of concurrent transactions

# Transactions

- requirements for a transaction: **ACID**
- ACID stands for:
  - atomic
  - consistent
  - **isolated**
  - durable

# Transaction example

- back to the bank transfer example
- the database should complete either all steps or none
- result of committed transaction must be immediately visible
- database failure must not leave unexpected charges or top ups
- the result of the transfer should be consistent

# Queries and transactions

- queries need to run inside a transaction
- even when the application code does not implicitly create a transaction
- when the application does not create a transaction, each query is executed in a separate transaction

# Transaction isolation

# Isolation

- three common types of concurrency side effects:
  - dirty read
  - non-repetable read
  - phantom read

# Dirty read

**Transaction 1**

```
update users.user_ set display_name =
‘trx1';
```

```
rollback;
```

**Transaction 2**

```
select * from users.user_;
```

```
update users.user_ set display_name =
concat(prev, 'trx2');

commit;
```

# Dirty read

**Transaction 1**

```
update users.user_ set display_name =
‘trx1';
```

```
rollback;
```

**Transaction 2**

```
select * from users.user_;
```




```
update users.user_ set display_name =
concat(prev, 'trx2');

commit;
```

# Dirty read

**Transaction 1**

```
update users.user_ set display_name =
‘trx1';
```

```
rollback;
```




**Transaction 2**

```
select * from users.user_;
```

```
update users.user_ set display_name =
concat(prev, 'trx2');

commit;
```

# Non-repeatable read

**Transaction 1**

```
select display_name from users.user_;
```

```
select display_name from users.user_;
```

**Transaction 2**

```
update users.user_ set display_name =
'trx2';
```

```
commit;
```

# Non-repeatable read

**Transaction 1**

```
select display_name from users.user_;
```




```
select display_name from users.user_;
```

**Transaction 2**

```
update users.user_ set display_name =
'trx2';

commit;
```

# Non-repeatable read

**Transaction 1**

```
select display_name from users.user_;
```




```
select display_name from users.user_;
```




**Transaction 2**

```
update users.user_ set display_name =
'trx2';

commit;
```

# Phantom read

**Transaction 1**

```
select * from users.user_;
```

```
select * from users.user_;
```

**Transaction 2**

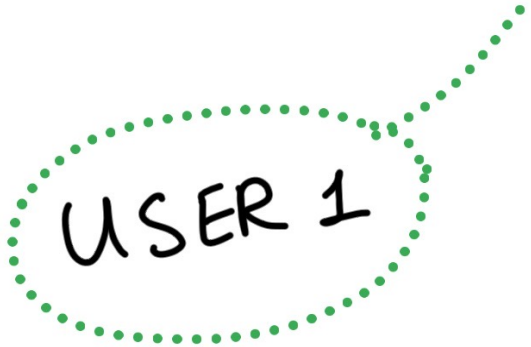
```
insert into users.user_ values (2,
'trx2');
```

```
commit;
```

# Phantom read

**Transaction 1**

```
select * from users.user_;
```




```
select * from users.user_;
```

**Transaction 2**

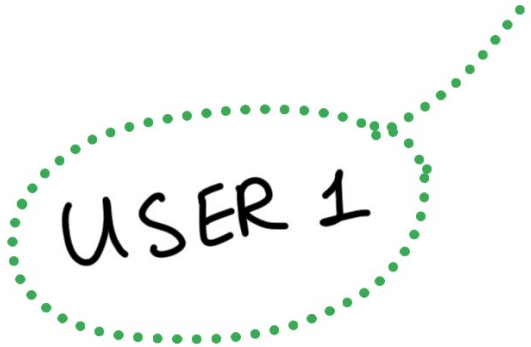
```
insert into users.user_ values (2,
'trx2');

commit;
```

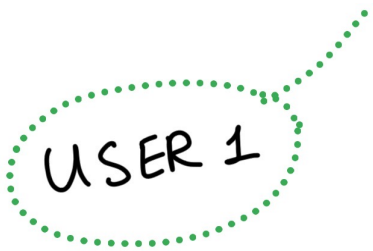
# Phantom read

**Transaction 1**

```
select * from users.user_;
```




```
select * from users.user_;
```




**Transaction 2**

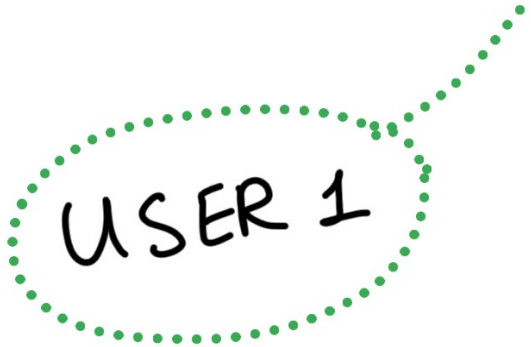
```
insert into users.user_ values (2,
'trx2');

commit;
```

