Transaction parameters, multi-versioning, some transaction internals

Vlad Khorsun, Firebird Project,
Dmitry Kuzmenko, IBSurgeon

Firebird Conference 2019

Berlin, 17-19 October

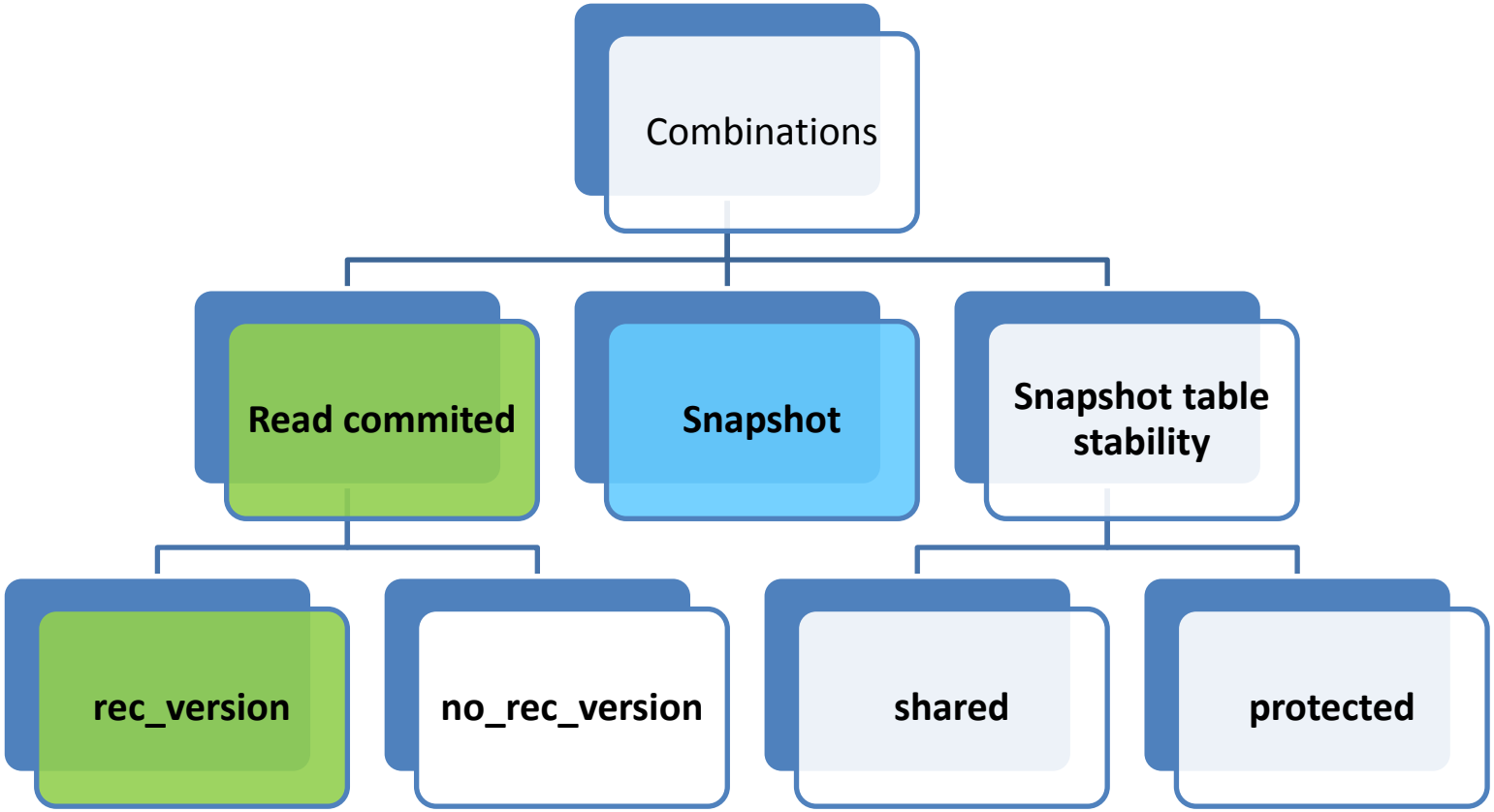
IBSurgeon

All options

- **SET TRANSACTION**
  - [**READ WRITE** | READ ONLY]
  - [**WAIT** | NO WAIT]
  - [[ISOLATION LEVEL]
    {**SNAPSHOT** [TABLE STABILITY]
    | READ COMMITTED [[NO] RECORD_VERSION]}]
  - [RESERVING <reserving_clause>]
    <reserving_clause> = table [, table …]
    [FOR [SHARED | PROTECTED] {READ | WRITE}]
    [, <reserving_clause>]

[LOCK TIMEOUT <seconds>]

Transaction's parameters



SNAPSHOT/READ COMMITED/SNAPSHOT TABLE STABILITY
WAIT/NO WAIT
READ WRITE / READ ONLY

Transaction Parameter Block

- Transaction Parameter Block (TPB)
- TPB specifies transaction's parameters when client application starts transaction
- isc_tpb_*

Firebird API Equivalents

- READ WRITE = isc_tpb_write
- READ ONLY = isc_tpb_read
- WAIT = isc_tpb_wait
- LOCK TIMEOUT = isc_tpb_lock_timeout
- NO WAIT = isc_tpb_nowait
- SNAPSHOT = isc_tpb_concurrency
- READ COMMITTED = isc_tpb_read_committed
  - NO RECORD VERSION = isc_tpb_no_rec_version
  - RECORD VERSION = isc_tpb_rec_version
- SNAPSHOT TABLE STABILITY = isc_tpb_consistency
  - isc_tpb_lock_read, isc_tpb_lock_write, isc_tpb_shared, isc_tpb_exclusive

mon$transactions

- **MON$TRANSACTION_ID - transaction ID**
