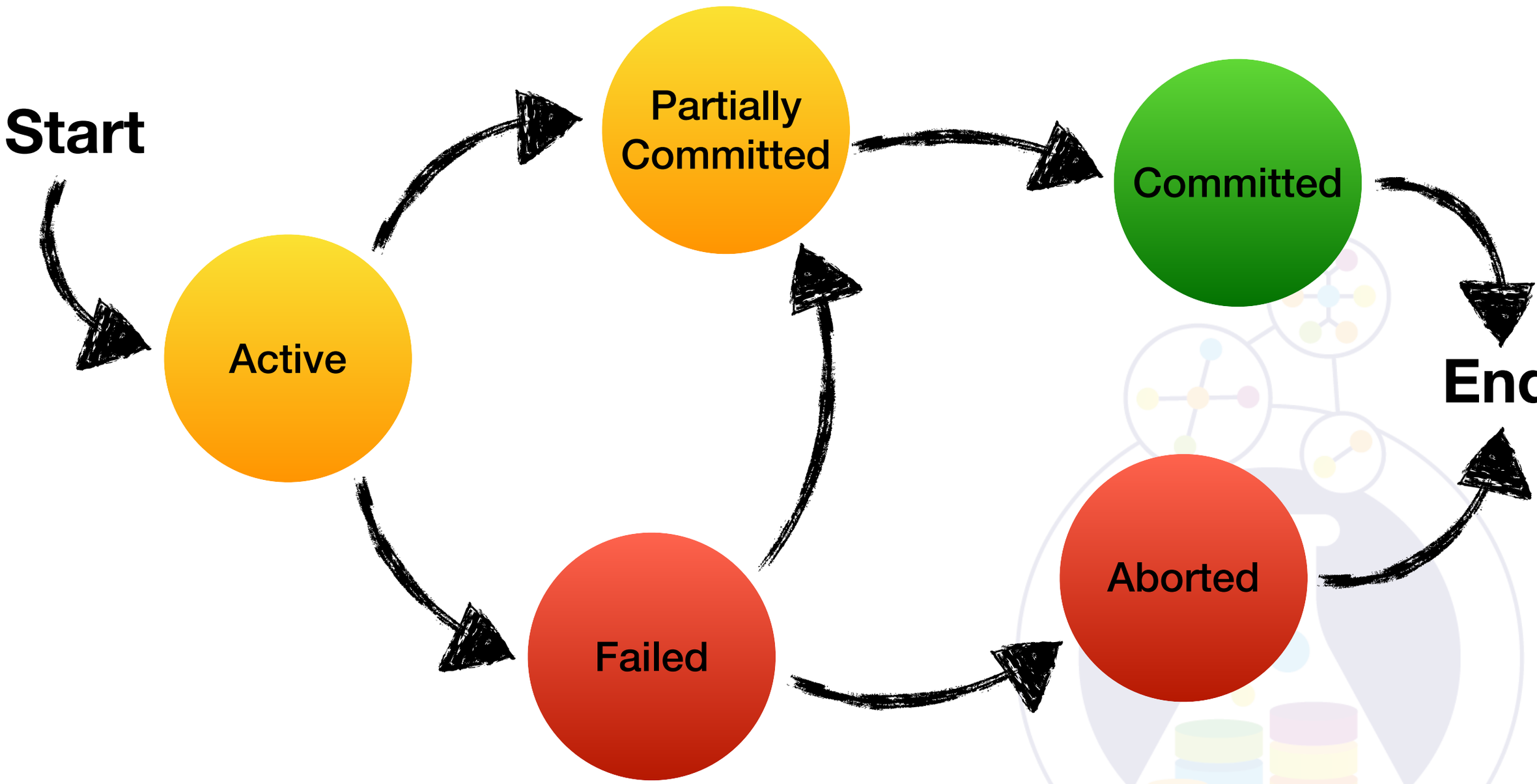
datalab

data is everywhere, value is hidden

Relational Databases

Lecturer: Азат Якупов (Azat Yakupov)
https://datalaboratory.one



transactional block

set of changes

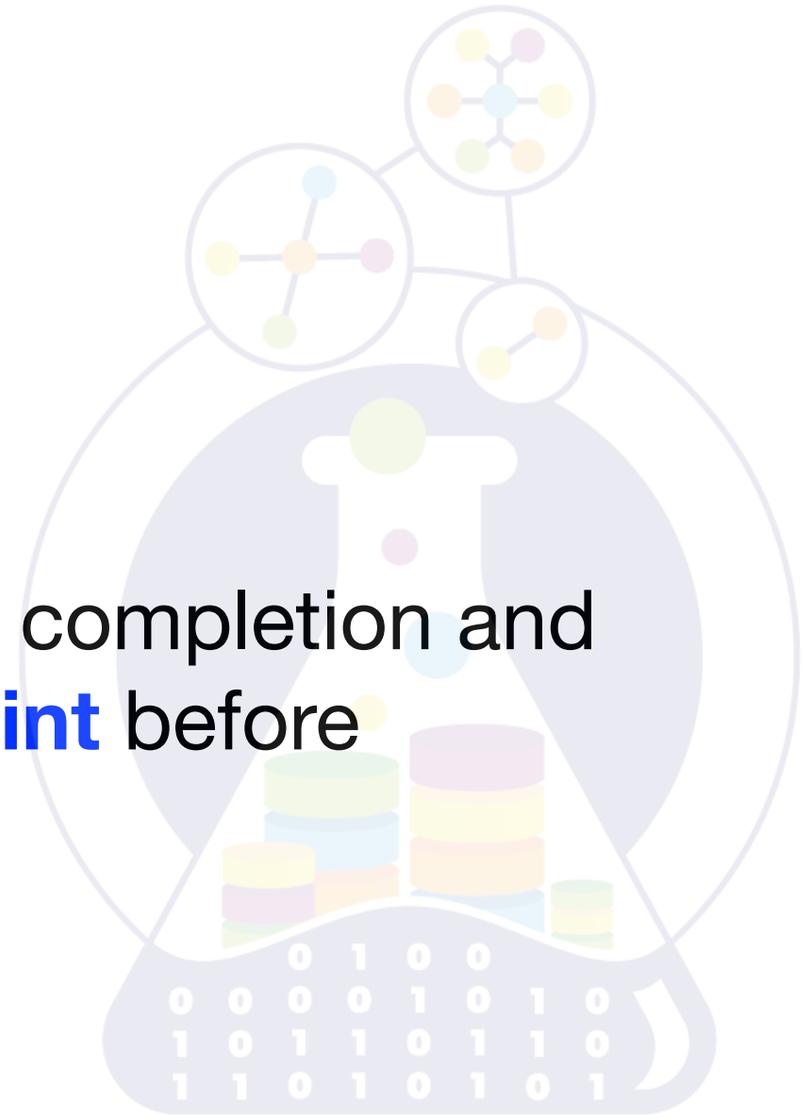
```
BEGIN TRANSACTION;

UPDATE account 100 {balance := balance - 100} ;
IF <exception> THEN GO TO UNDO ; END IF;

UPDATE account 200 {balance := balance + 100} ;
IF <exception> THEN GO TO UNDO ; END IF;

COMMIT;
RETURN ;

UNDO:
    ROLLBACK;
```

- **COMMIT**

  signalizes about successful completion and fixes a changes from transactional block
  ~ syncpoint

- **ROLLBACK**

  signalizes about unsuccessful completion and restores to a previous **syncpoint** before transactional block



Start Session

COMMIT

ROLLBACK

ROLLBACK

COMMIT

COMMIT

End Session

time scale

Start Session

End Session

Start Session

End Session

Start Session

End Session

Start Session

End Session

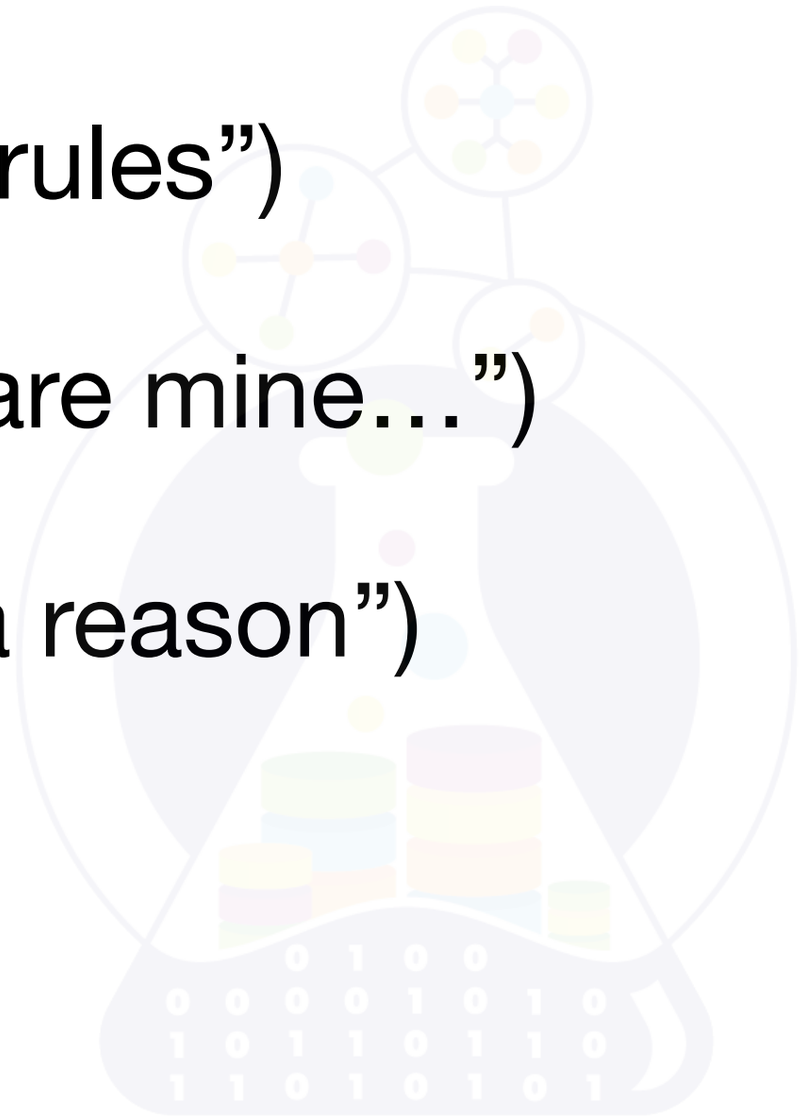
time scale

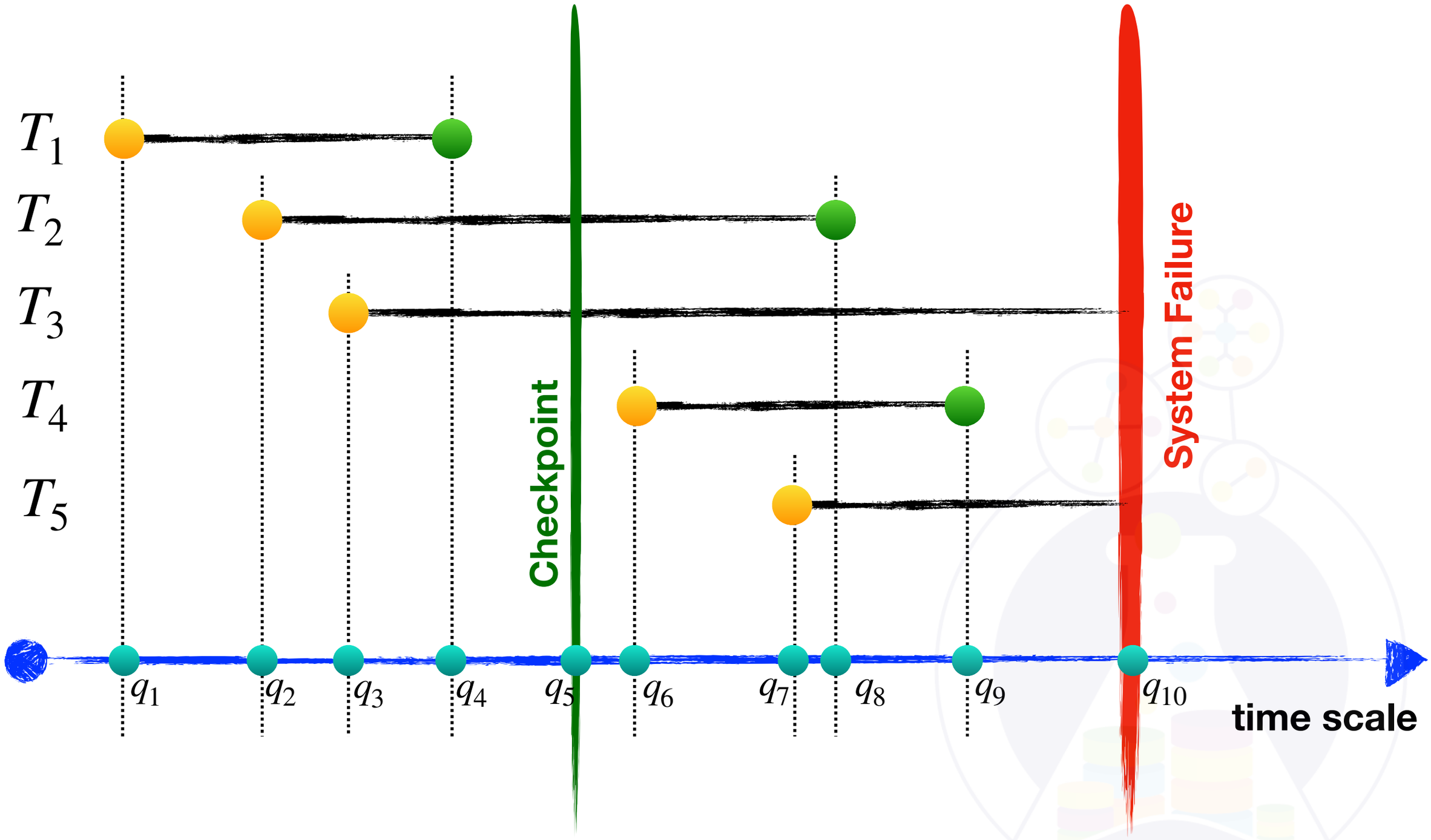
A tomicity (“all or nothing”)

C onsistency (“follow the rules”)

I solation (“my changes are mine…”)

D urability (“death is not a reason”)





DB log - journal with all Database events

q_1 : *BEGIN TRANSACTION* T_1
q_2 : *BEGIN TRANSACTION* T_2
q_3 : *BEGIN TRANSACTION* T_3
q_4 : *COMMIT* T_1
q_5 : *CHECKPOINT*

last checkpoint

T_2, T_3 are in the progress

T_4

T_5

q_6 : *BEGIN TRANSACTION* T_4
q_7 : *BEGIN TRANSACTION* T_5
q_8 : *COMMIT* T_2
q_9 : *COMMIT* T_4
q_{10} : Ø

When DB is returned back from **failure** then

- 2 lists of type of transactions will be created
  UNDO (list of transactions have to be cancelled)
  REDO (list of transactions have to be reapplied)

- Initial **UNDO** $=\{T_2, T_3, T_4, T_5\}$ by using **DB Log**
  $\{T_2, T_3\}$ have been in a progress during checkpoint
  $\{T_4, T_5\}$ have been happened after checkpoint

- If **COMMIT** has been detected in **DB Log** after checkpoint then **UNDO** $=\{T_3, T_5\}$ and **REDO** $= \{T_2, T_4\}$

UNDO = $\{T_3, T_5\}$

q_1 : *BEGIN TRANSACTION* T_1

q_2 : *BEGIN TRANSACTION* T_2

q_3 : *BEGIN TRANSACTION* T_3

q_4 : *COMMIT* T_1

q_5 : *CHECKPOINT*

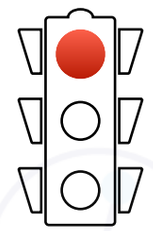
q_6 : *BEGIN TRANSACTION* T_4

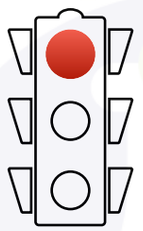
q_7 : *BEGIN TRANSACTION* T_5

q_8 : *COMMIT* T_2

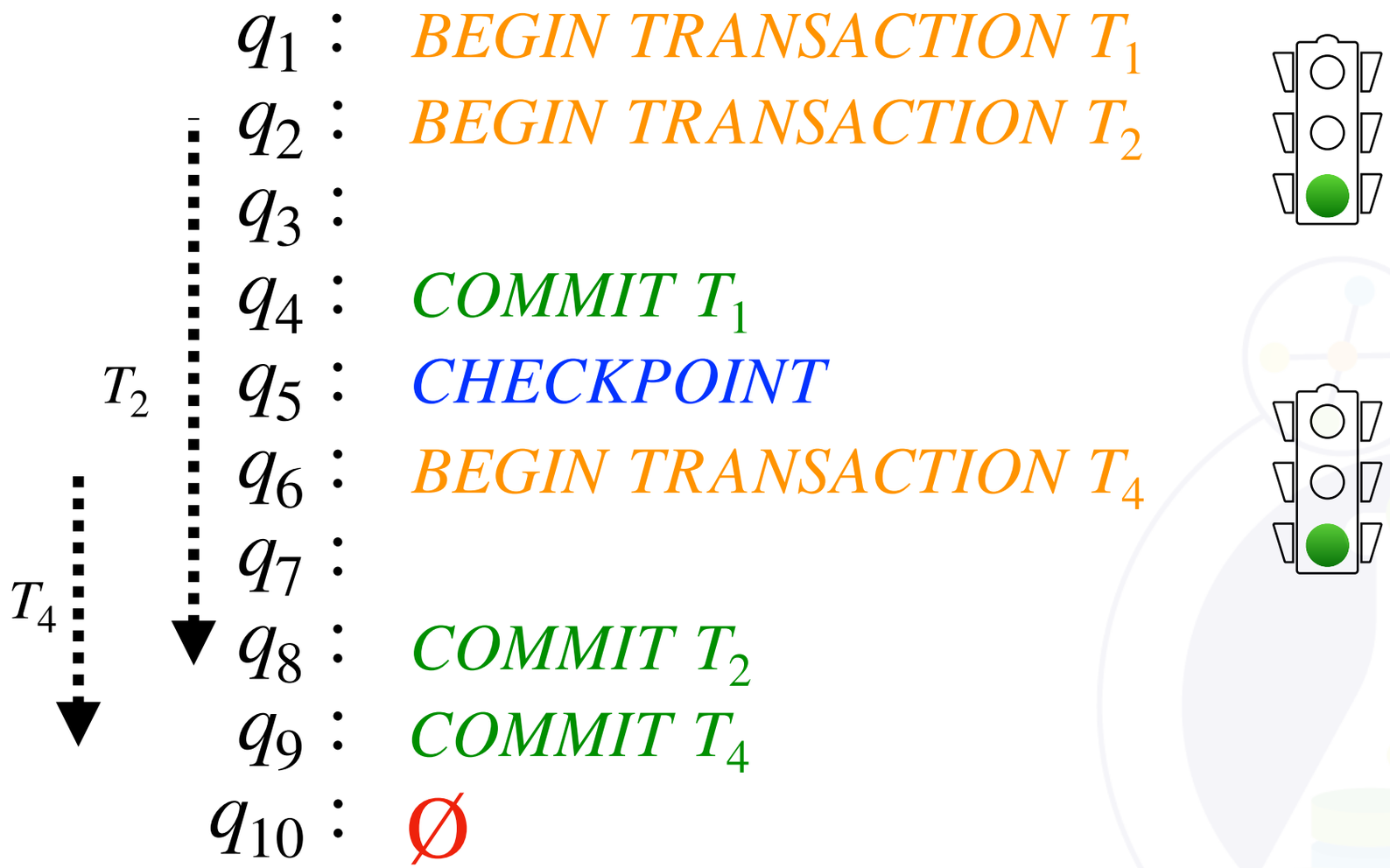
q_9 : *COMMIT* T_4

q_{10} : $\emptyset$





REDO = $\{T_2, T_4\}$

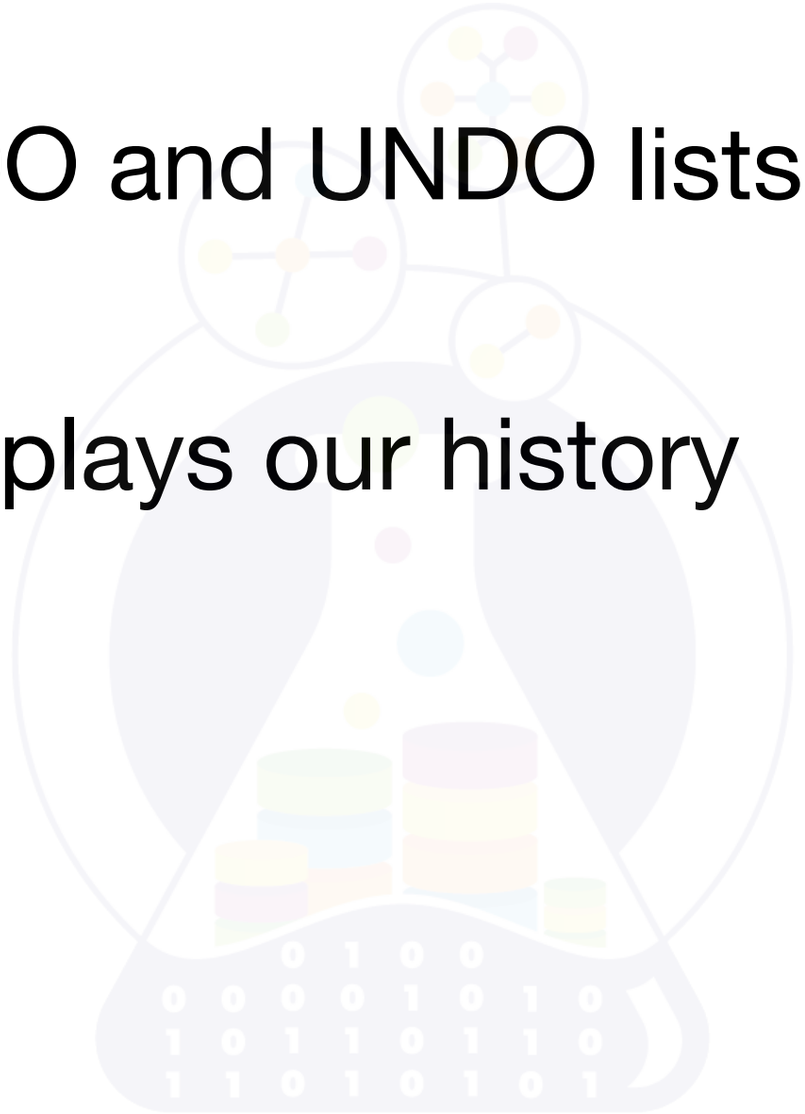


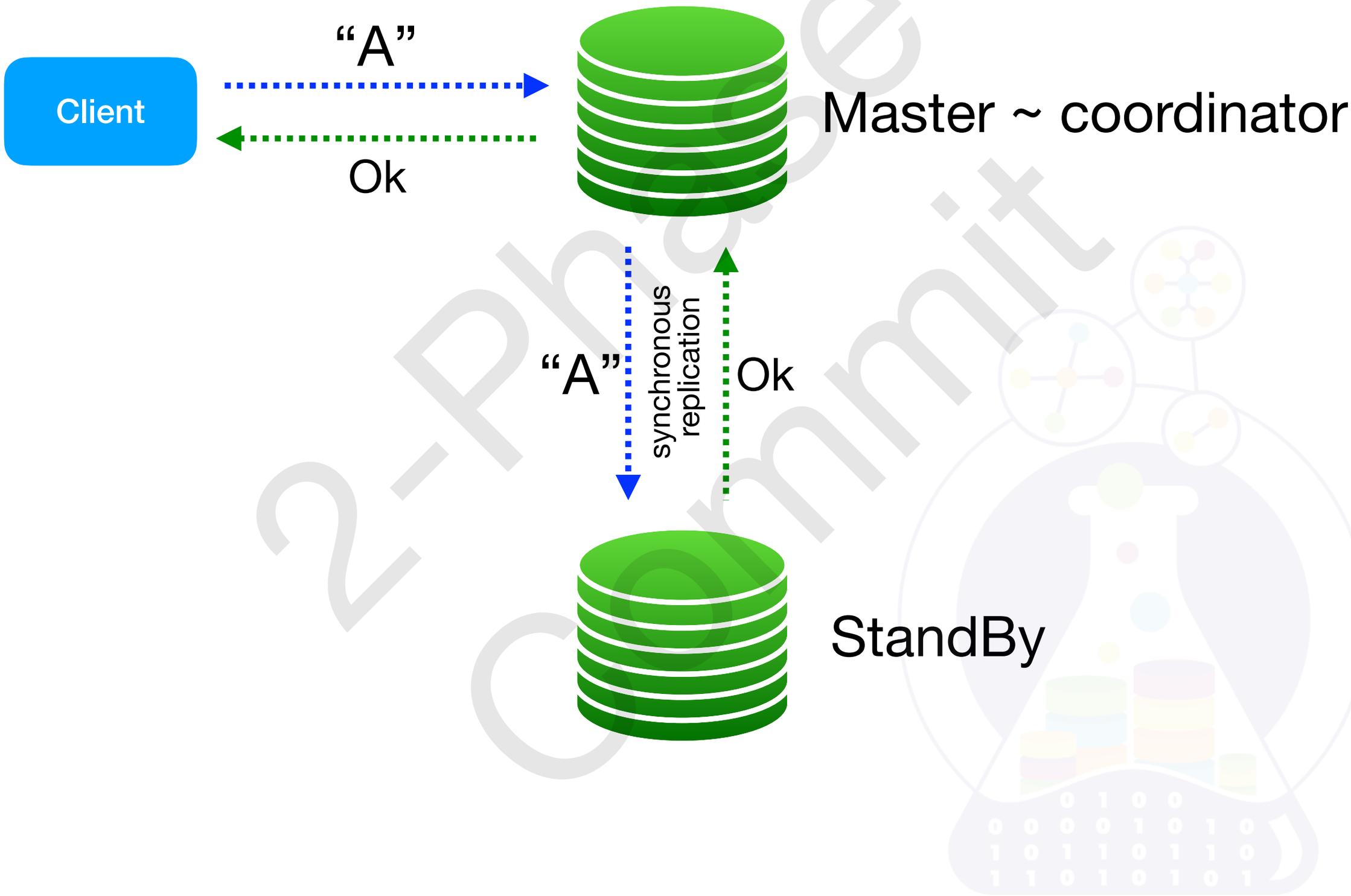
Content of **xlog** / **wal** file of **PostgreSQL**