# Phantom read

**Transaction 1**

```
select * from users.user_;
```




```
select * from users.user_;
```




**Transaction 2**

```
insert into users.user_ values (2,
'trx2');

commit;
```

# Isolation level

| Isolation level | Dirty read | Non-repeatable read | Phantom read |
|---|---|---|---|
| Read uncommitted | yes | yes | yes |
| Read committed | no | yes | yes |
| Repeatable read | no | no | yes |
| Serializable | no | no | no |

# Isolation level

- the lower the isolation level, the more performant the database
- however, more side effects
- the higher the isolation level, the less concurrency

- default isolation level for many databases is *READ_COMMITTED*

# Lost update

- there is a forth side-effect that usually goes together with other three
- there are usually two different scenarios that refer to lost update

- `update users.user_ set display_name = concat(display_name, ‘trx1’);`

- `update users.user_ set display_name = ‘trx1';`

# Lost update

**Transaction 1**

```
update users.user_ set display_name =
'trx1';
```

```
commit;
```

**Transaction 2**

```
update users.user_ set display_name =
'trx2';
```

```
commit;
```

# Lost update

**Transaction 1**

```
update users.user_ set display_name =
'trx1';
```

```
commit;
```



**Transaction 2**

```
update users.user_ set display_name =
'trx2';
```

```
commit;
```

# Lost update

**Transaction 1**

```
update users.user_ set display_name =
concat(display_name, 'trx1');
```

```
commit;
```

**Transaction 2**

```
update users.user_ set display_name =
concat(display_name, 'trx2');
```

```
commit;
```

# Lost update

**Transaction 1**

```
update users.user_ set display_name =
concat(display_name, 'trx1');
```

```
commit;
```

**Transaction 2**

```
update users.user_ set display_name =
concat(display_name, 'trx2');
```

```
commit;
```



# Locks

# Locks

- multiple nodes in cluster can only synchronize on the database
- on the database level synchronization is possible with locks
- there are two lock types:
  - optimistic locking
  - pessimistic locking

# Optimistic locking

- the logic is on the application side
- the idea is to have a version column (a number or a timestamp)
- the application needs to check the state of the version column before a commit

# Optimistic locking

- the flow:
  - read the version column value
  - do work
  - read the version column again
  - if version is the same -> update the version and commit
  - if the version is different -> abort

# Optimistic locking

- isolation level matters
- optimistic locking needs a recovery mechanism

# Pessimistic locking

- there are two main types:
  - shared (read) lock
  - exclusive (write) lock

# Pessimistic locking

- there are two main types:
  - shared (read) lock == `SELECT FOR SHARE`(PostgreSQL)/`LOCK IN SHARE MODE`(MySQL)
  - exclusive (write) lock == `SELECT FOR UPDATE`

# Pessimistic locking

- a cron job in a cluster environment is a use-case for locking
- nodes need to agree who runs the job
- first job run is different from next ones
- one of the solutions -> general locks table

- another use-case is a sequential ID generator

# Pessimistic locking

- pessimistic locks are not a silver bullet
- check resulting SQL queries

```
> select id from users.chunk where status = ‘ACTIVE’;
> select * from users.chunk where id = 1 for update;
```

# Deadlocks

- deadlock may happen when two transactions acquire locks on the same resources
- usually databases can handle deadlocks
- database may abort a deadlocked transaction randomly

# Deadlock example

**Transaction 1**

```
update users.user_ set display_name =
'trx1' where id = 1;
```

```
update users.user_ set display_name =
'trx1' where id = 2;
```

**Transaction 2**

```
update users.user_ set display_name =
'trx2' where id = 2;
```

```
update users.user_ set display_name =
'trx2' where id = 1;
```

# Deadlock example

**Transaction 1**

```
update users.user_ set display_name =
'trx1' where id = 1;
```

```
update users.user_ set display_name =
'trx1' where id = 2;
```




**Transaction 2**

```
update users.user_ set display_name =
'trx2' where id = 2;
```

```
update users.user_ set display_name =
'trx2' where id = 1;
```

# Deadlock example

- deadlock can also occur in inner transactions
- `@Transactional(Transactional.TxType.REQUIRES_NEW)`
- inner transactions do not share locks

# Database migrations

- Flyway/Liquibase and similar tools are great for DB migrations
- however, each node will try to run the migration
- rolling update deployment may help
- be aware of long-running migrations and health check timeouts

# Summary

# Summary

- keep in mind concurrency when accessing a database
- manage transactions
- keep transactions short if possible
- check resulting SQL queries

# Thank you

- twitter.com/vv_kruglov