- **MON$ATTACHMENT_ID - attachment ID**
- **MON$STATE - transaction state**
  - **0: idle**
  - **1: active**
- **MON$TIMESTAMP - transaction start date/time**
- **MON$TOP_TRANSACTION top transaction MON$OLDEST_TRANSACTION - local OIT number**
- **MON$OLDEST_ACTIVE - local OAT number**
- **MON$ISOLATION_MODE - isolation mode**
  - **0: consistency**
  - **1: concurrency**
  - **2: read committed record version**
  - **3: read committed no record version**
- **MON$LOCK_TIMEOUT - lock timeout**
  - **0: no wait**
  - **1: infinite wait**
  - **N: timeout N**
- **MON$READ_ONLY - read-only flag 0/1**
- **MON$AUTO_COMMIT - auto-commit flag**
- **MON$AUTO_UNDO - auto-undo flag**
- **MON$STAT_ID - statistics ID**

Default (API, some components)

- wait
- write
- concurrency

- READ WRITE WAIT SNAPSHOT

- Surprise for the novice developer, when application “does not see changes made to the database until application restart”

Read Committed

- write
- nowait
- read_committed
- rec_version

- Typical **Read Committed** isolation level, allowing to read other concurrent committed changes

Read Read Committed

- Introduced in InterBase 6.0
  - All Firebird versions support it
- Marked committed on its start, but visible in mon$transactions

- mon$transactions
  mon$timestamp = datetime when transaction started
  mon$isolation_mode = 2
  mon$read_only = 1