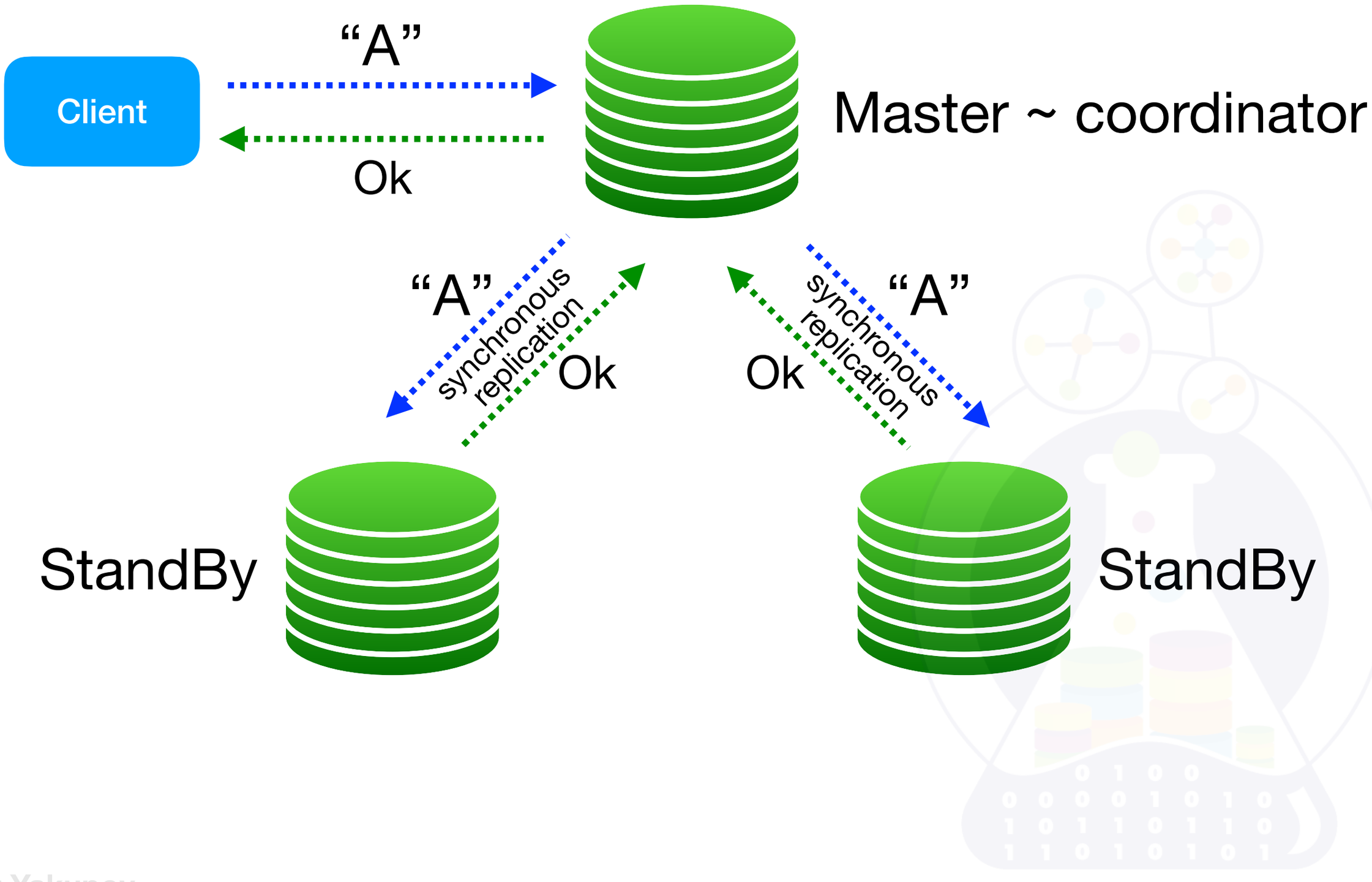
```
...
rmgr: Heap        len (rec/tot):     63/    63, tx:
  1580, lsn: 0/04EB6BC0, prev 0/04EB6B88, desc: INSERT+INIT off 1, blkref #0: rel 1663/16384/57376 blk 0
rmgr: Transaction len (rec/tot):     46/    46, tx:
  1580, lsn: 0/04EB6C00, prev 0/04EB6BC0, desc: COMMIT 2018-06-04 15:54:49.362550 CDT
rmgr: Standby     len (rec/tot):     54/    54, tx:
  0, lsn: 0/04EB6C30, prev 0/04EB6C00, desc: RUNNING_XACTS nextXid 1581 latestCompletedXid 1579
oldestRunningXid
  1580; 1 xacts: 1580
rmgr: Heap2       len (rec/tot):     59/    59, tx:
  0, lsn: 0/04EB6C68, prev 0/04EB6C30, desc: VISIBLE cutoff xid 1580 flags 1, blkref #0: rel
1663/16384/57376 fork vm blk 0, blkref #1: rel 1663/16384/57376 blk 0
rmgr: Heap        len (rec/tot):    233/   233, tx:
  0, lsn: 0/04EB6CA8, prev 0/04EB6C68, desc: INPLACE off 75, blkref #0: rel 1663/16384/32918 blk 13
rmgr: Standby     len (rec/tot):     90/    90, tx:
  0, lsn: 0/04EB6D98, prev 0/04EB6CA8, desc: INVALIDATIONS ; inval msgs: catcache 50 catcache 49
relcache 40962
rmgr: Standby     len (rec/tot):     50/    50, tx:
  0, lsn: 0/04EB6DF8, prev 0/04EB6D98, desc: RUNNING_XACTS nextXid 1581 latestCompletedXid 1580
oldestRunningXid 1581
...
```

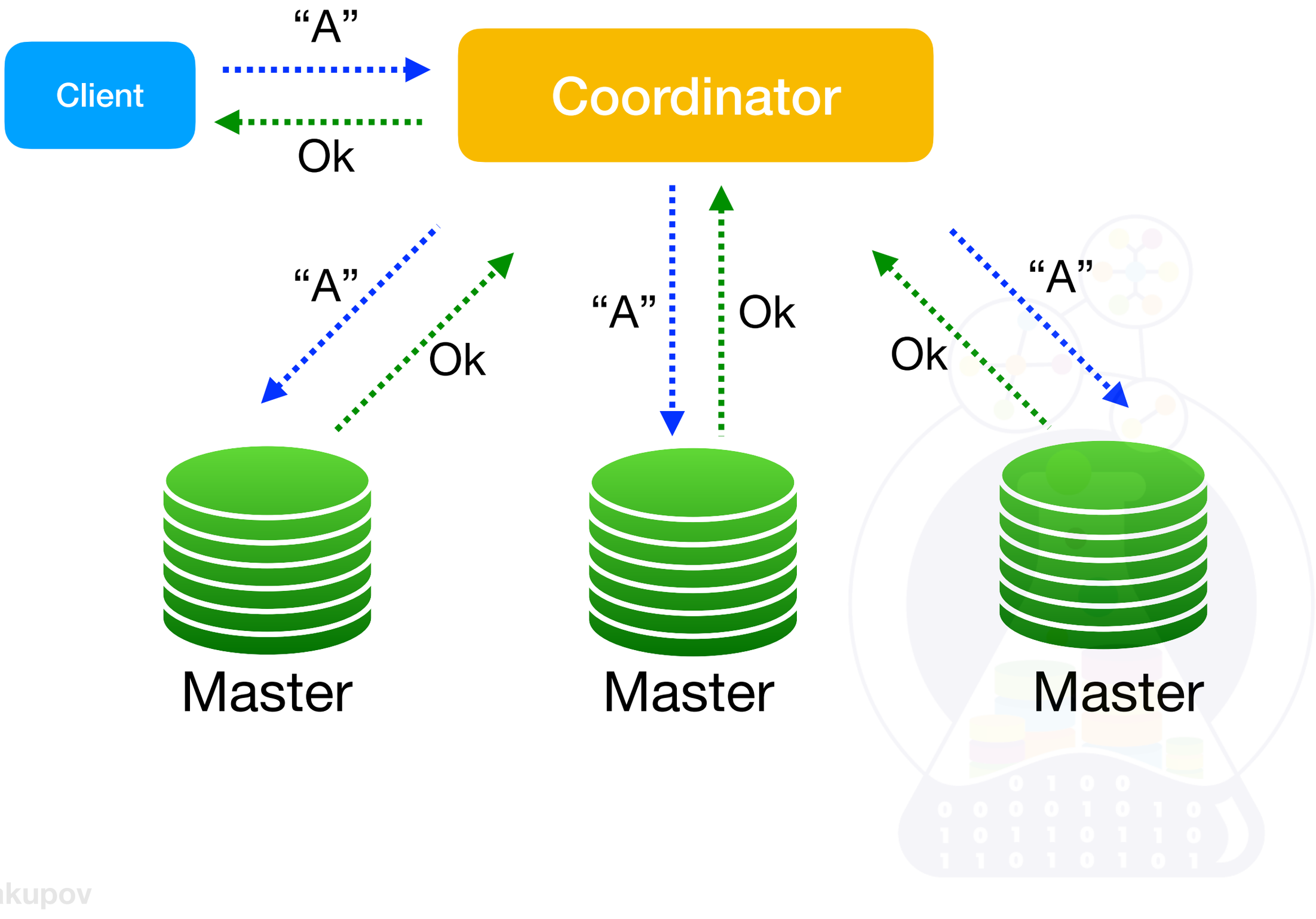
Algorithm for **R**ecovery and **I**solation **E**xploiting **S**emantics

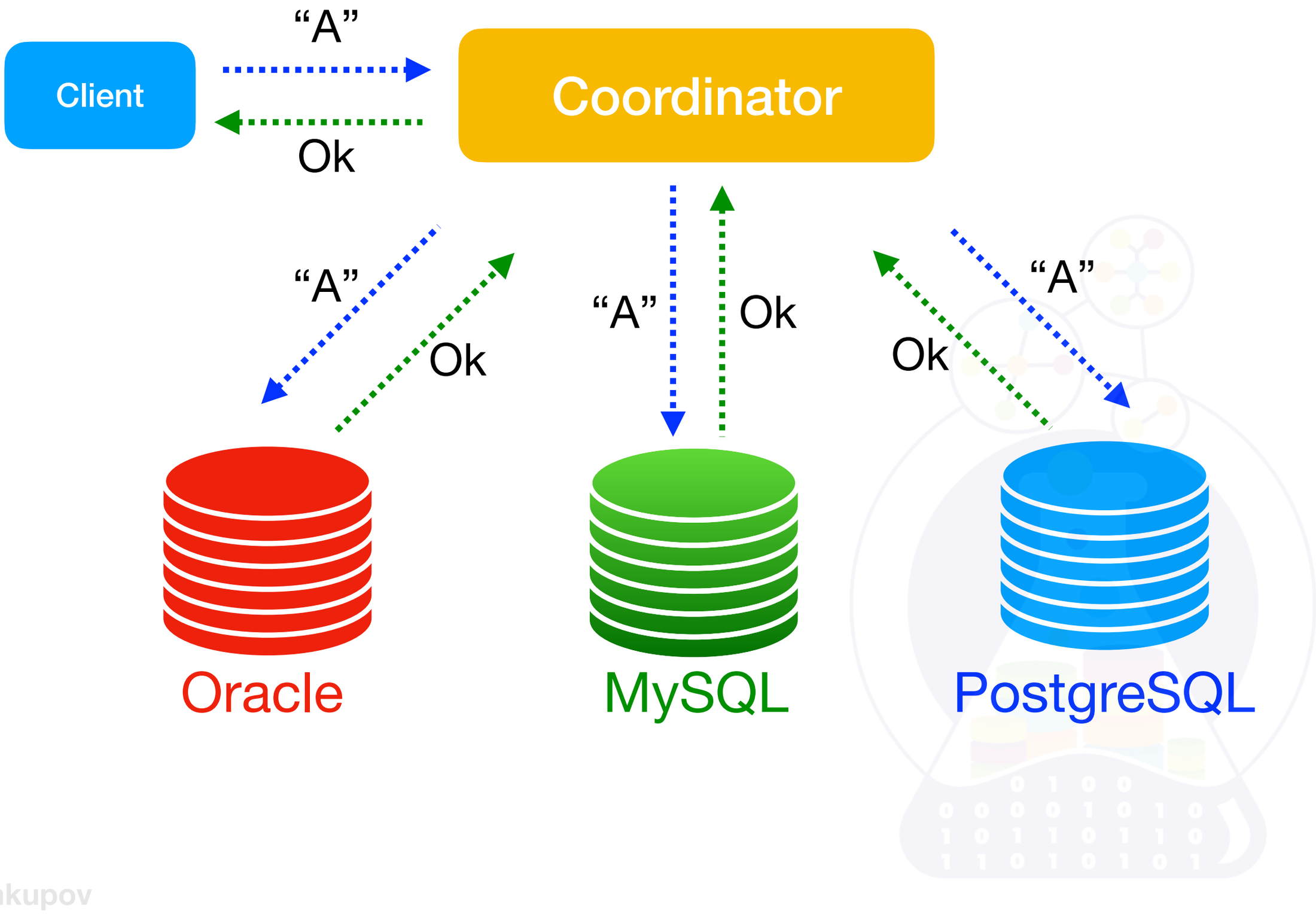
- **Analysis State:** makes REDO and UNDO lists
- **REDO Recovery State** ~ replays our history
- **UNDO Recovery State**

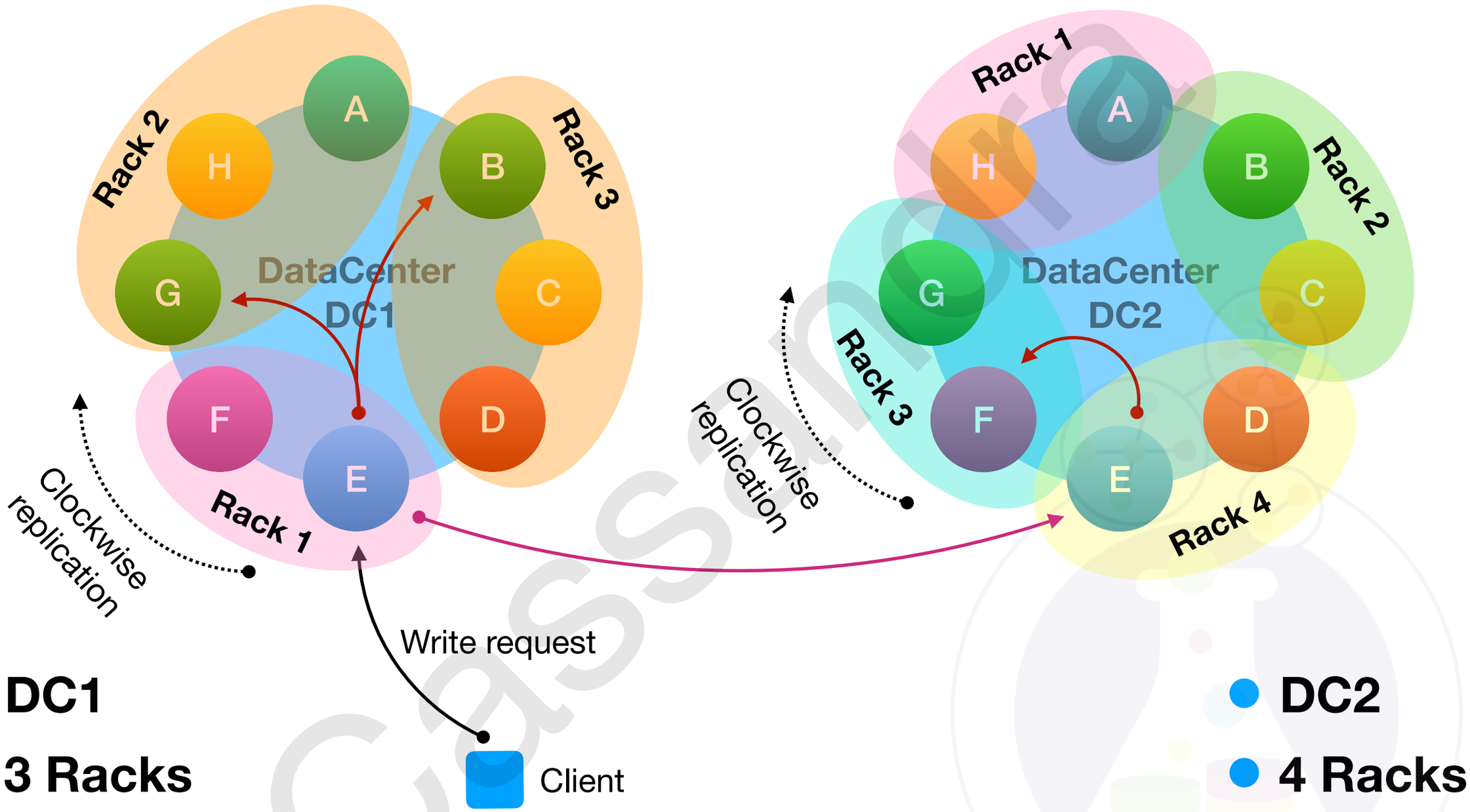








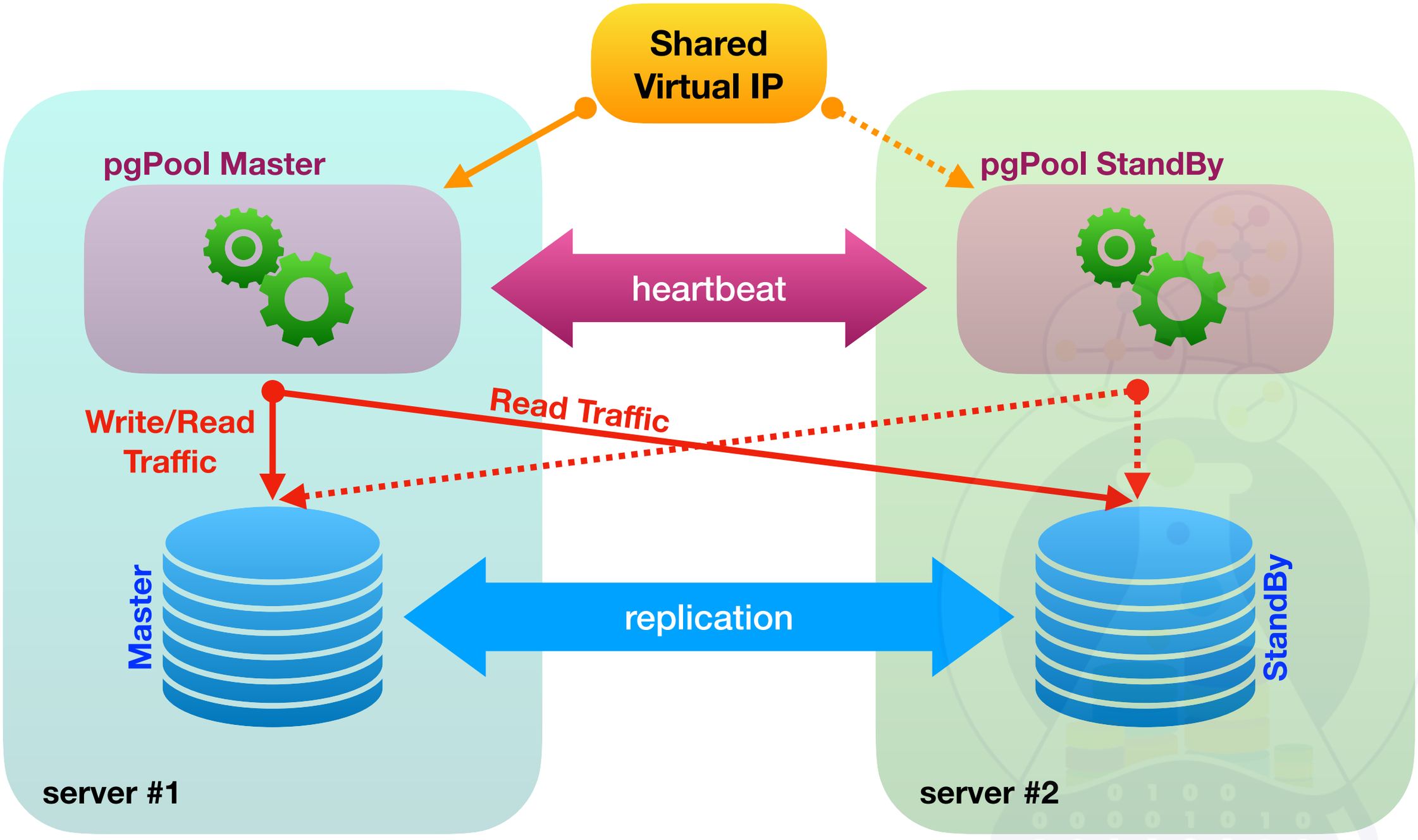




- DC1
- 3 Racks
- RF = 3

- DC2
- 4 Racks
- RF = 2

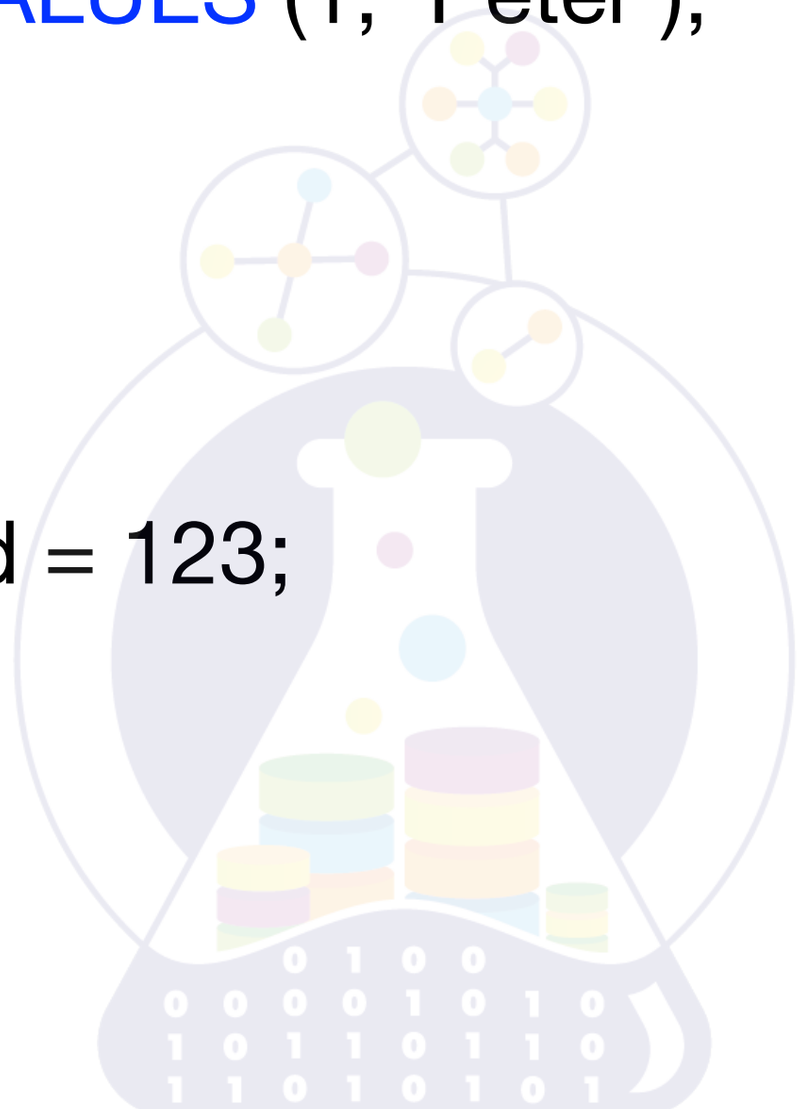
PostgreSQL HA topology



session

```
UPDATE employee SET phone = '12345' WHERE id =100;
INSERT INTO manager(id, name) VALUES (1, 'Peter');
COMMIT;
```