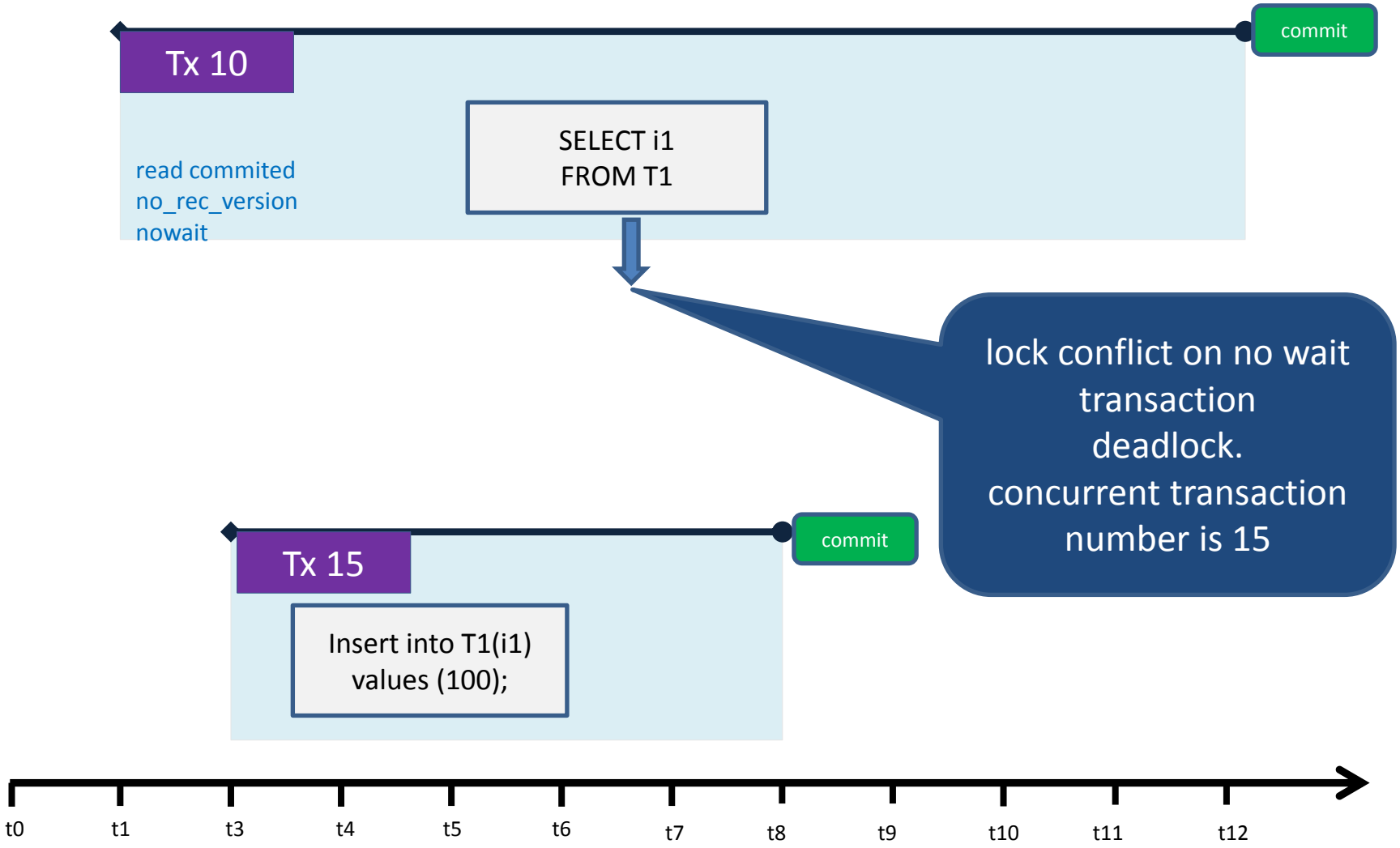
- Can continue (be active) forever, not affecting garbage collection (not holding versions to be garbage collected)

Read commited no_rec_version

- write
- nowait
- read_committed
- [no_rec_version] **by default** in many drivers

- Shows DEADLOCK when trying to read changed but not committed data
  - Can be used to check for not yet committed data

Read commited no_rec_version



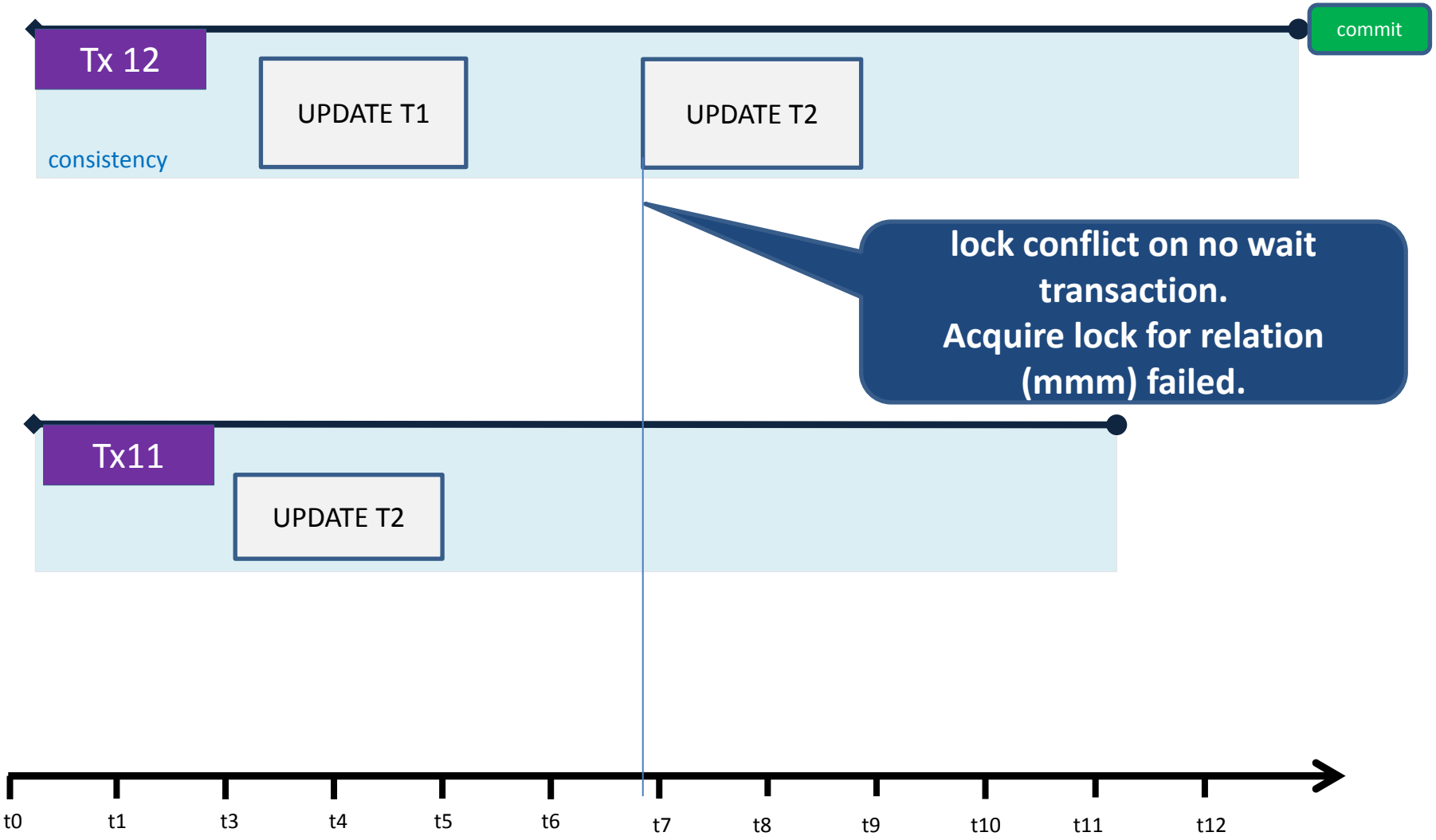
SNAPSHOT TABLE STABILITY

- Locks table (whole) on first access
- With TABLE RESERVATION option - reserves specified tables on transaction start
  - Nowait causes deadlock at start, if other transactions reads or changes these tables
  - Wait causes transaction to wait for other transactions releasing locks

- consistency
  lock_read=CUSTOMERS
  lock_write=ORDERS
  exclusive

- All transactions can read CUSTOMERS, except those who try lock CUSTOMERS in shared_write or protected_write
- Nobody can read ORDERS