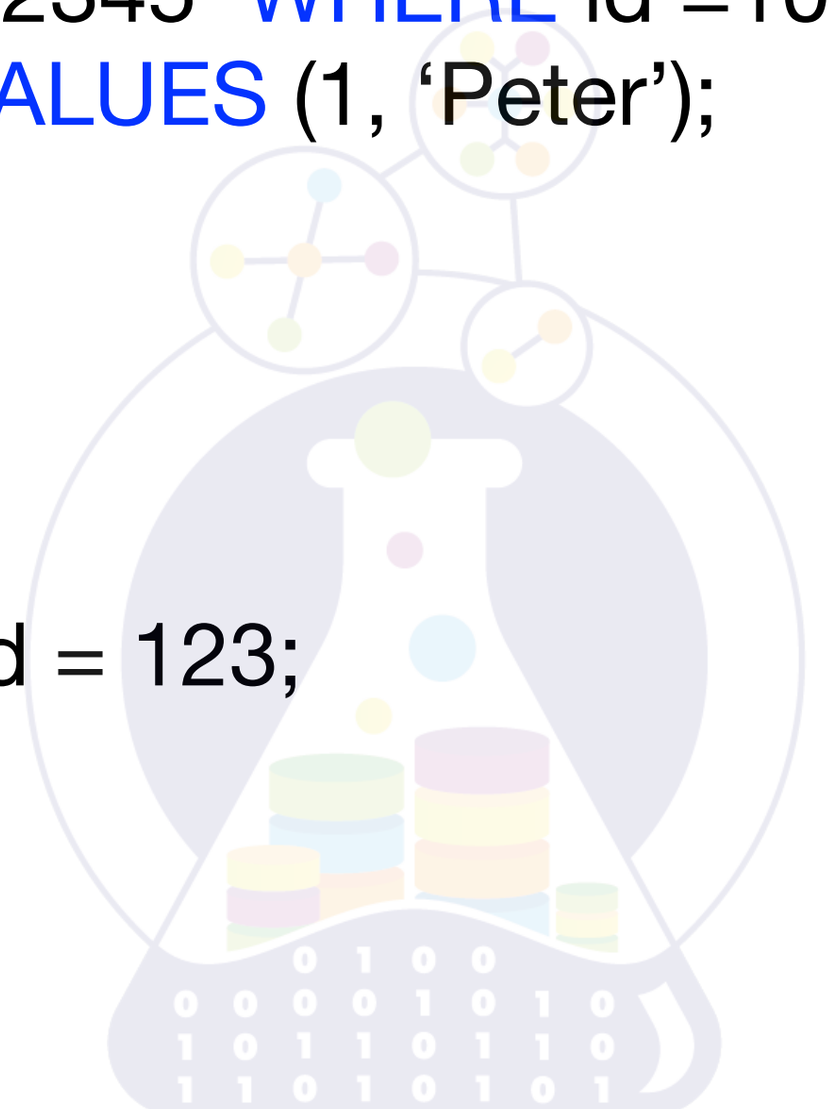
```
DELETE FROM manager WHERE id = 123;
ROLLBACK;
```



session

```
BEGIN TRANSACTION;
UPDATE employee SET phone = ‘12345’ WHERE id =100;
INSERT INTO manager(id, name) VALUES (1, ‘Peter’);
COMMIT;
```

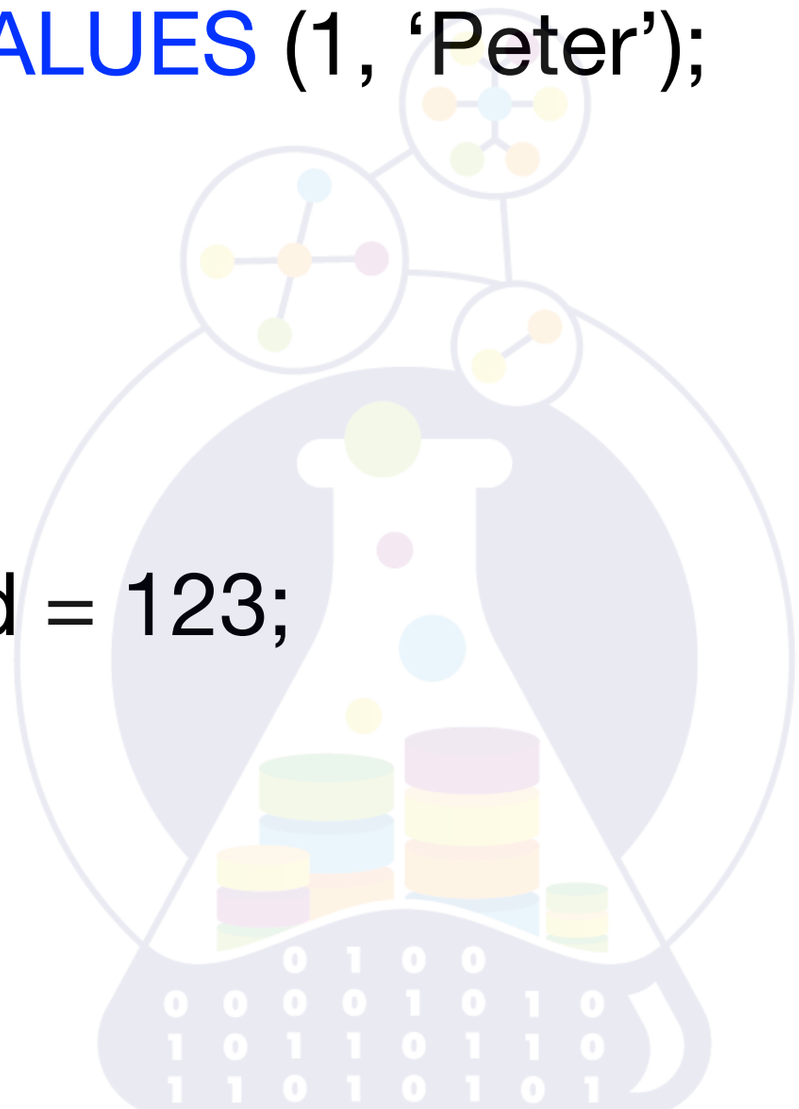
```
BEGIN;
DELETE FROM manager WHERE id = 123;
ROLLBACK;
```



```
START TRANSACTION;
UPDATE employee SET phone = ‘12345’ WHERE id =100;
INSERT INTO manager(id, name) VALUES (1, ‘Peter’);
COMMIT;
```

session

```
BEGIN;
DELETE FROM manager WHERE id = 123;
ROLLBACK;
```



ORACLE

session

```
UPDATE employee SET phone = '12345' WHERE id =100;

CREATE TABLE test (id NUMBER);

INSERT INTO manager(id, name) VALUES (1, 'Peter');
COMMIT;
```

PostgreSQL

session

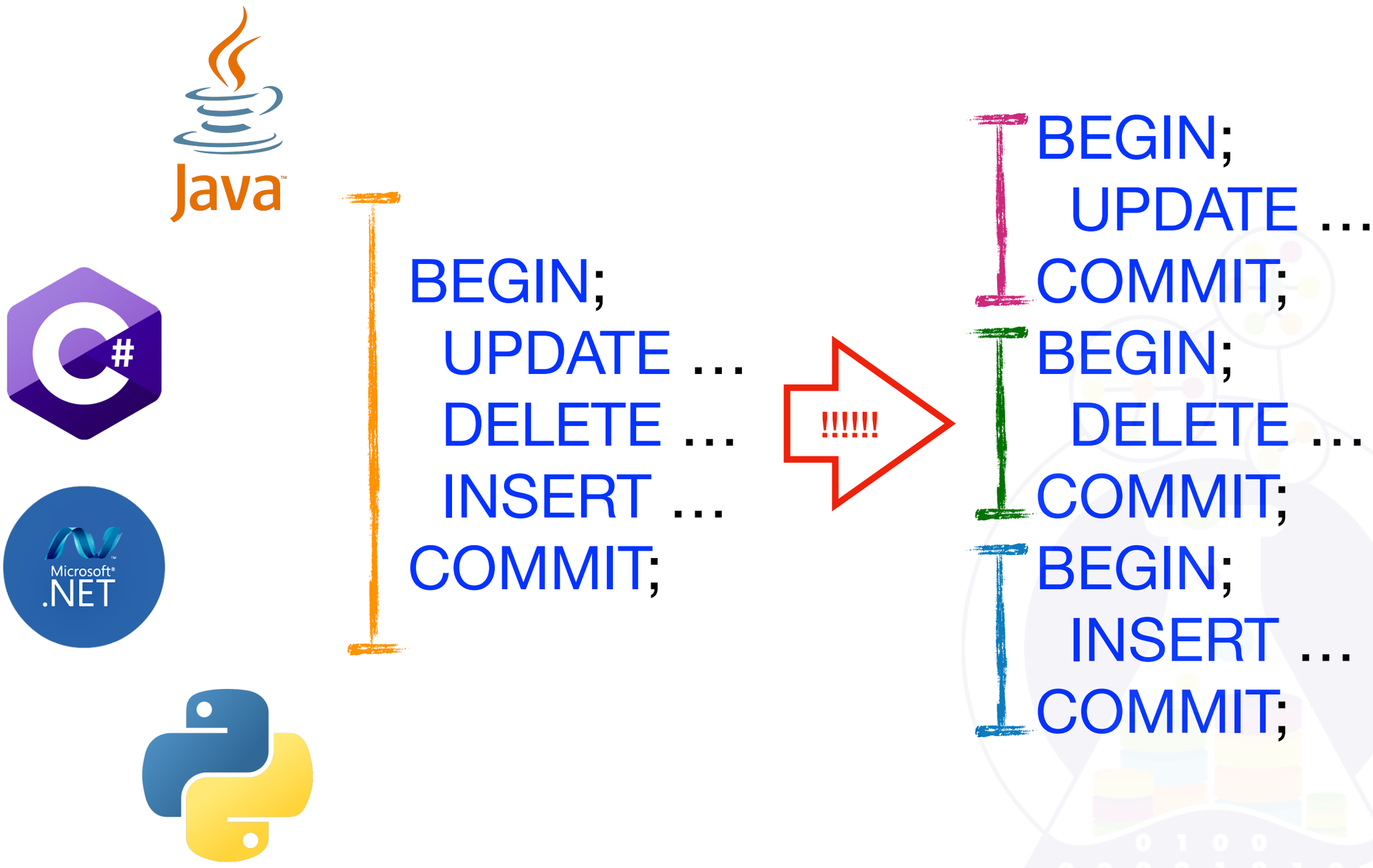
```
BEGIN TRANSACTION;
UPDATE employee SET phone = ‘12345’ WHERE id =100;

CREATE TABLE test (id NUMBER);

INSERT INTO manager(id, name) VALUES (1, ‘Peter’);
COMMIT;
```

UPDATE employee SET phone = ‘12345’ WHERE id =100;

CREATE TABLE test (id NUMBER);

INSERT INTO manager(id, name) VALUES (1, ‘Peter’);
COMMIT;

Autocommit


Java
C#
Microsoft .NET
BEGIN;
UPDATE …
DELETE …
INSERT …
COMMIT;
!!!!!
BEGIN;
UPDATE …
COMMIT;
BEGIN;
DELETE …
COMMIT;
BEGIN;
INSERT …
COMMIT;

Start Transaction

UPDATE

INSERT

SP #1

I

DELETE

FAILED

ROLLBACK TO SP #1

II

INSERT

SP #2

COMMIT

End Transaction

Azat Yakupov

Start Transaction

UPDATE

INSERT

SP #1

I

DELETE

FAILED

ROLLBACK TO SP #1

II

INSERT

SP #2

End Transaction

III

ROLLBACK

Azat Yakupov

Oracle savepoint

```
BEGIN
   INSERT …
   SAVEPOINT do_insert;

   UPDATE …
   DELETE …

   COMMIT;
EXCEPTION WHEN OTHERS THEN
   ROLLBACK do_insert;
END;
```

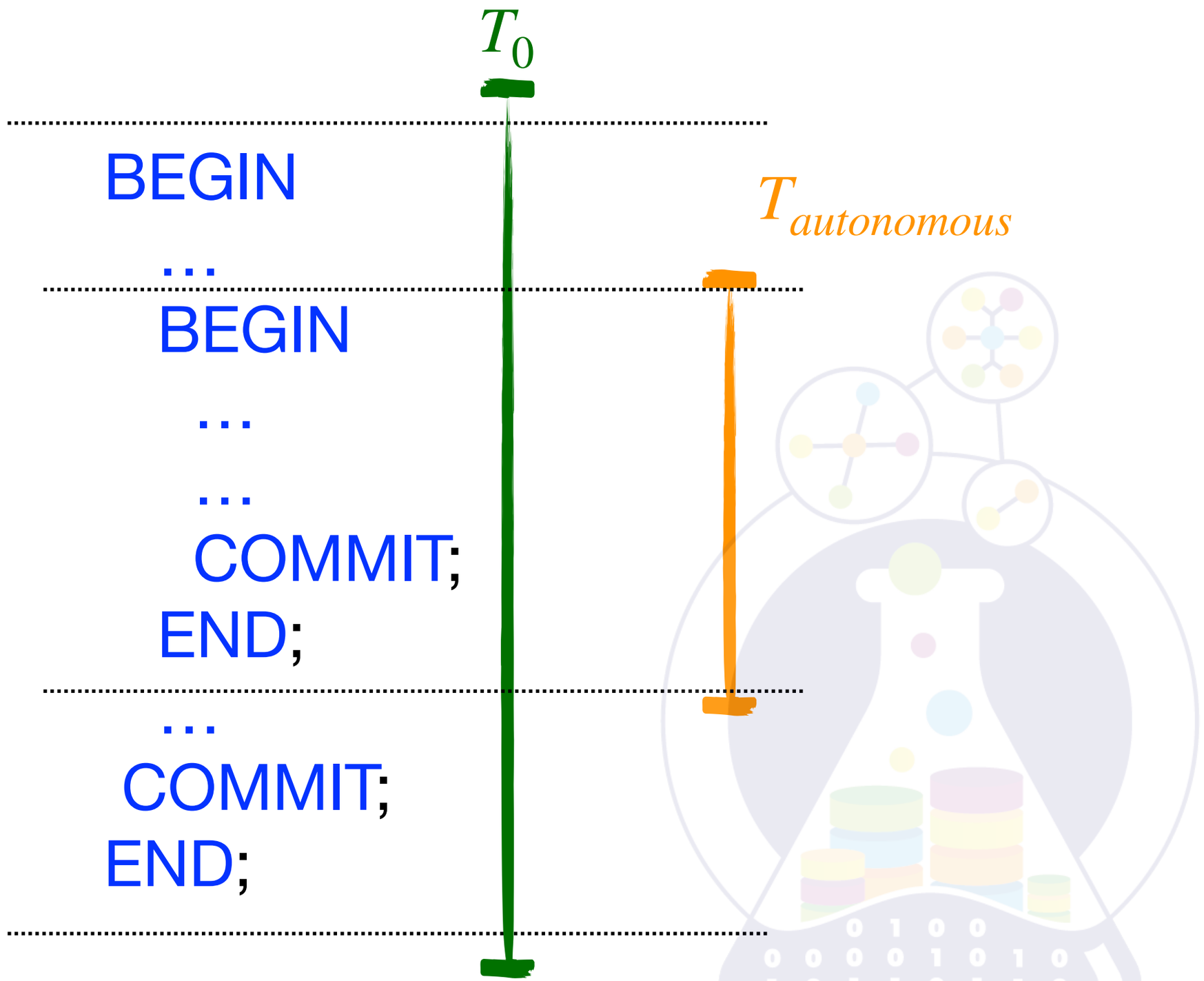
PostgreSQL savepoint

```
BEGIN;
   INSERT …
   SAVEPOINT do_insert;
   UPDATE …
   ROLLBACK TO do_insert;
   DELETE …

   RELEASE do_insert;

COMMIT;
```

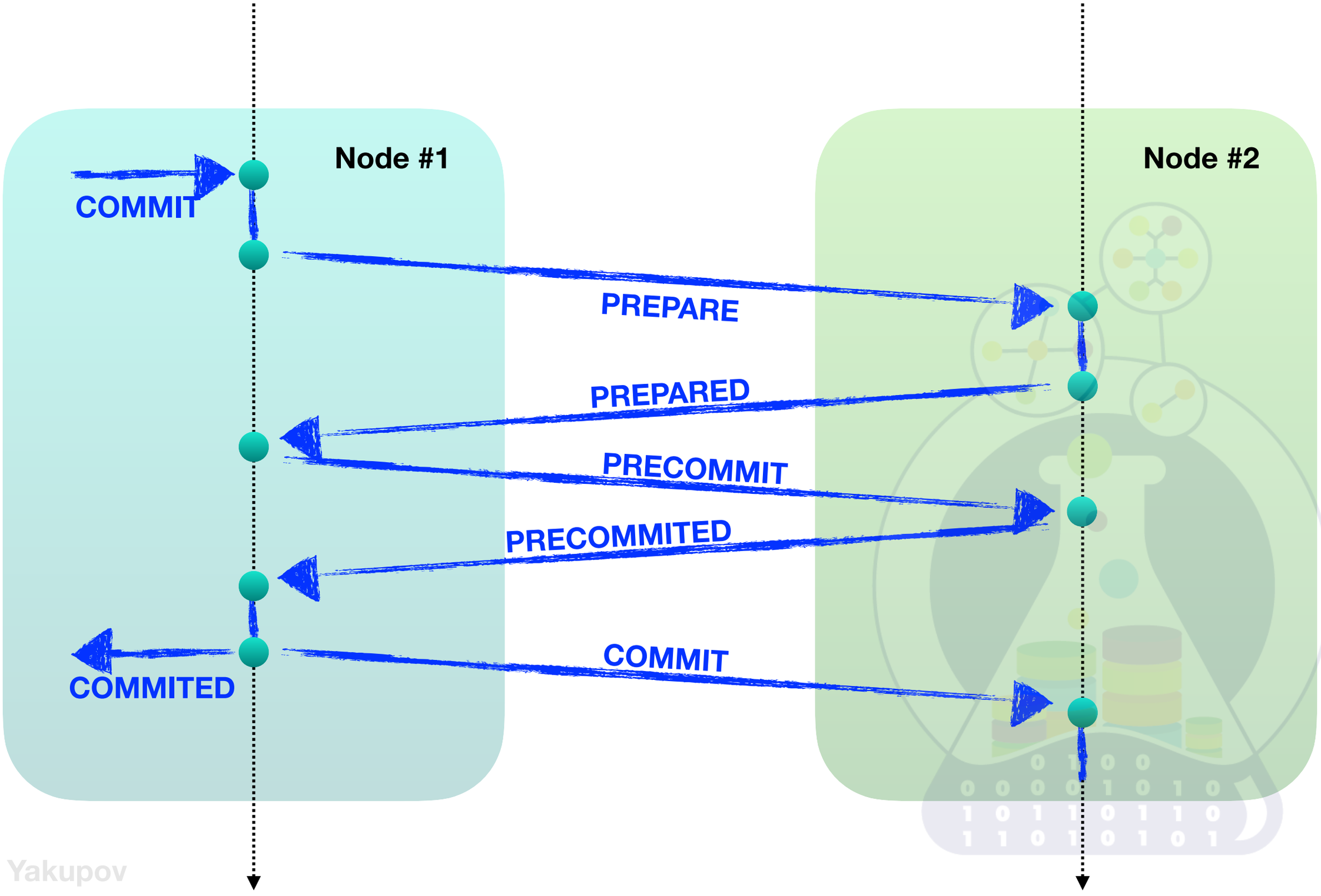
MySQL savepoint

```
BEGIN;
   INSERT …
   SAVEPOINT do_insert;
   UPDATE …
   ROLLBACK TO do_insert;
   DELETE …

   RELEASE SAVEPOINT do_insert;

COMMIT;
```

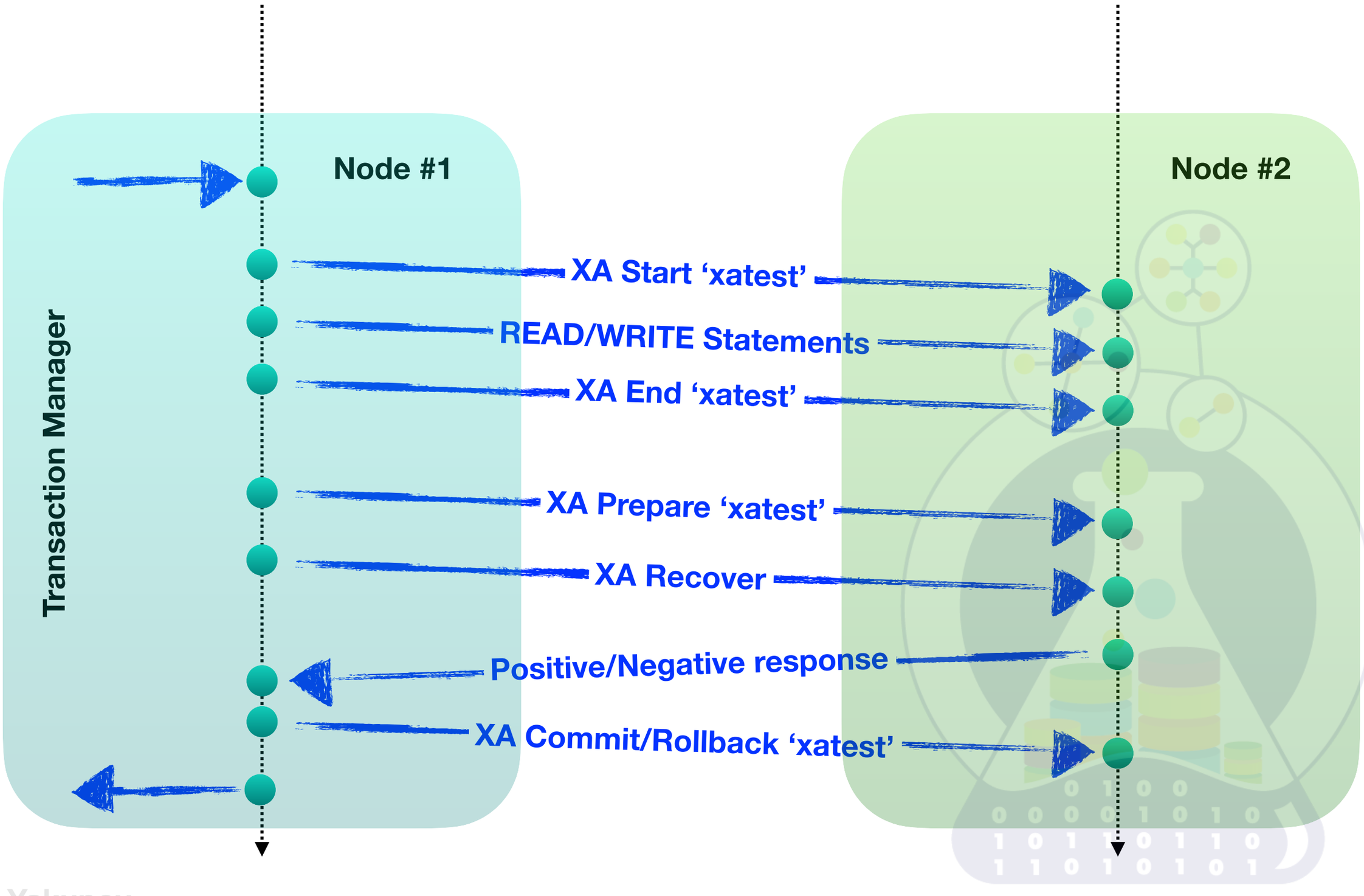
Oracle autonomous transaction



PostgreSQL prepared transaction



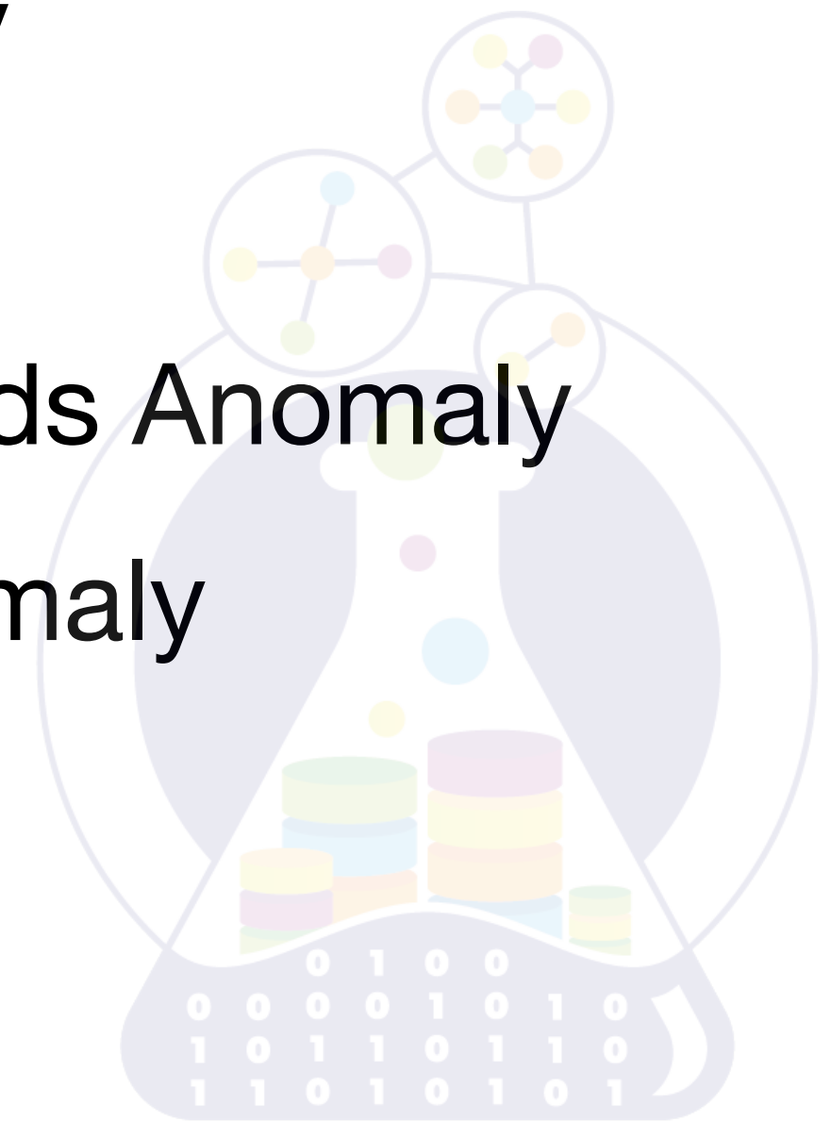
MySQL XA transaction



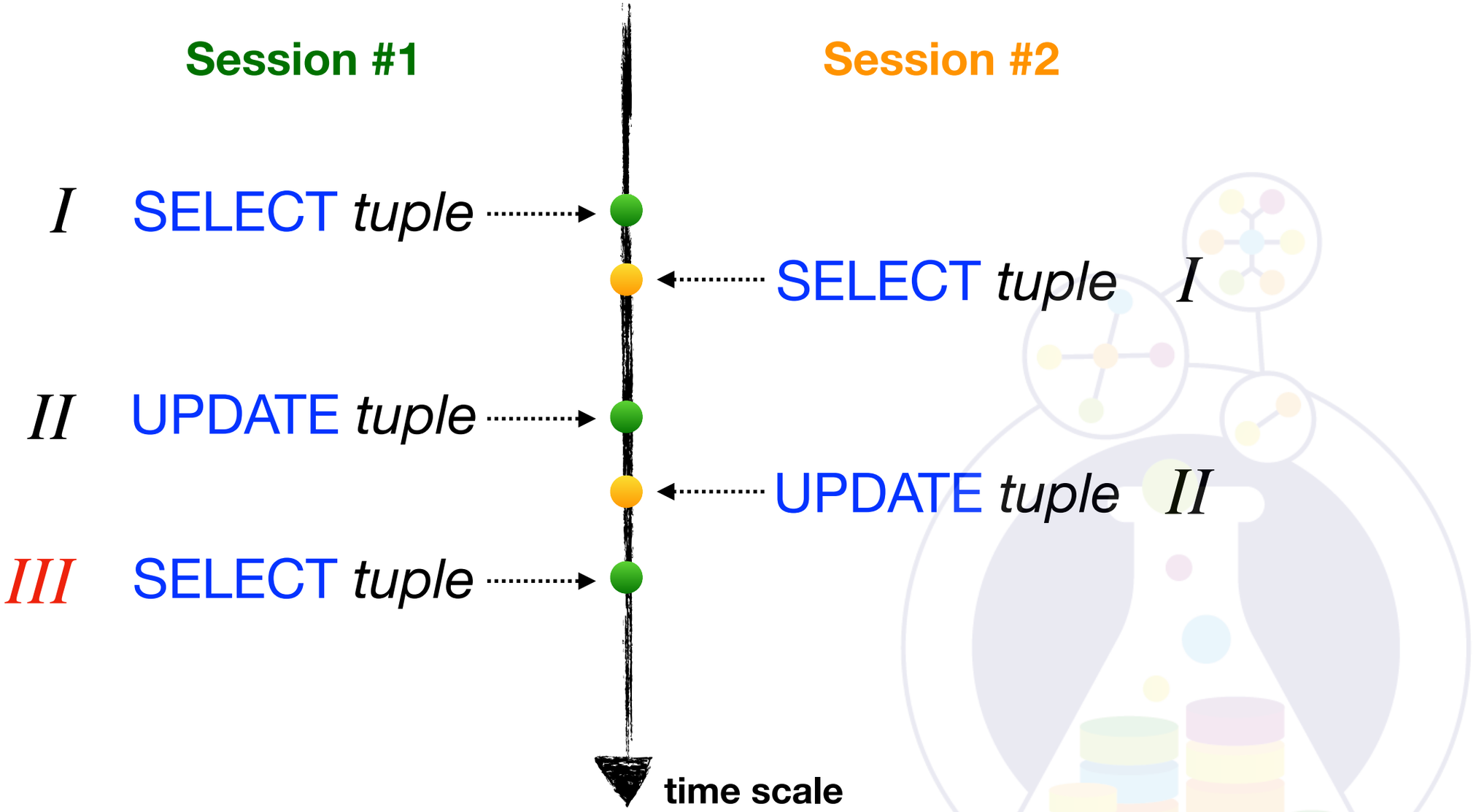
Azat Yakupov

Standard Anomalies

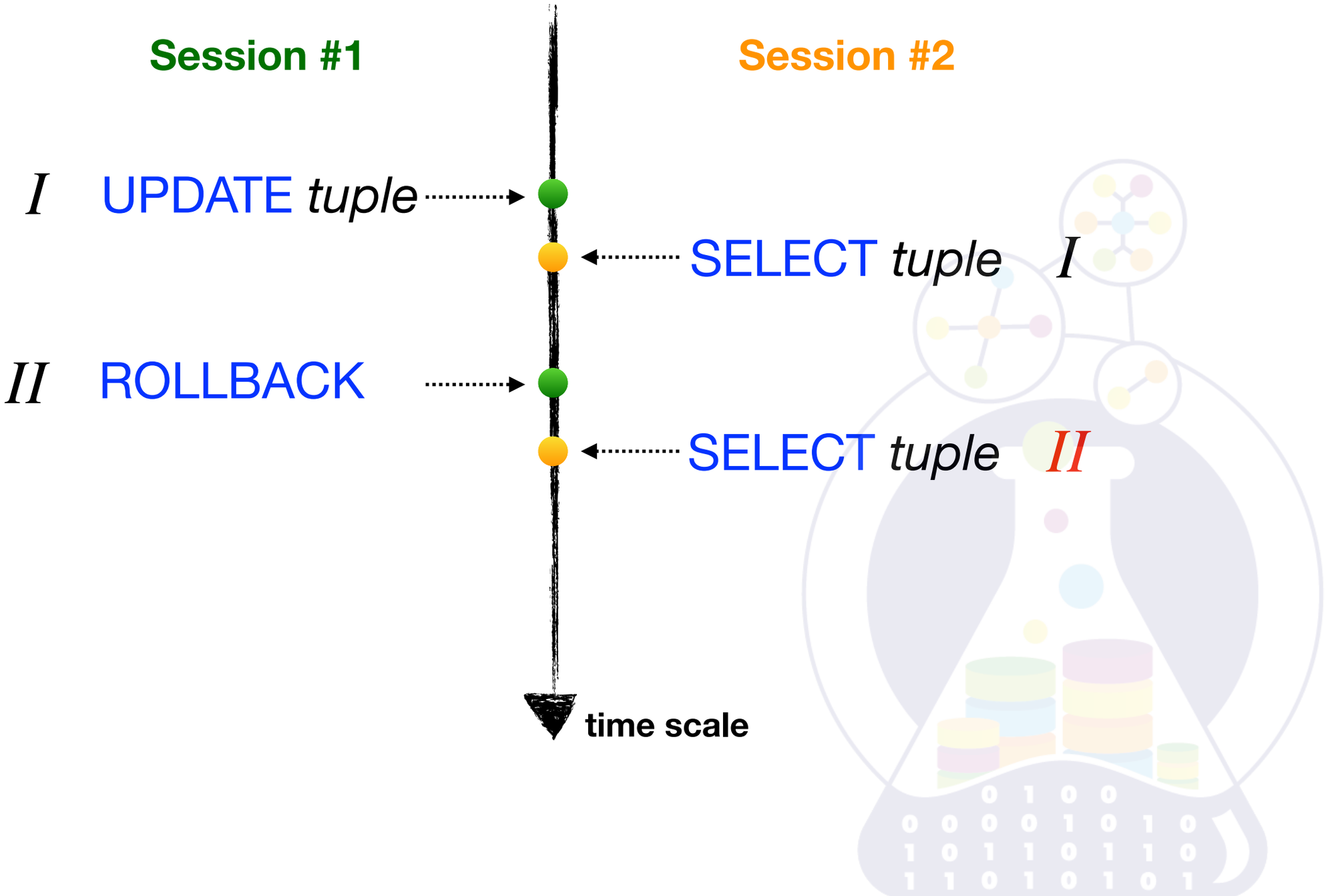
- Lost Update Anomaly
- Dirty Reads Anomaly
- Non-Repeatable Reads Anomaly
- Phantom Reads Anomaly



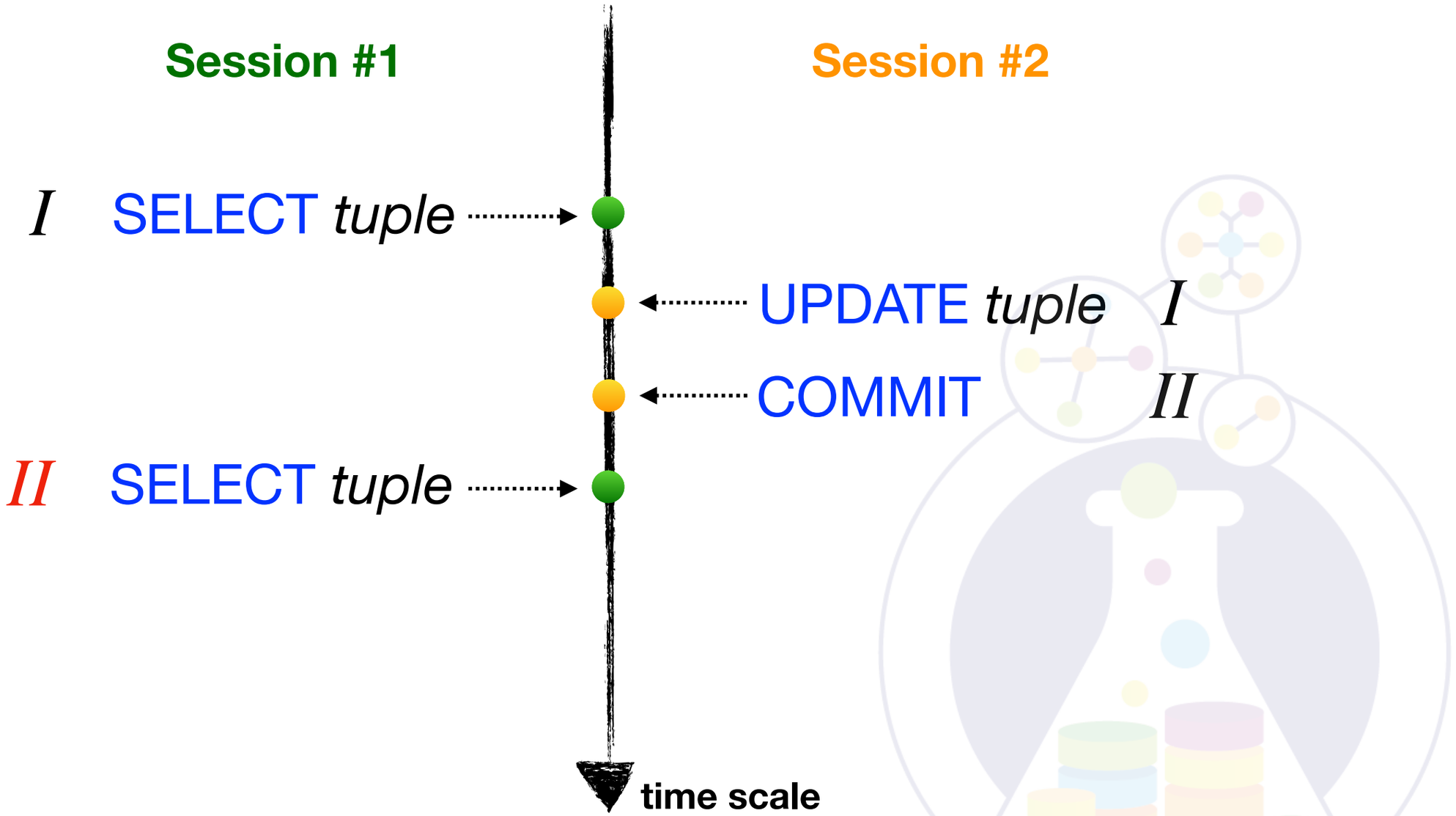
Lost Update Anomaly



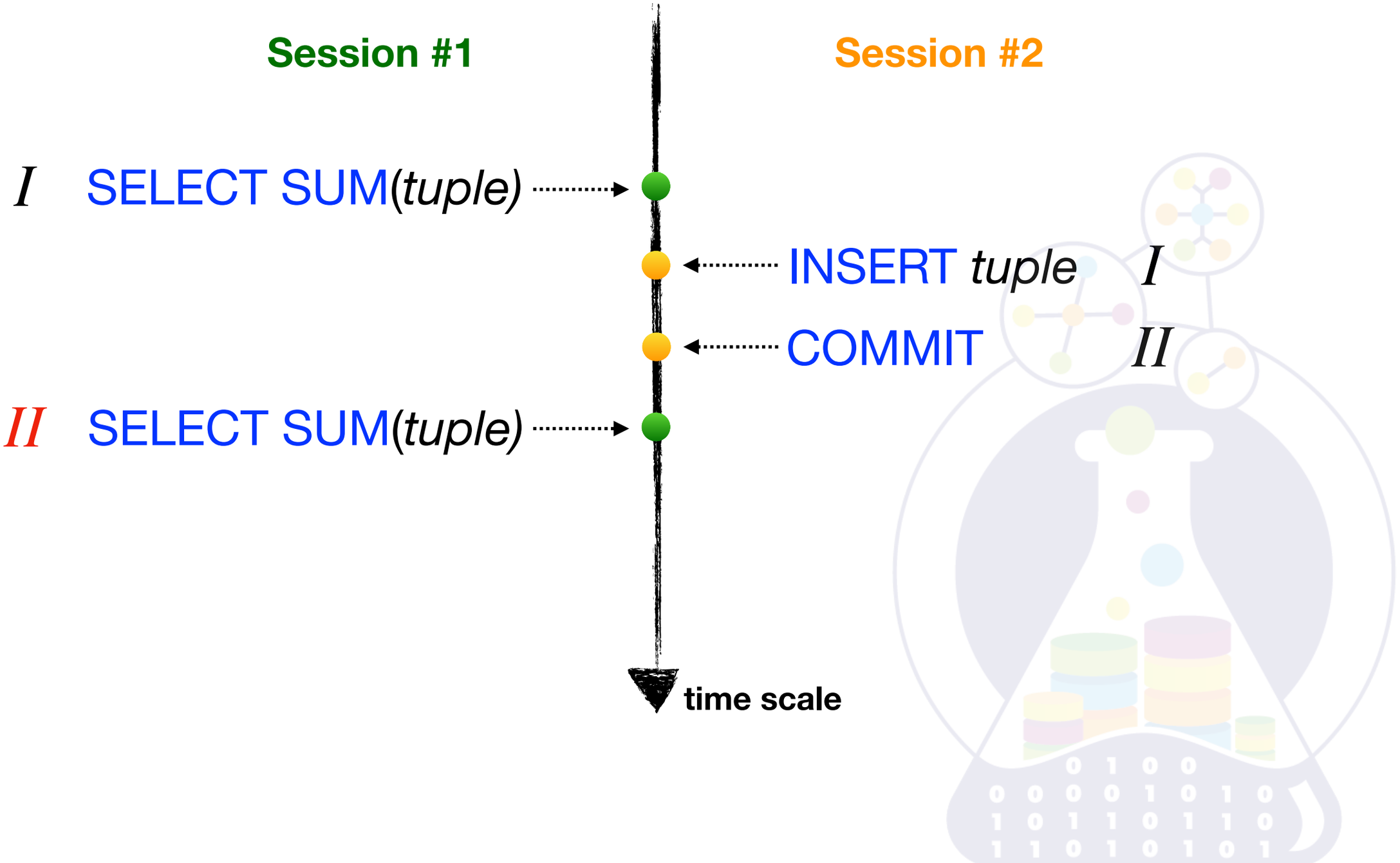
Dirty Reads Anomaly



Non-Repeatable Reads Anomaly



Phantom Reads Anomaly



SQL Standard

ISOLATION LEVEL	Phantom Reads	Non-Repeatable Reads	Dirty Reads	Lost Update
SERIALIZABLE	+	+	+	+
REPEATABLE READ	-	+	+	+
READ COMMITTED	-	-	+	+
READ UNCOMMITTED	-	-	-	+
NO LEVEL	-	-	-	-

Additional Anomalies

- Read Skew Anomaly
- Write Skew Anomaly
- Serialization Anomaly

Post	
id	integer
title	varchar(100)

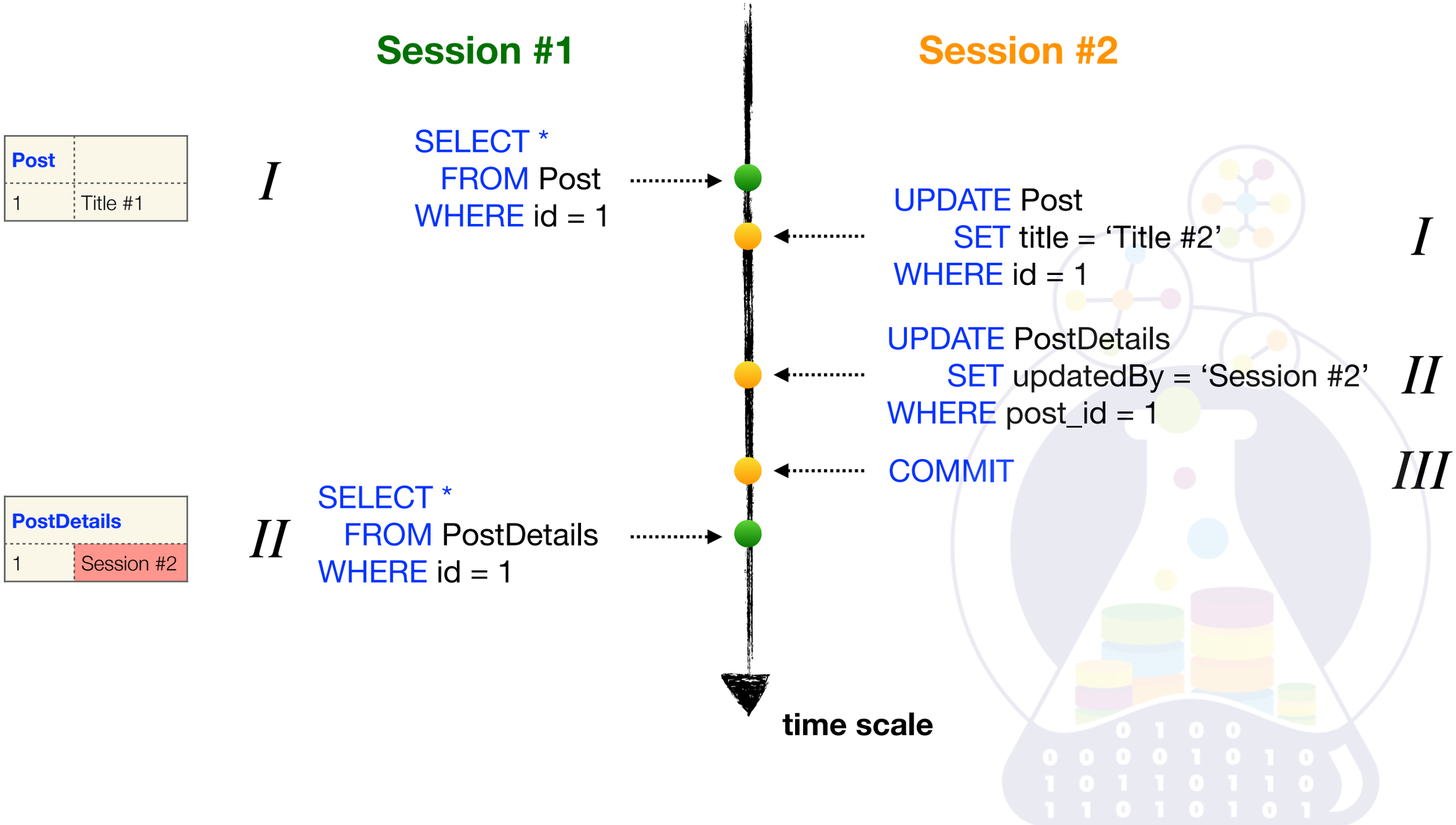
1 — 1

PostDetails	
post_id	integer
updatedBy	varchar(100)