Lock conflicts with SNAPSHOT TABLE STABILITY

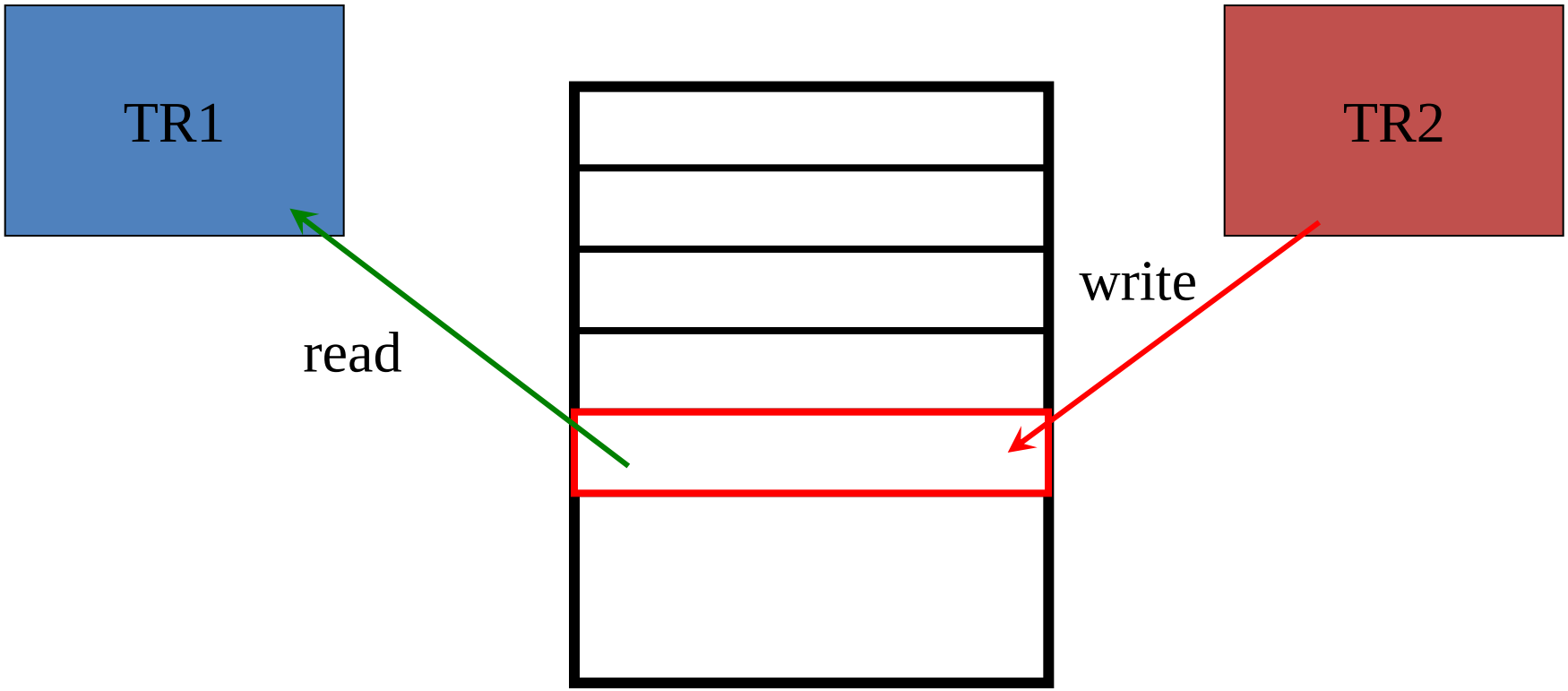


- consistency
  lock_write=CUSTOMERS
  shared
  lock_write=ORDERS
  exclusive

- Table CUSTOMERS can be changed only by read_committed and concurrency transactions