Post	
id	**title**
1	‘Title #1’

PostDetails	
post_id	**updatedBy**
1	‘Session #1’

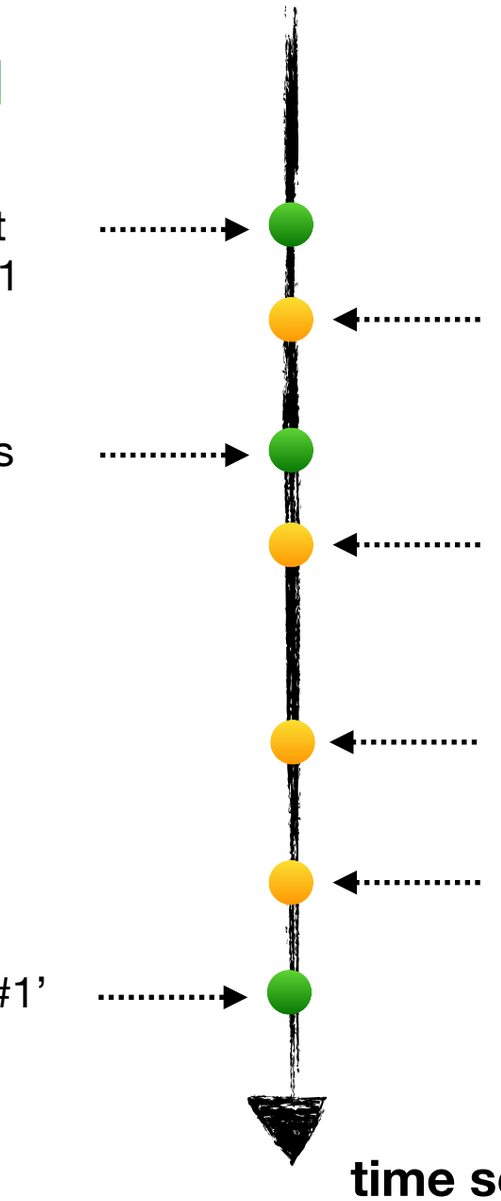
Read Skew Anomaly



Write Skew Anomaly

Post	
1	Title #1

PostDetails	
1	Session #1



I

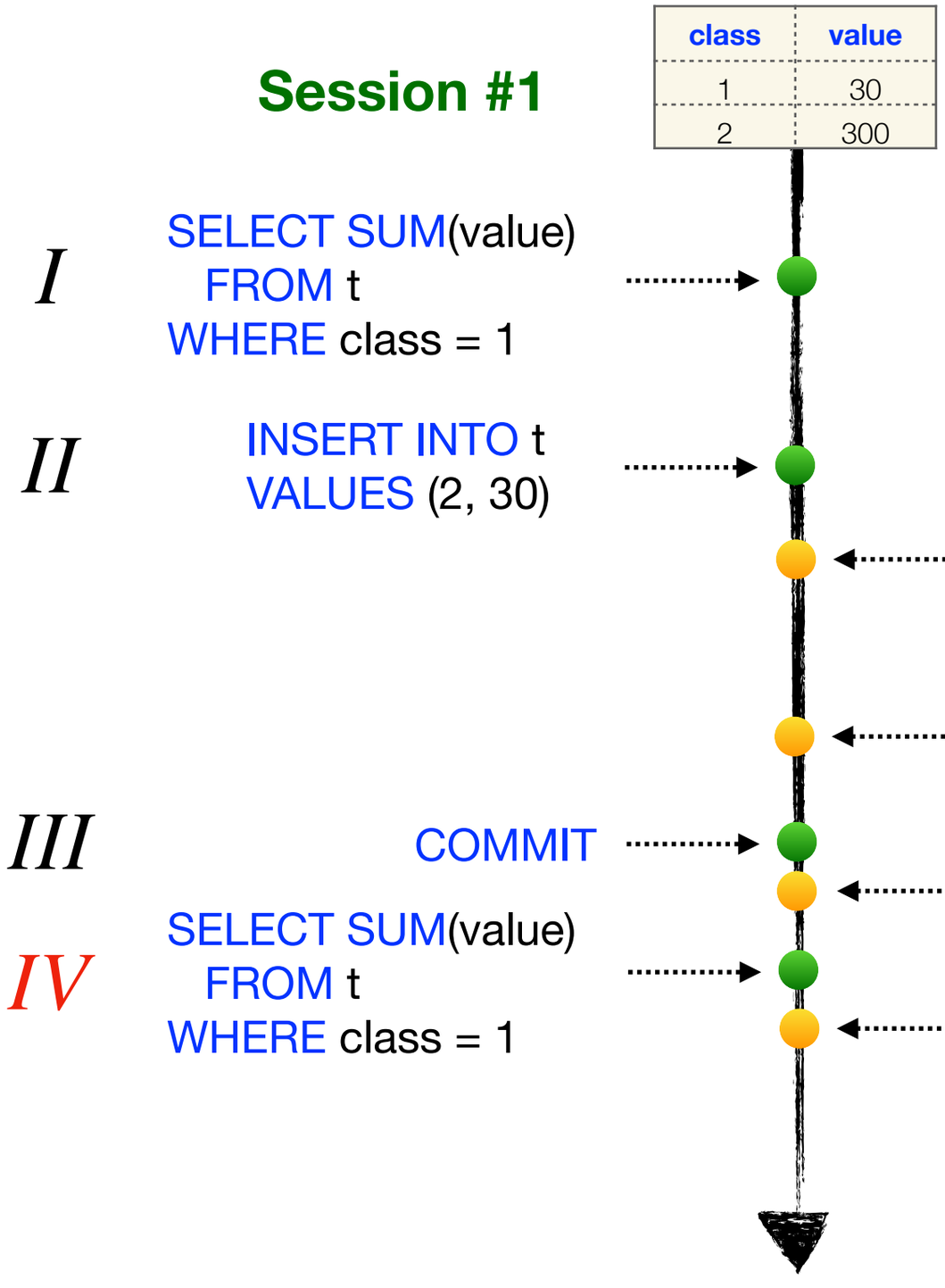
Post	
1	Title #1

II

PostDetails	
1	Session #1

Serialization Anomaly for PostgreSQL

class	value
1	30
2	300
2	**30**

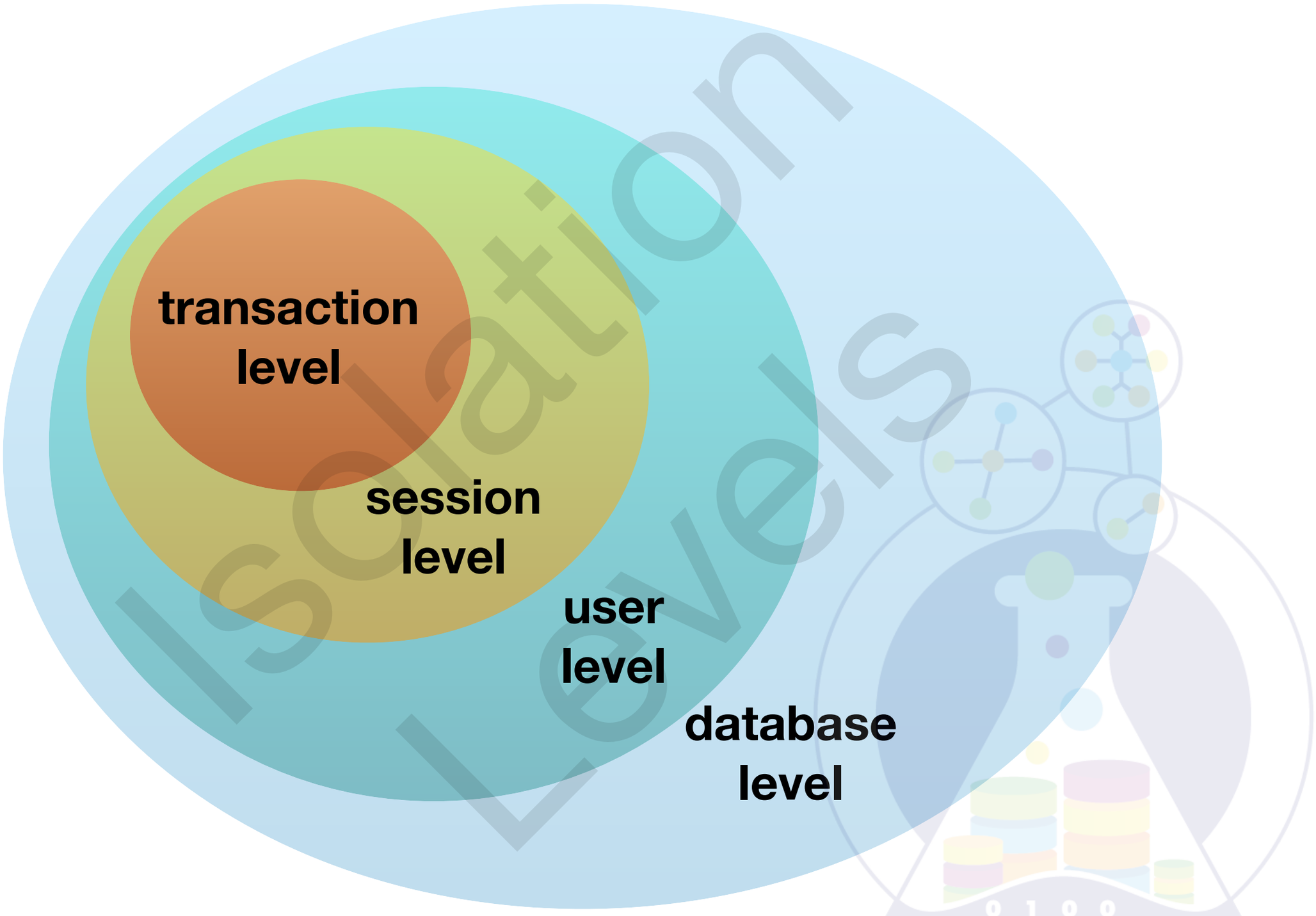


class	value
1	30
2	300
1	**300**

ISOLATION LEVEL	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Read Skew	Write Skew	Lost Update	Serialization Anomaly
SERIALIZABLE	+	+	+	+	+	+	+
REPEATABLE READ	+	+	+	+	+	+	-
READ COMMITTED	+	-	-	-	-	-	-
READ UNCOMMITTED	+	-	-	-	-	-	-
NO LEVEL	-	-	-	-	-	-	-

ISOLATION LEVEL	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Read Skew	Write Skew	Lost Update
SERIALIZABLE	+	+	+	+	-	+
READ COMMITTED	+	-	-	-	-	-
NO LEVEL	-	-	-	-	-	-

ISOLATION LEVEL	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Read Skew	Write Skew	Lost Update
SERIALIZABLE	+	+	+	+	+	+
REPEATABLE READ	+	+	+	+	-	-
READ COMMITTED	+	-	-	-	-	-
READ UNCOMMITTED	-	-	-	-	-	-
NO LEVEL	-	-	-	-	-	-



BEGIN TRANSACTION ISOLATION LEVEL *level_name option*;

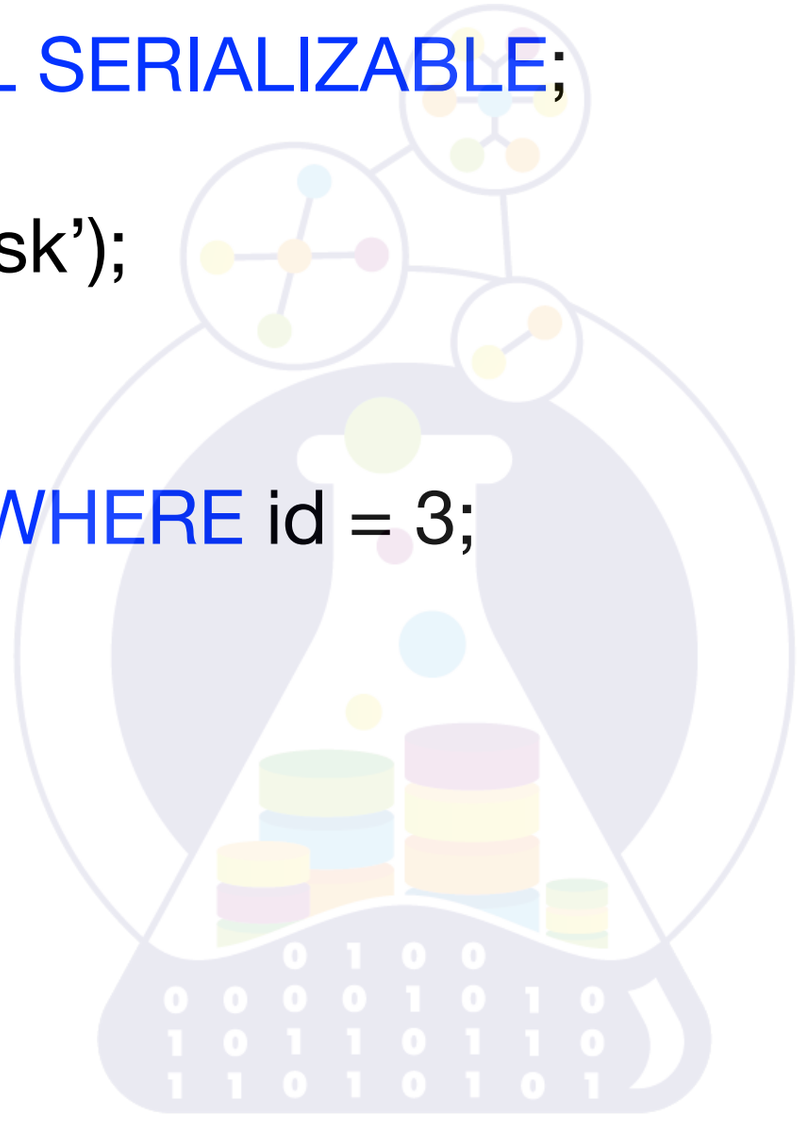
level_name

- SERIALIZABLE
- REPEATABLE READ
- **READ COMMITTED**
- READ UNCOMMITTED

option

- **READ WRITE**
- READ ONLY

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
    INSERT INTO t VALUES (1, ‘Name of Task’);
    SAVEPOINT i1;

    UPDATE employee SET status = ‘fired’ WHERE id = 3;

COMMIT;
```



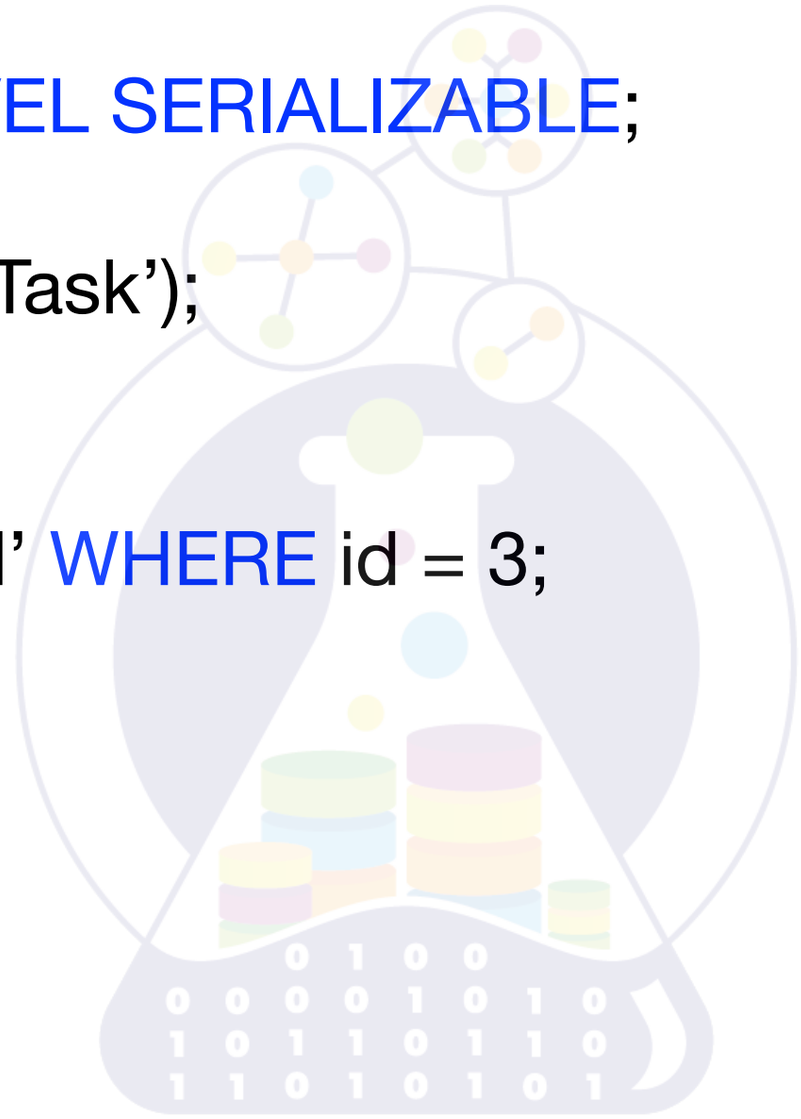
ORACLE

SET TRANSACTION ISOLATION LEVEL *level_name*;

level_name

- SERIALIZABLE
- READ ONLY
- **READ COMMITTED**

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
    INSERT INTO t VALUES (1, ‘Name of Task’);
    SAVEPOINT i1;

    UPDATE employee SET status = ‘fired’ WHERE id = 3;

COMMIT;
```



SET SESSION TRANSACTION ISOLATION LEVEL *level_name*;

level_name

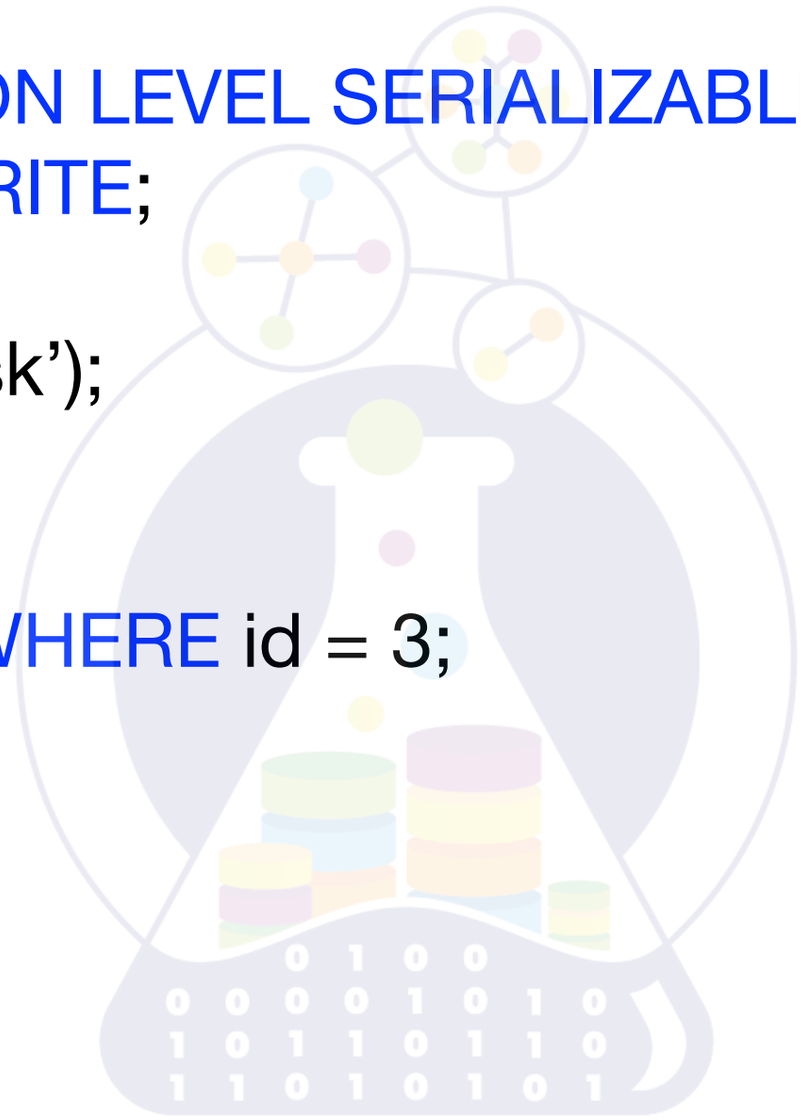
- SERIALIZABLE
- **REPEATABLE READ**
- READ COMMITTED
- READ UNCOMMITTED

SET SESSION TRANSACTION *option*;

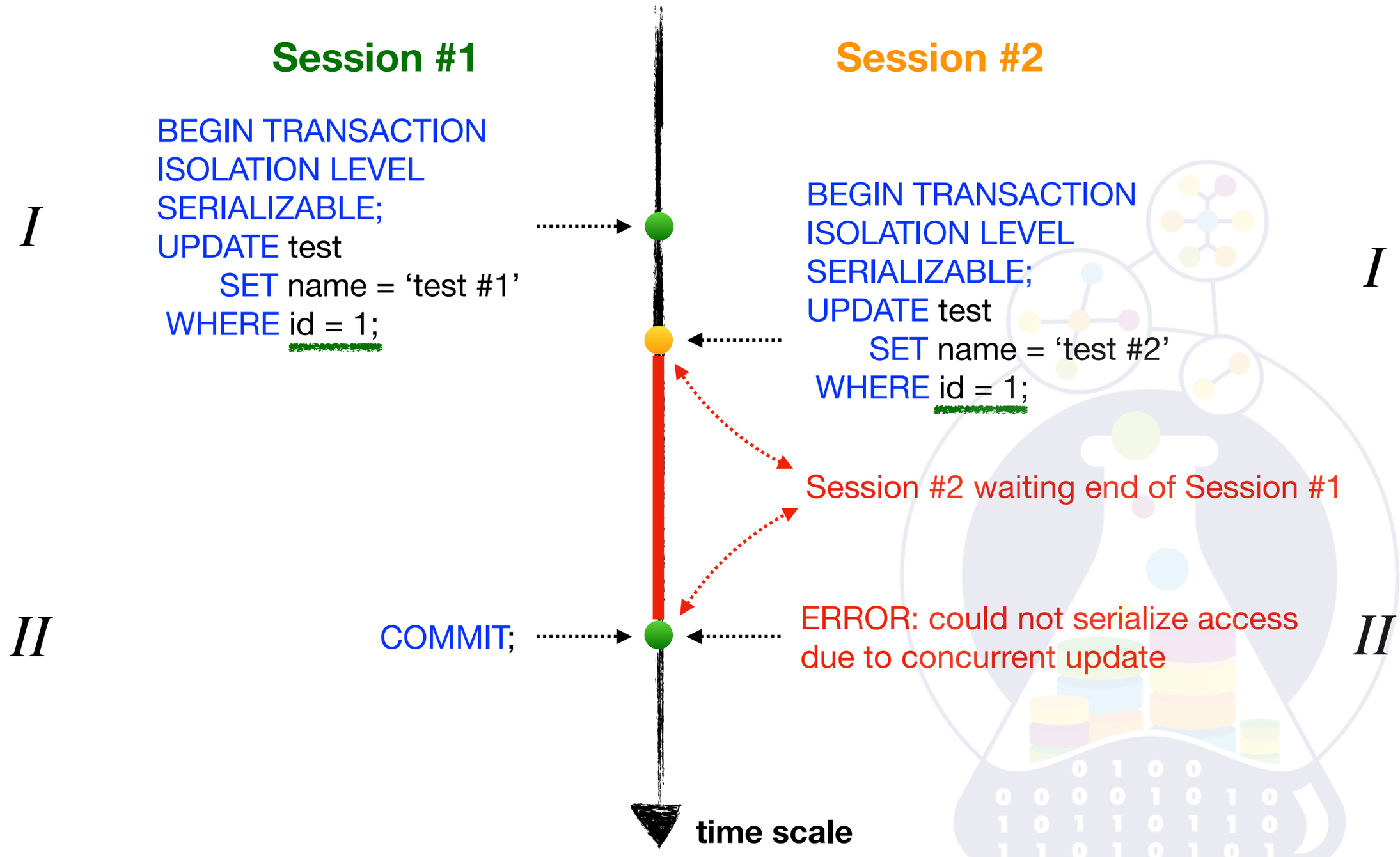
option

- **READ WRITE**
- READ ONLY

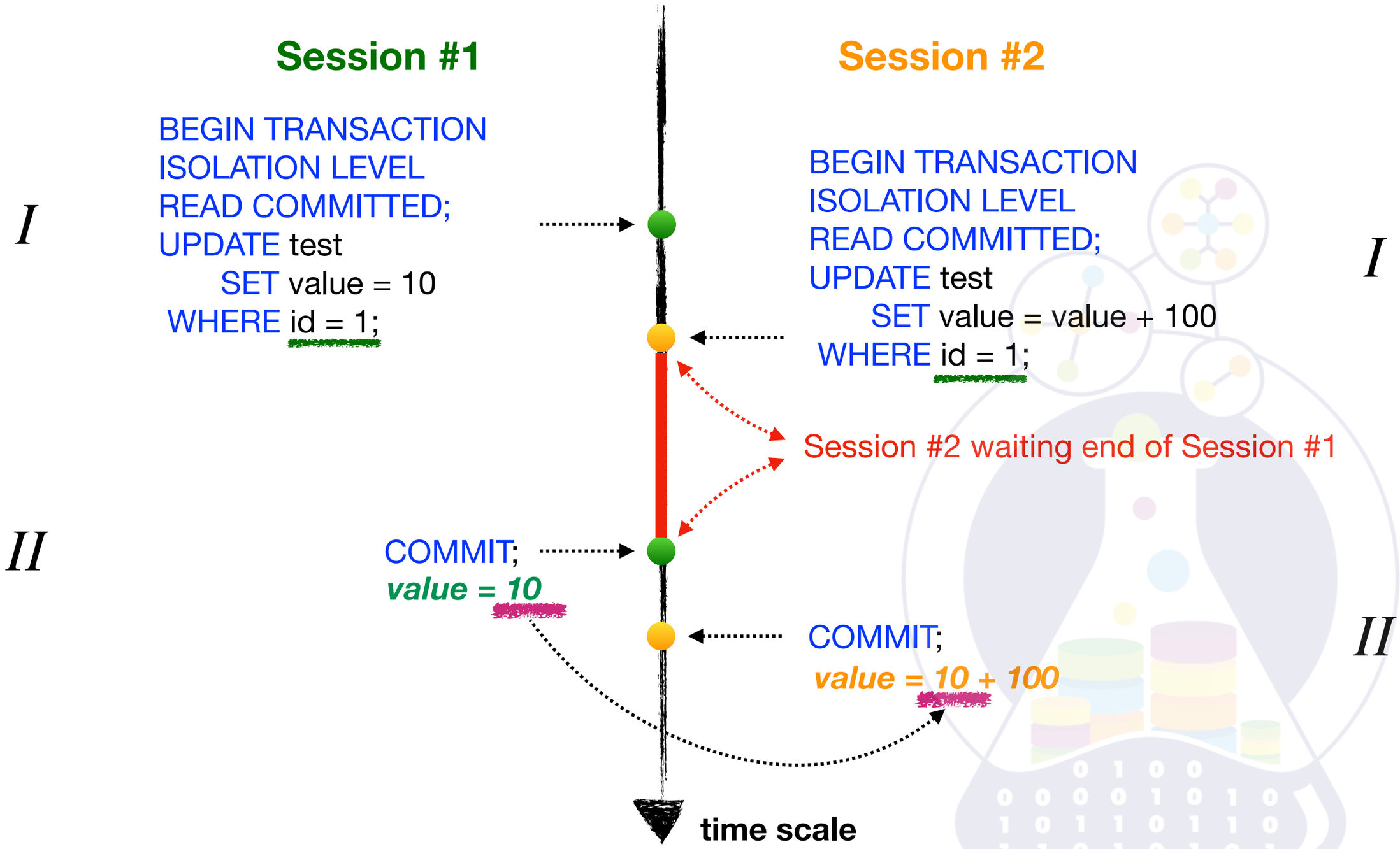
```
SET autocommit = 0;
START TRANSACTION;
   SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
   SET SESSION TRANSACTION READ WRITE;

   INSERT INTO t VALUES (1, ‘Name of Task’);
   SAVEPOINT i1;

   UPDATE employee SET status = ‘fired’ WHERE id = 3;

COMMIT;
```



“Serializable” example

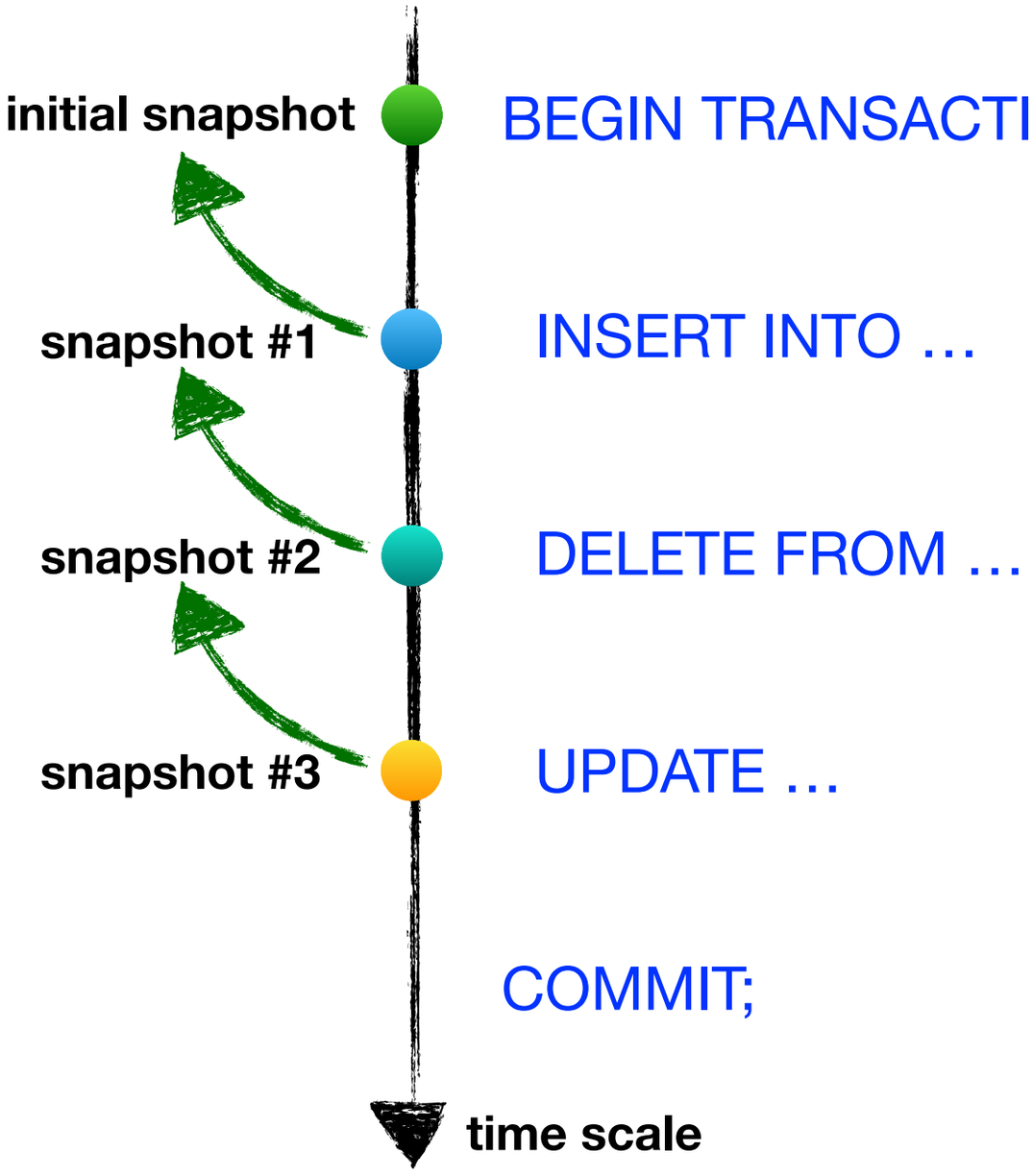


“Read Committed” example



Data Visibility

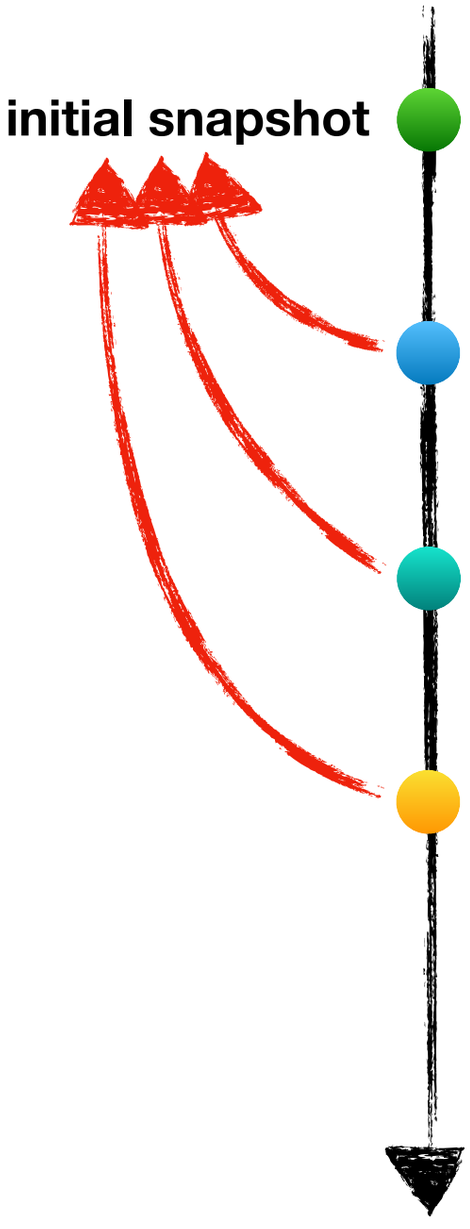
READ [UN]COMMITTED



- makes a snapshot of involved data before each command inside a transaction
- each command can see changes from previous command inside a transaction

Data Visibility

REPEATABLE READ, SERIALIZABLE



- makes a snapshot of involved data before **first** command of a transaction
- all commands have the similar initial snapshot inside a transaction

Dealing with conflicts

- **Conflict avoidance**

  2PL protocol

- **Conflict detection**

  MVCC

Database Locks

eXclusive Locks (~ lock of write)

Shared Locks (~ lock of read)

Compatibility matrix

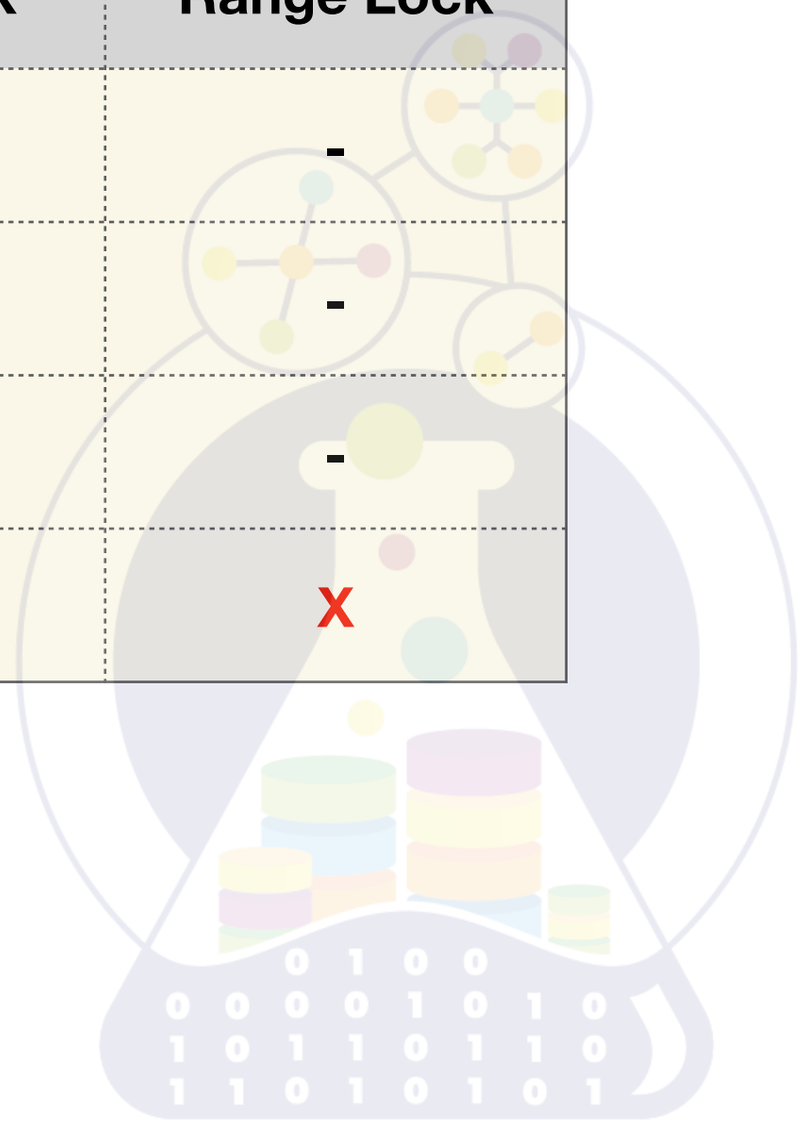
existing lock \ requested lock	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

N means a conflict

Y means no conflict

Isolation Levels and Locks

Isolation Level	Read Lock	Write Lock	Range Lock
Read Uncommitted	-	-	-
Read Committed	S	X	-
Repeatable Read	X	X	-
Serializable	X	X	X



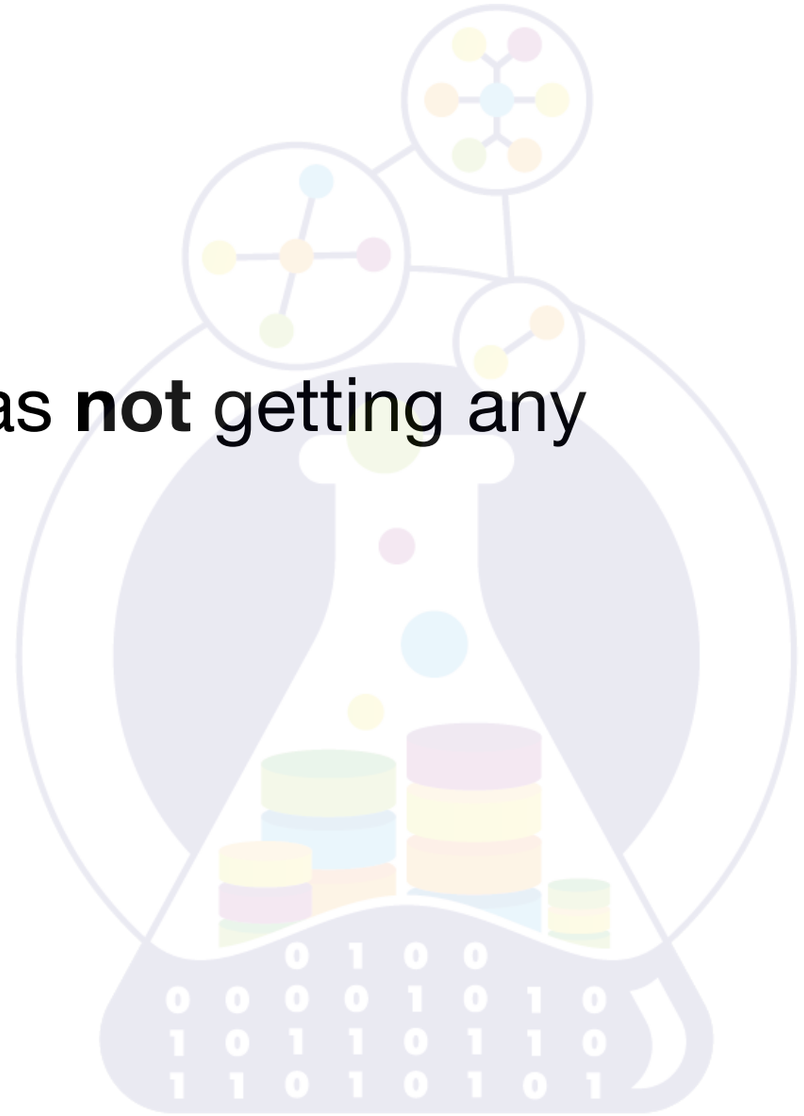
2 Phase Locking protocol

- the transaction has getting a lock on object (or tuple) before any action with this object (or tuple)

  ~ locking phase

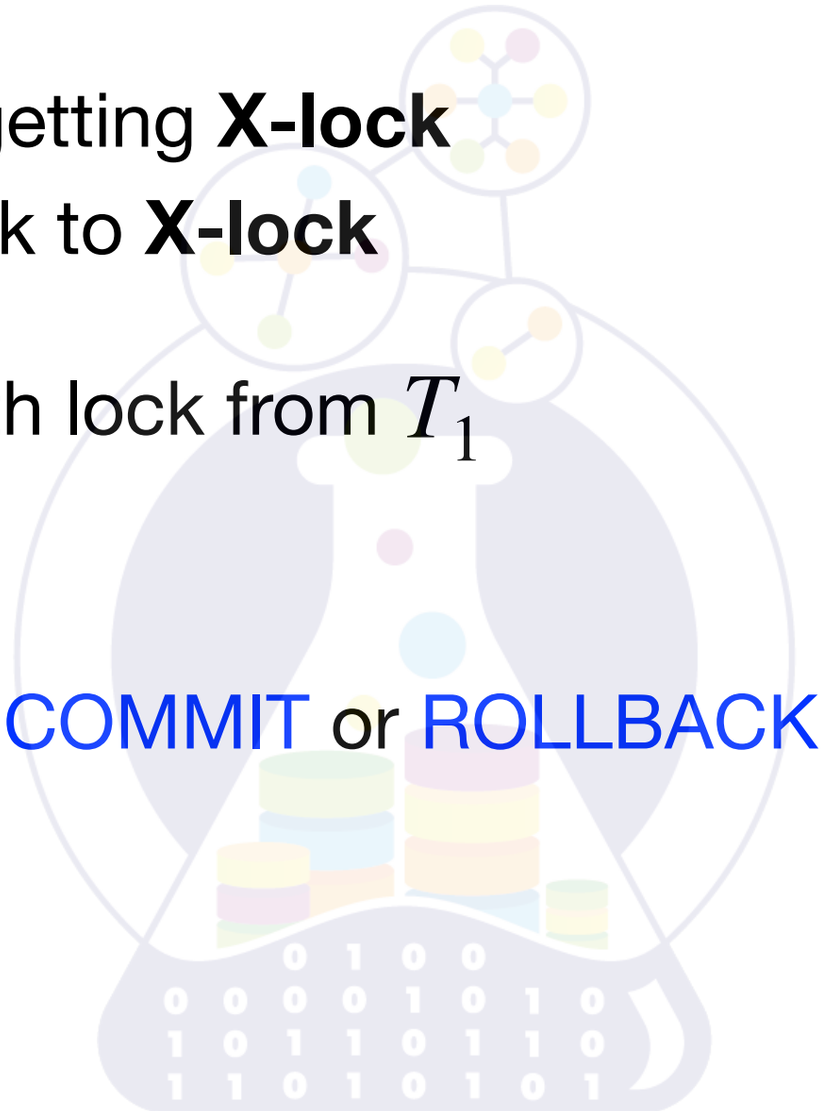
- after release a lock the transaction has **not** getting any additional locks

  ~ release phase



The **strict** 2 Phase Locking protocol

- To select any data from table the T_1 has getting **S-lock**
- To update data from table the T_1 has getting **X-lock**
  If T_1 has **S-lock** then T_1 expands lock to **X-lock**
- If T_2 needs to lock an object (tuple) with lock from T_1 then T_2 waits lock's release from T_1
- **X-Lock** and **S-lock** will be released by COMMIT or ROLLBACK



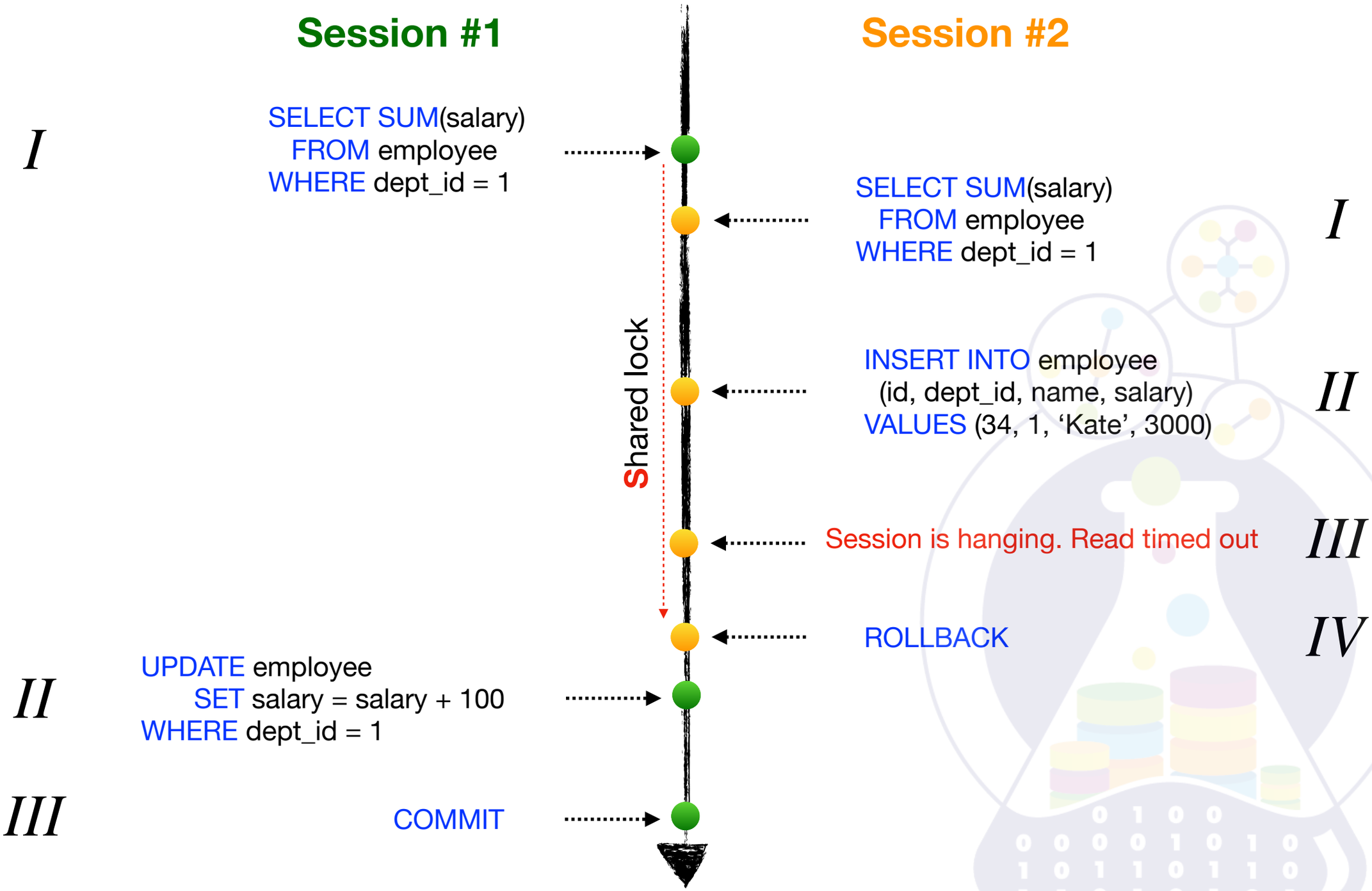
MVCC

Readers don’t block **Writers**

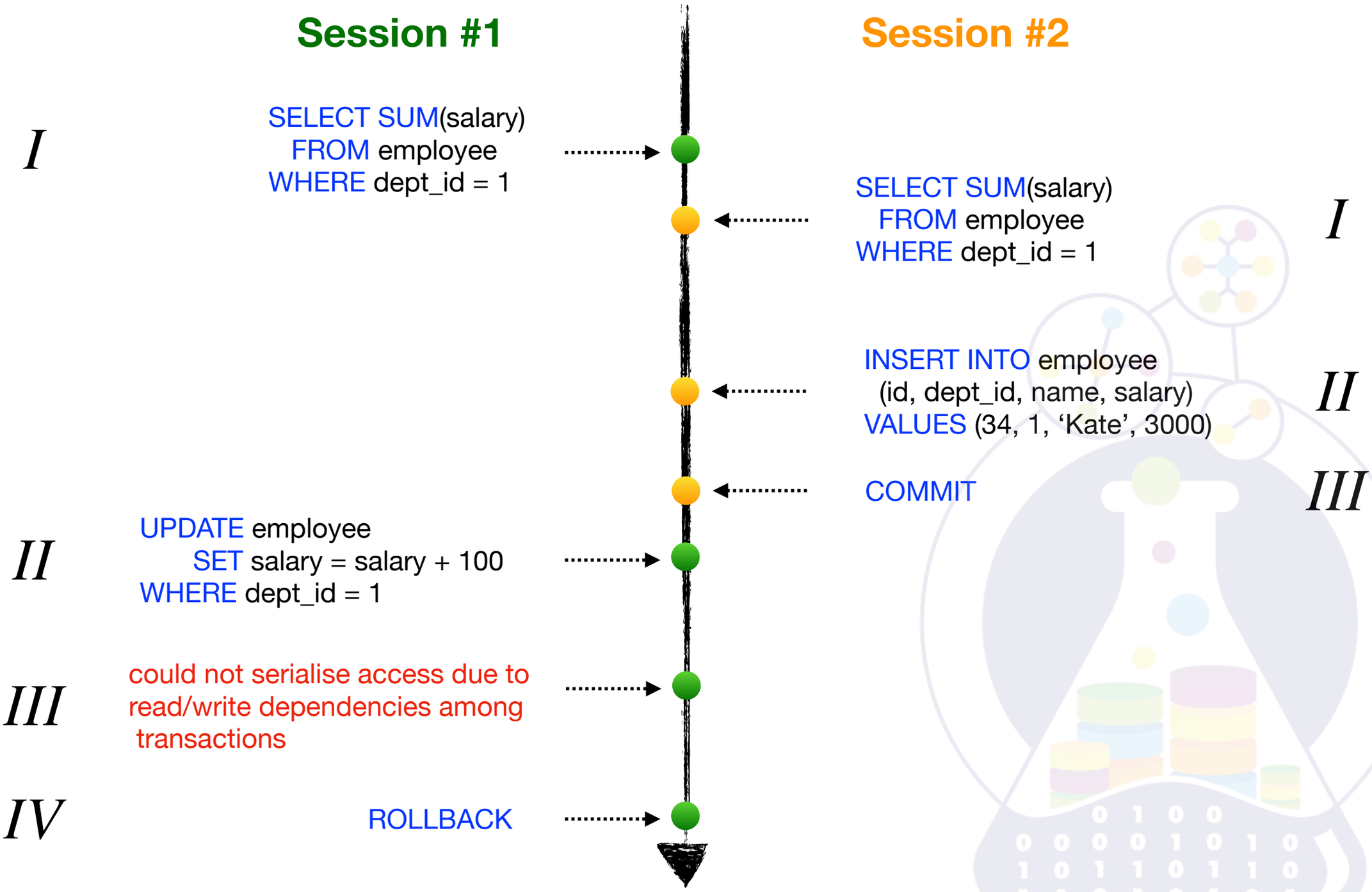
Writers don’t block **Readers**

Writers block **Writers** to prevent Dirty Writes

2PL protocol. How to...

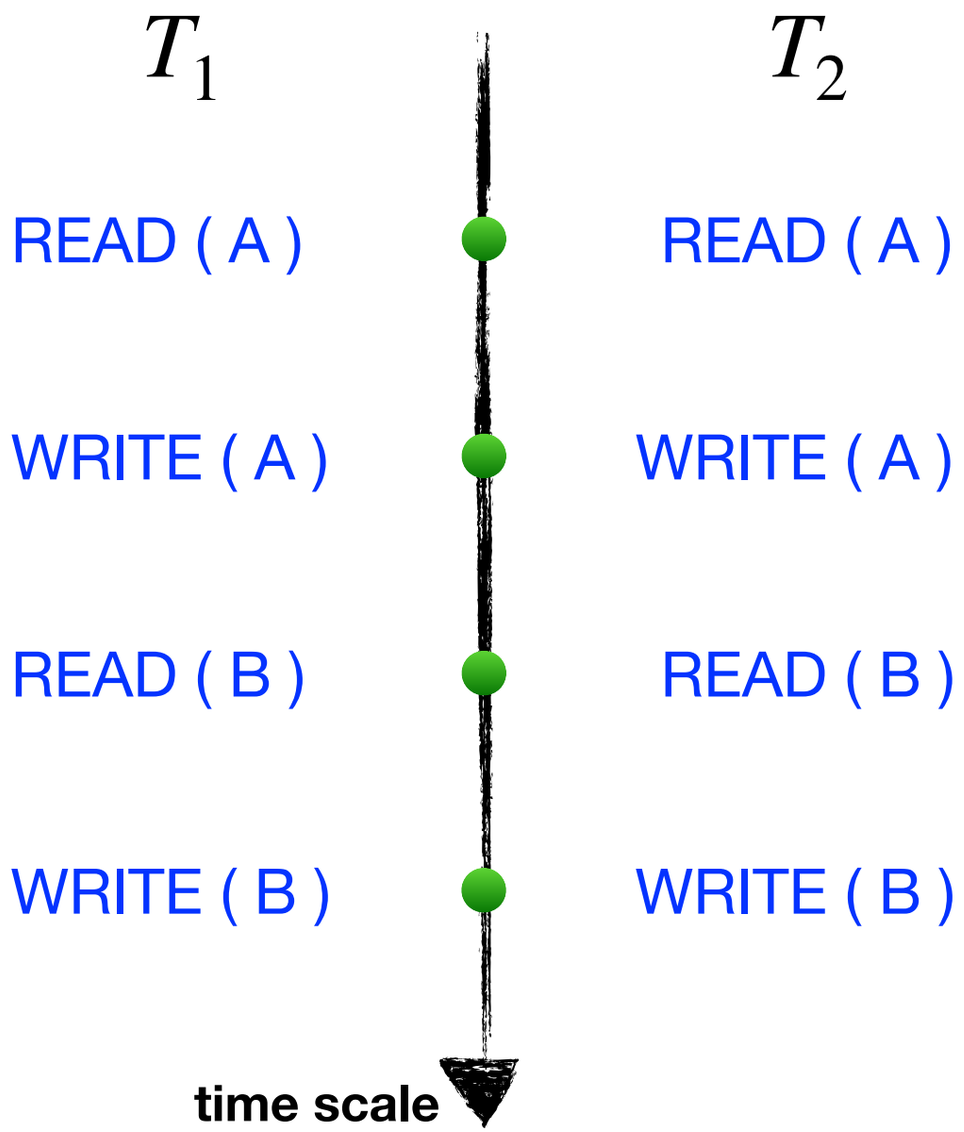


MVCC. How to...



Precedence Graph

Schedule - sequence of actions with data ordering in time



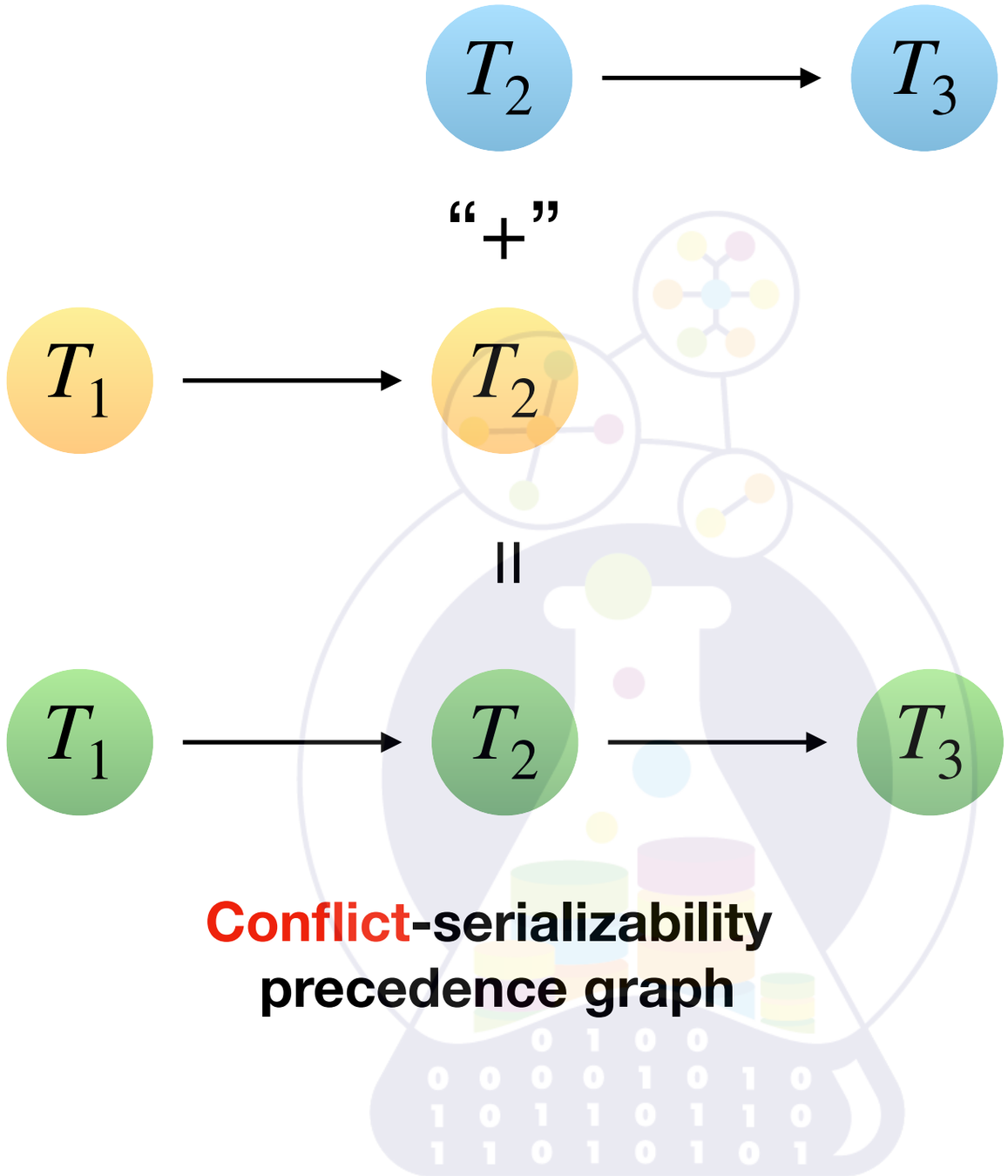
$$T_1 = r_1(A); w_1(A); r_1(B); w_1(B);$$

$$T_2 = r_2(A); w_2(A); r_2(B); w_2(B);$$

$$S = r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$$

T_1	T_2	T_3
	READ (A)	
READ (B)		
	WRITE (A)	
		READ (A)
WRITE (B)		
		WRITE (A)
	READ (B)	
	WRITE (B)	

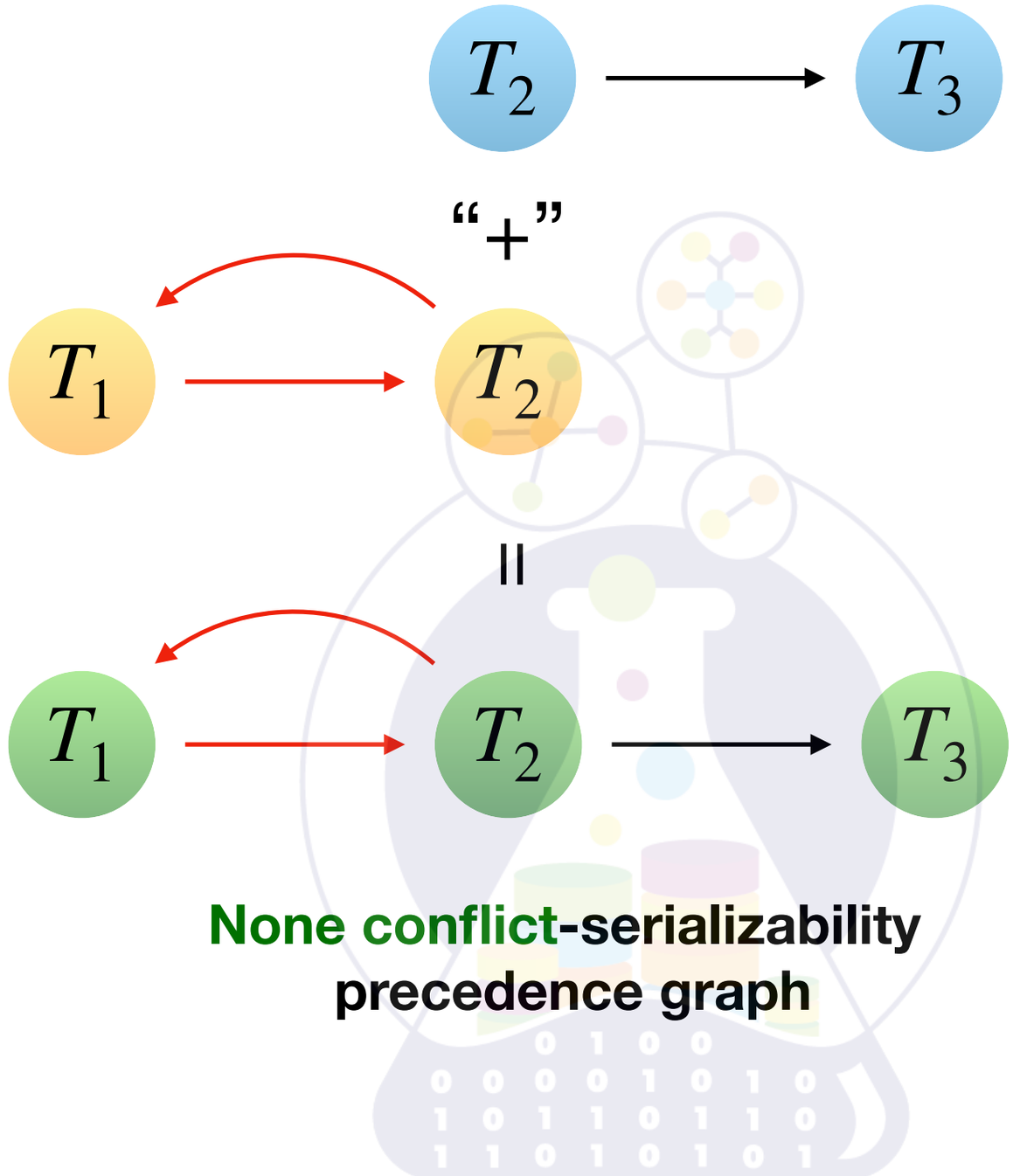
time scale



$$S = r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$$

T_1	T_2	T_3
	READ (A)	
READ (B)		
	WRITE (A)	
	READ (B)	
		READ (A)
WRITE (B)		
		WRITE (A)
	WRITE (B)	

time scale



Intent Locks

Intent **S**hared (~ **IS**)

Intent e**X**clusive (~ **IX**)

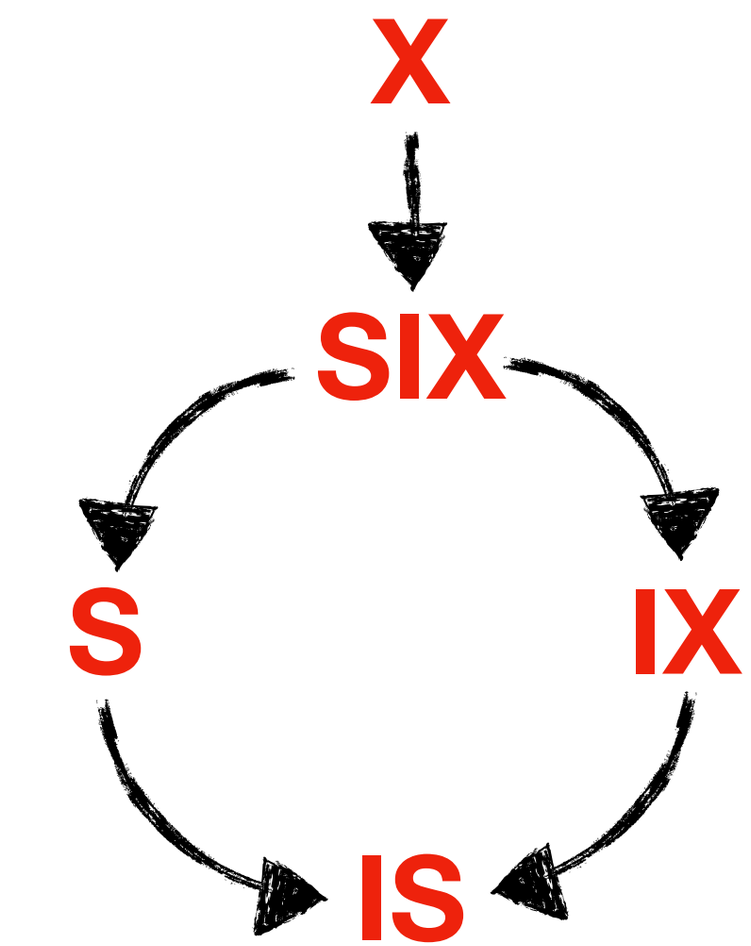
Shared **I**ntent e**X**clusive (~ **SIX**)

N means a conflict

Y means no conflict

existing lock \ requested lock	X	SIX	IX	S	IS	-
X	N	N	N	N	N	Y
SIX	N	N	N	N	Y	Y
IX	N	N	Y	N	Y	Y
S	N	N	N	Y	Y	Y
IS	N	Y	Y	Y	Y	Y
-	Y	Y	Y	Y	Y	Y

Warning-Locking Protocol

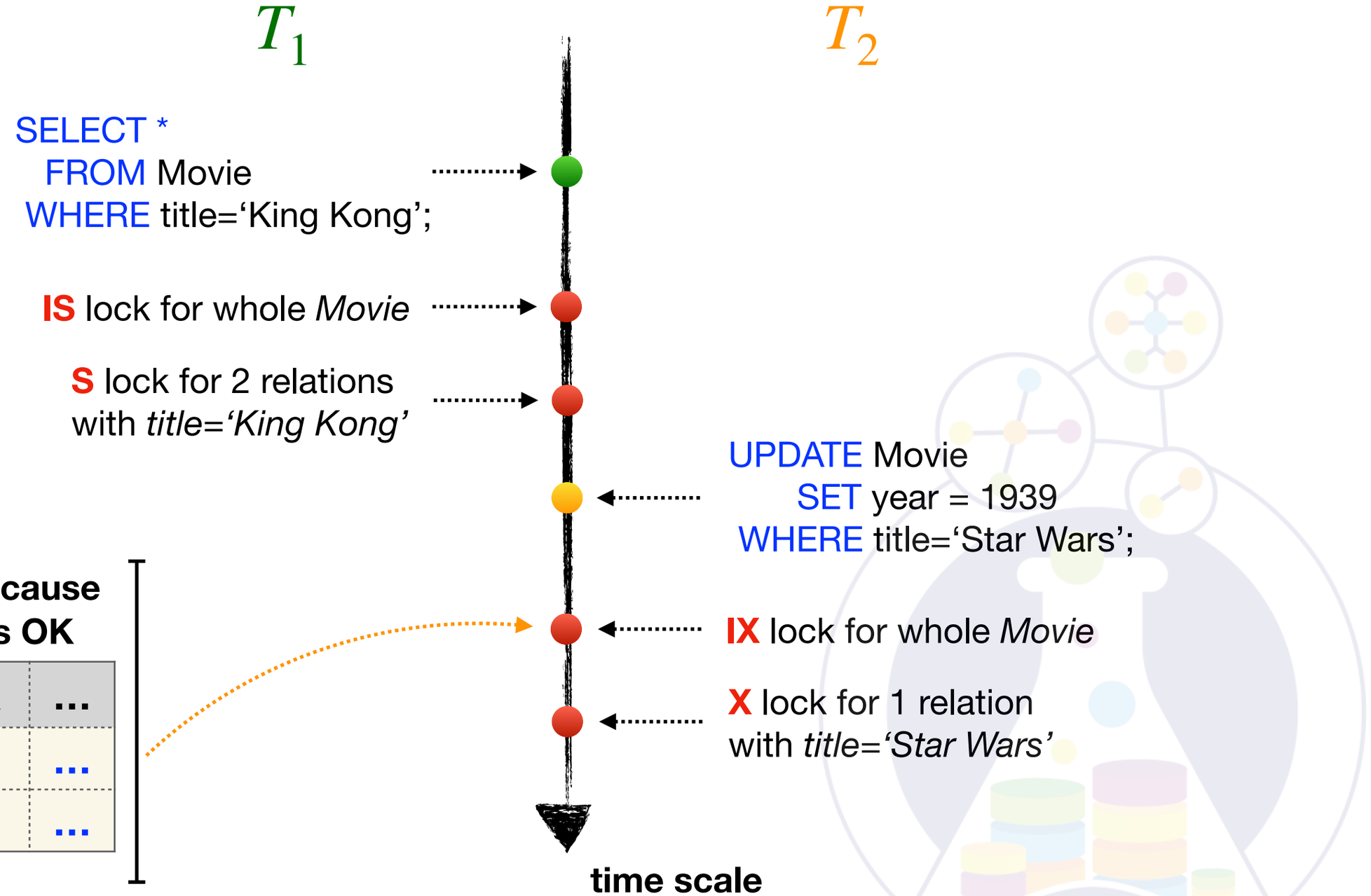


Precedence Graph of Locks

- to get **S** on a tuple,
  the transaction has to get **IS**
  or **strongest lock** on a relation

- to get **X** on a tuple,
  the transaction has to get **IX**
  or **strongest lock** on a relation

$$Movie(title, year, length, studioName)$$



PostgreSQL, Oracle, MySQL

```
SELECT *
    FROM pg_locks;
```

Table-level Lock Modes

- access share
- row share
- row exclusive
- share update exclusive
- share
- share row exclusive
- exclusive
- access exclusive

```
SELECT *
    FROM v$locks;
```

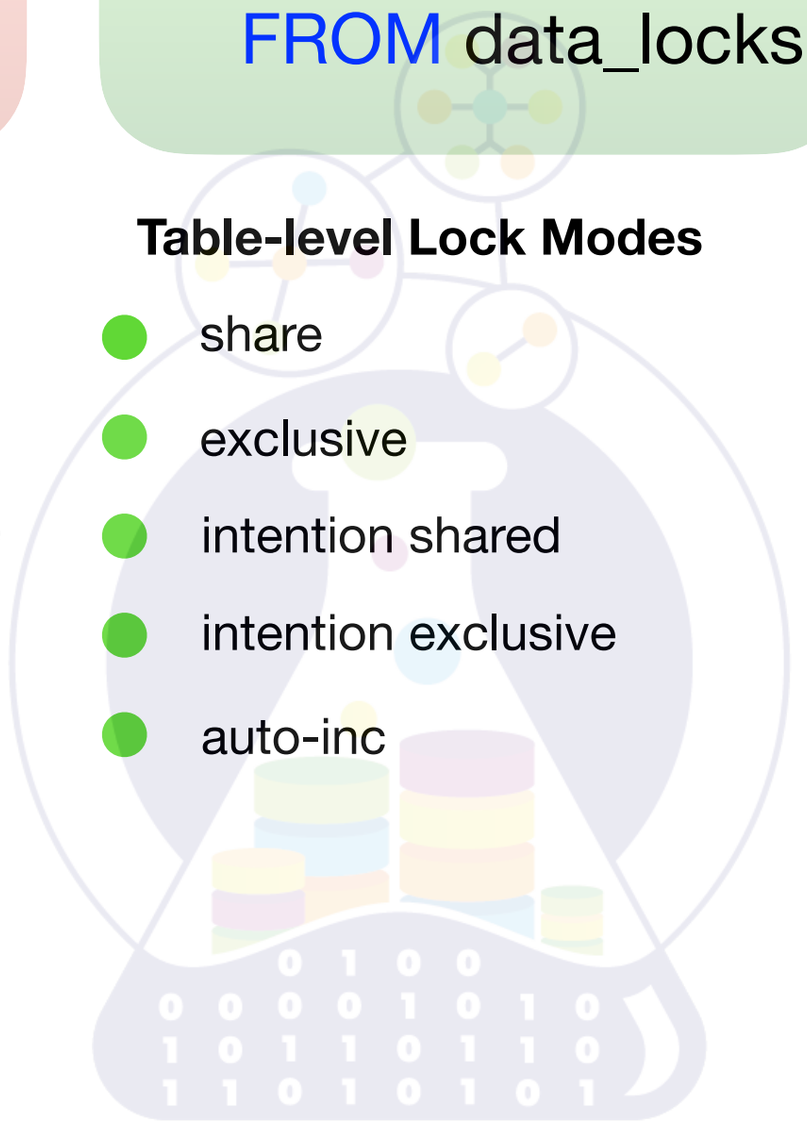
Table-level Lock Modes

- share
- row share ~ subshare
- row exclusive ~ subexclusive
- share row exclusive ~ share-subexclusive
- exclusive

```
SELECT *
    FROM data_locks;
```

Table-level Lock Modes

- share
- exclusive
- intention shared
- intention exclusive
- auto-inc



PostgreSQL, Oracle, MySQL

```
SELECT *
    FROM pg_locks;
```

Row-level Lock Modes

- for update
- for no key update
- for share
- for key share

```
SELECT *
    FROM v$locks;
```

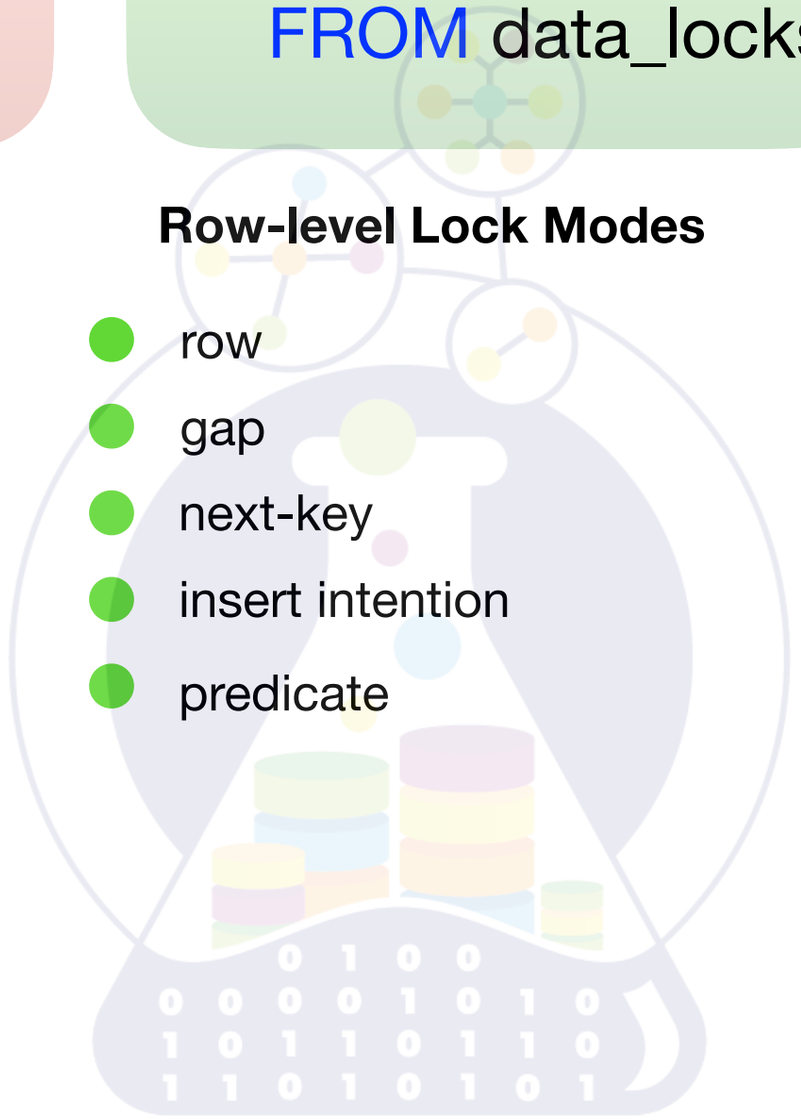
Row-level Lock Modes

- row ~ TX lock

```
SELECT *
    FROM data_locks;
```

Row-level Lock Modes

- row
- gap
- next-key
- insert intention
- predicate



PostgreSQL Table Explicit Lock

LOCK TABLE *table_names [,]* IN *lock_mode* MODE [NOWAIT]

- *lock_mode* = **Table-level Lock Modes**
- NOWAIT = if resource is busy another locks then transaction is aborted

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

  LOCK TABLE employee IN SHARE MODE;
  ...

COMMIT;
```

Oracle Table Explicit Lock

LOCK TABLE *table_names [,]* IN *lock_mode* MODE [NOWAIT]

- *lock_mode* = **Table-level Lock Modes**
- NOWAIT = if resource is busy another locks then transaction is aborted

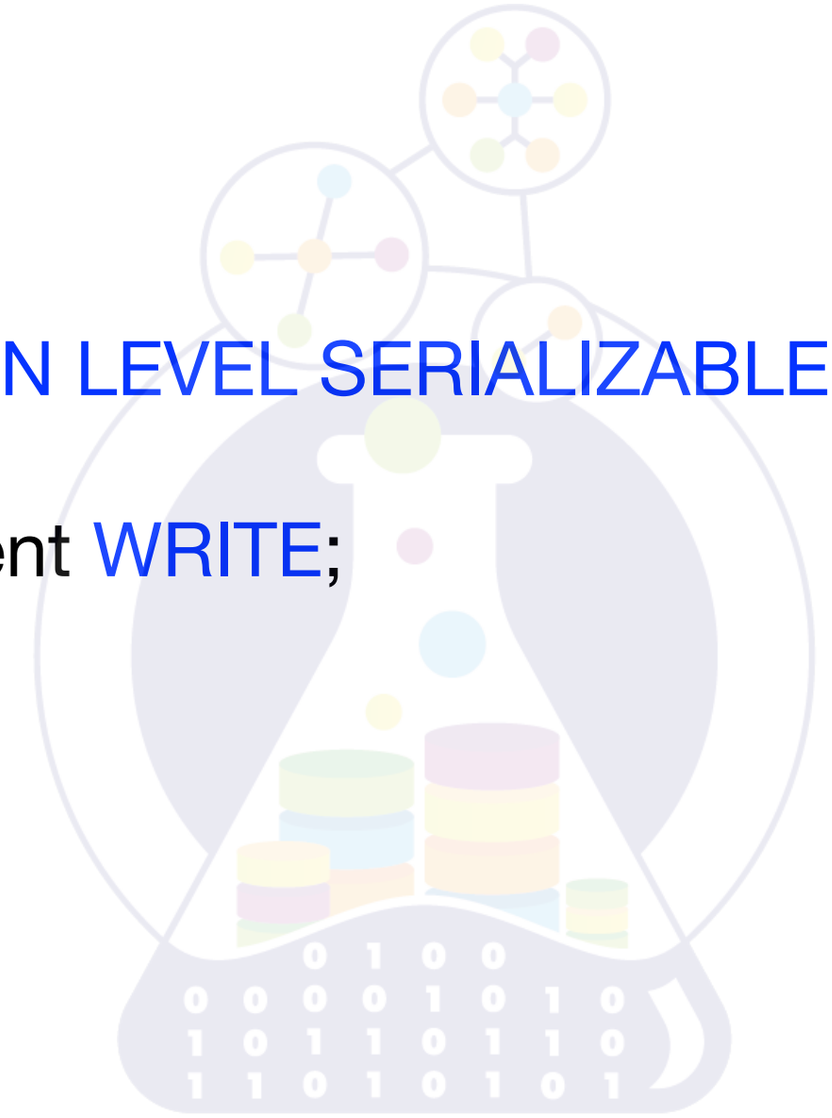
```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

  LOCK TABLE employee IN SHARE MODE NOWAIT;
  ...

COMMIT;
```

MySQL Table Explicit Lock

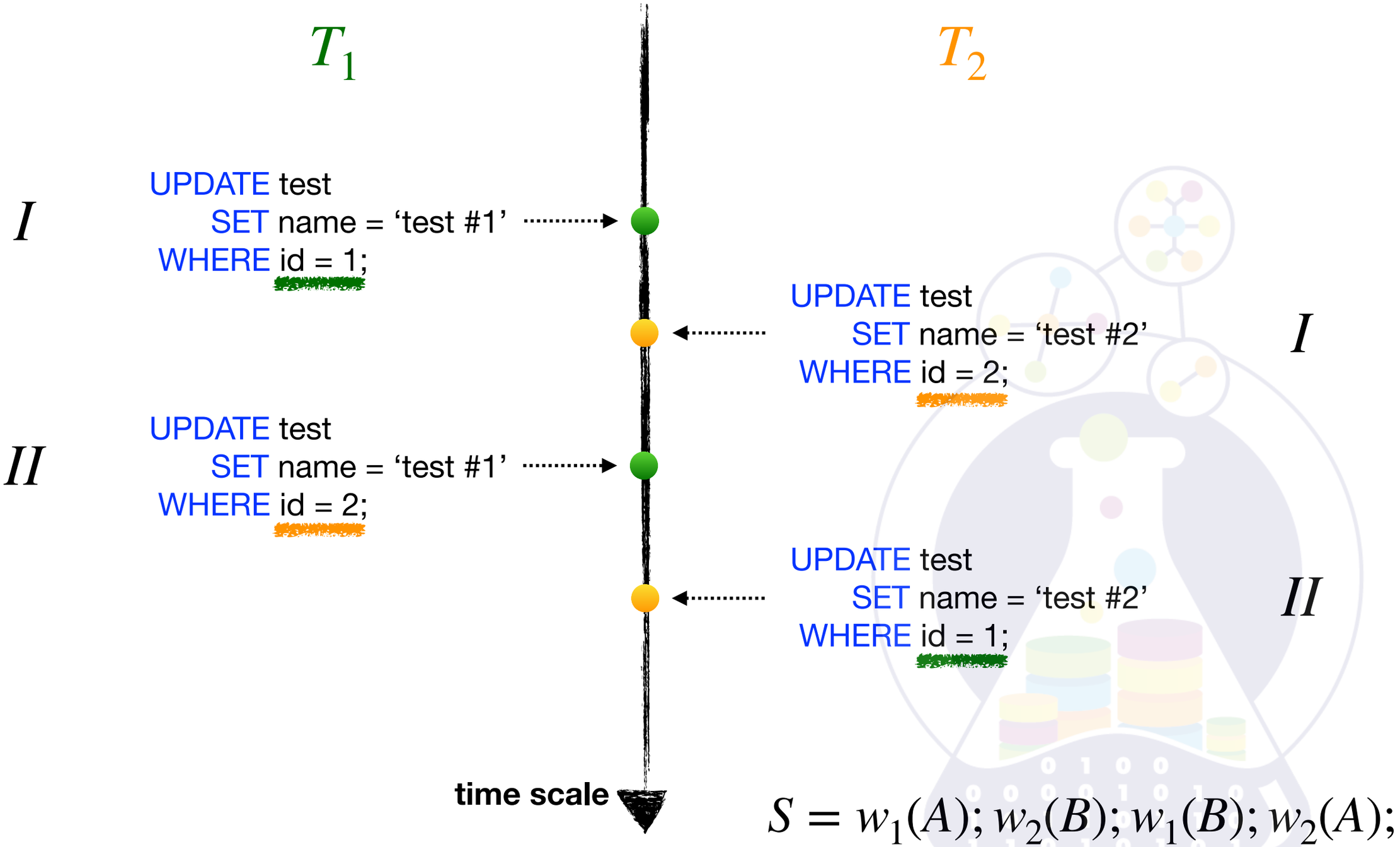
LOCK TABLE *table_names [,] lock_type*

- *lock_type* = **READ** | **WRITE**

```
SET autocommit = 0;
START TRANSACTION;
  SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;

  LOCK TABLE employee READ, department WRITE;
  ...

COMMIT;

UNLOCK TABLES;
```

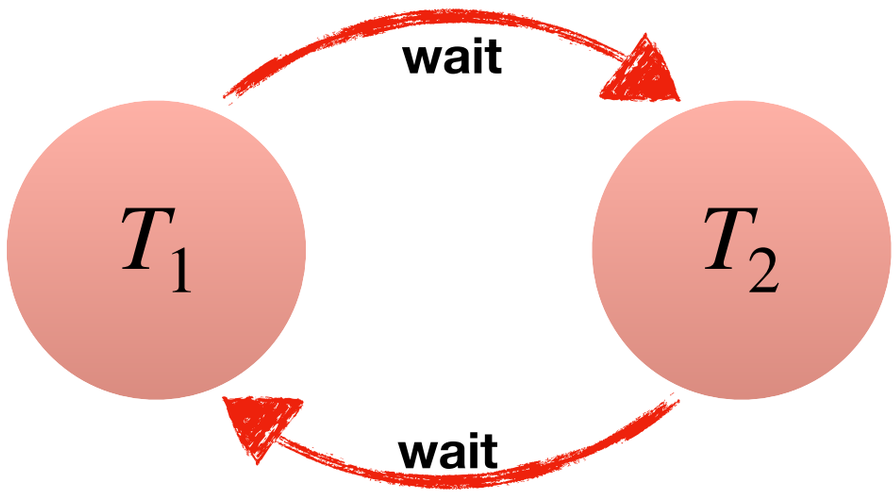


Database Deadlock

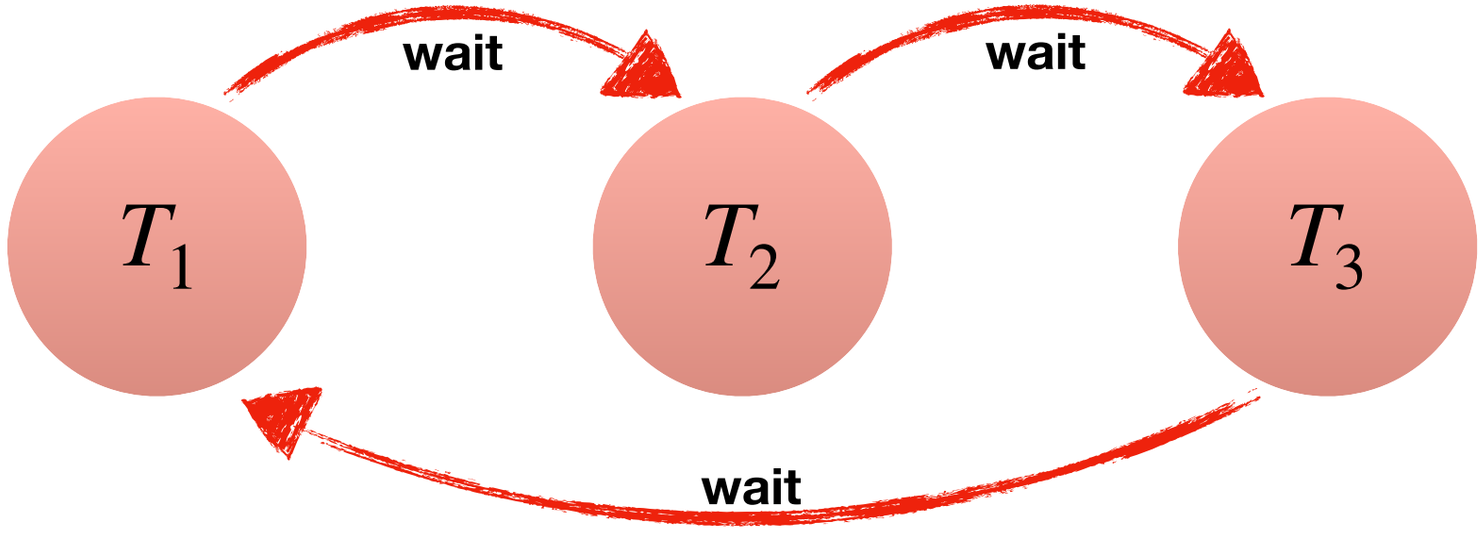


$$S = w_1(A); w_2(B); w_1(B); w_2(A);$$

Wait-For Graph

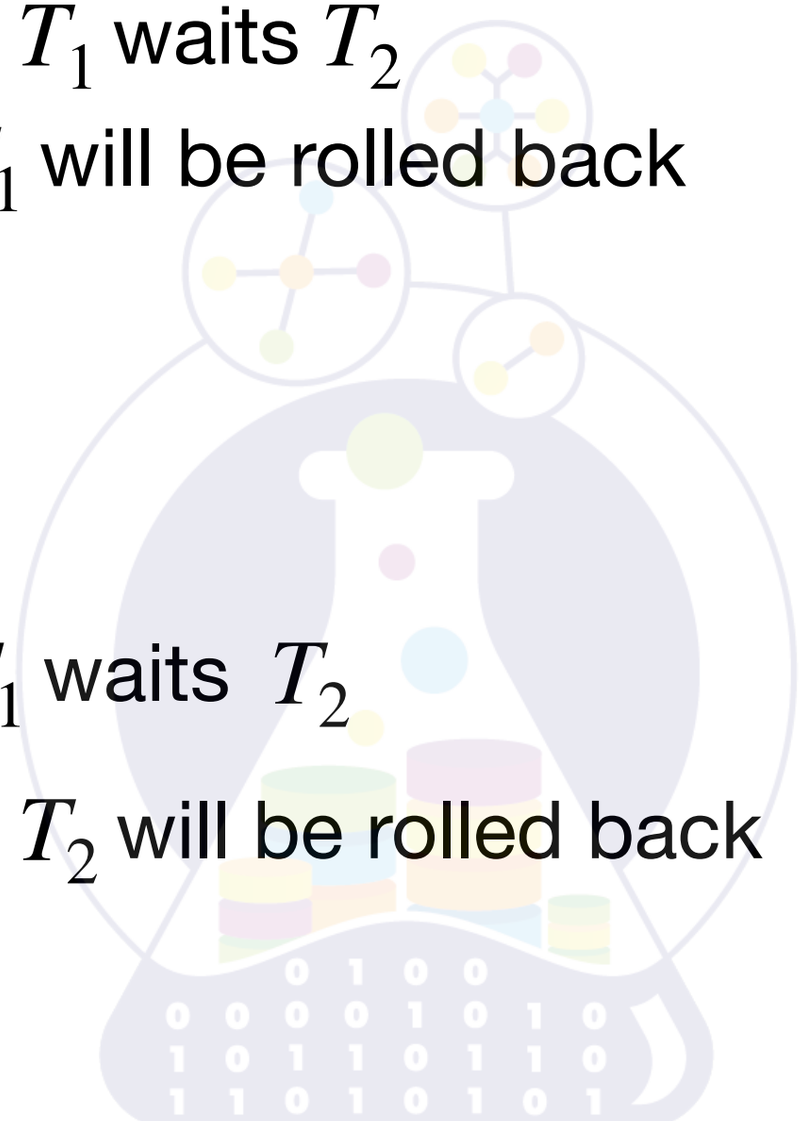


Node is a transaction;
Edge is a fact that transaction is waiting for a lock held by another transaction.



Resolve deadlocks

- **“Wait-Die” schema**
  - if T_1 begins before T_2 , then T_1 waits T_2
  - if T_1 begins later T_2, then T_1 will be rolled back and rescheduled
- **“Wound-Wait” schema**
  - if T_1 begins later T_2, then T_1 waits T_2
  - if T_1 begins before T_2 , then T_2 will be rolled back and rescheduled

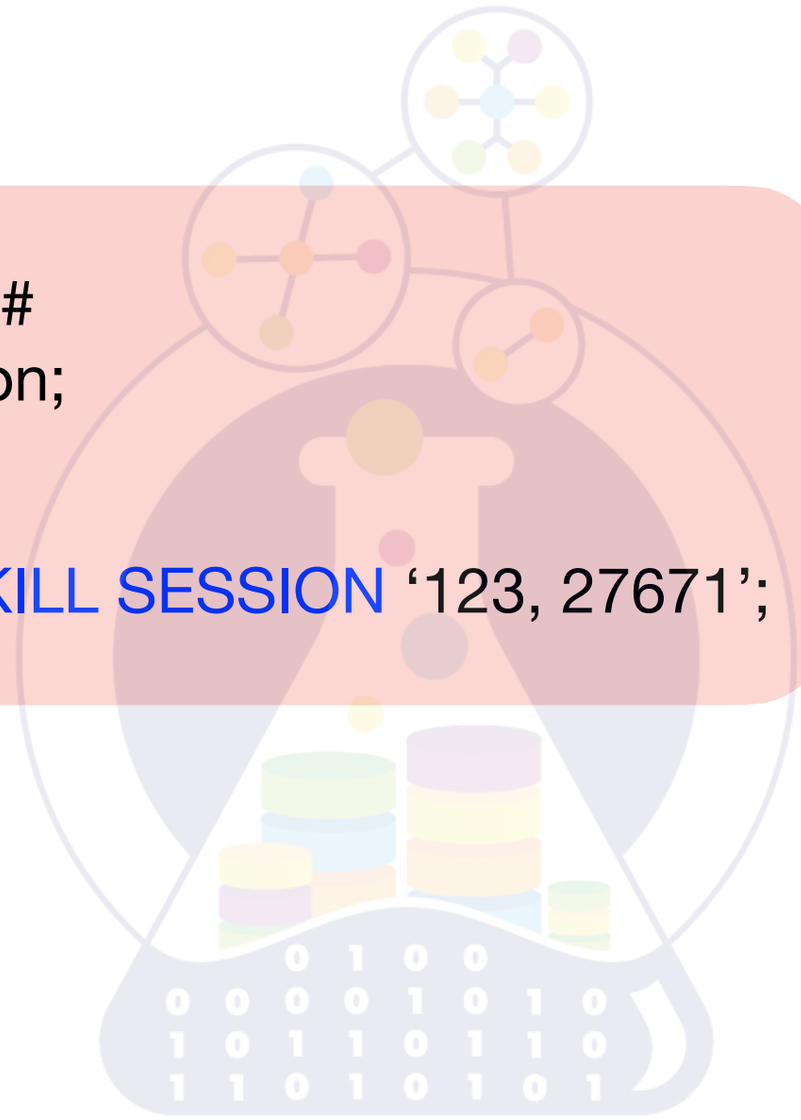


PostgreSQL, Oracle, MySQL

```
SELECT pid, query
    FROM pg_stat_activity;


SELECT pg_terminate_backend(32767);
```

```
SELECT sid, serial#
    FROM v$session;

ALTER SYSTEM KILL SESSION ‘123, 27671’;
```

```
SHOW PROCESSLIST;

KILL 71463;
```



COMMIT;

Azat Yakupov