MULTI-VERSIONING

Locking in lock engines



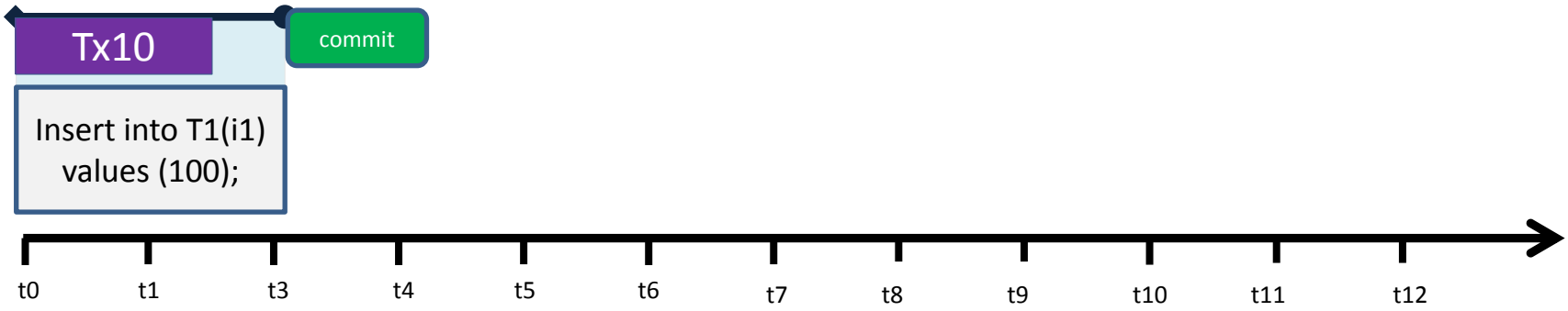
Lock TR1 or lock TR2

What you need to know

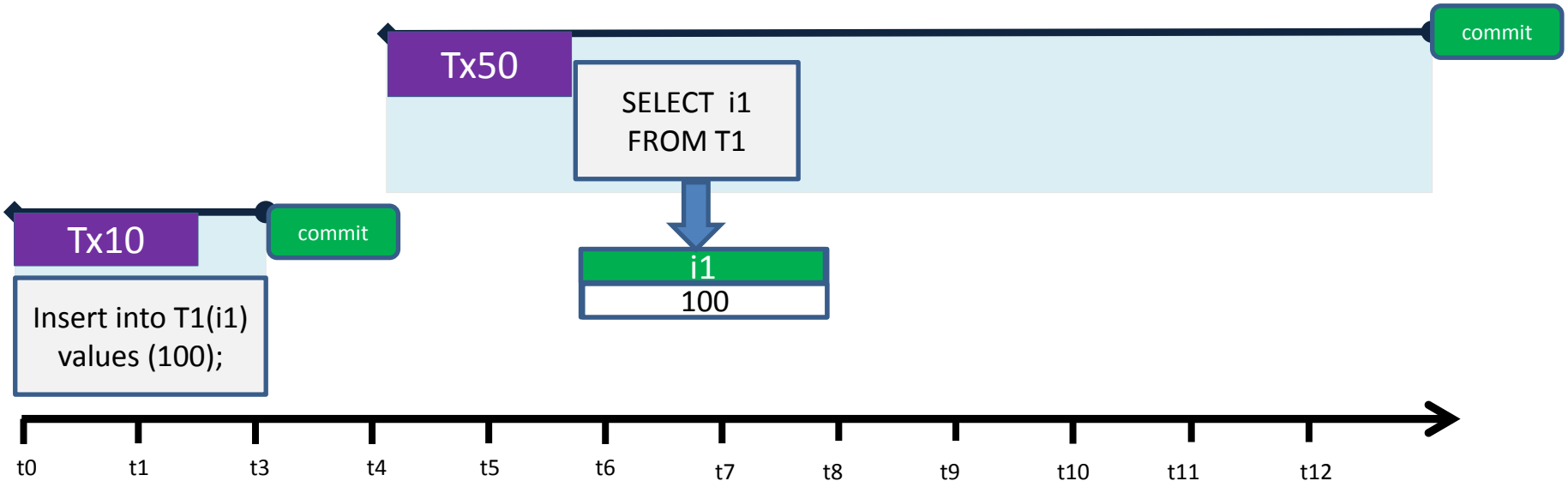
- Everything is done within transaction
- Each transaction get it's own incremented number
  1, 2, 3, … etc
- All changes, made within transaction, are marked by it's number

- ! Since any operation with the database must be done within transaction, Header Page is the most changed page in the database file.

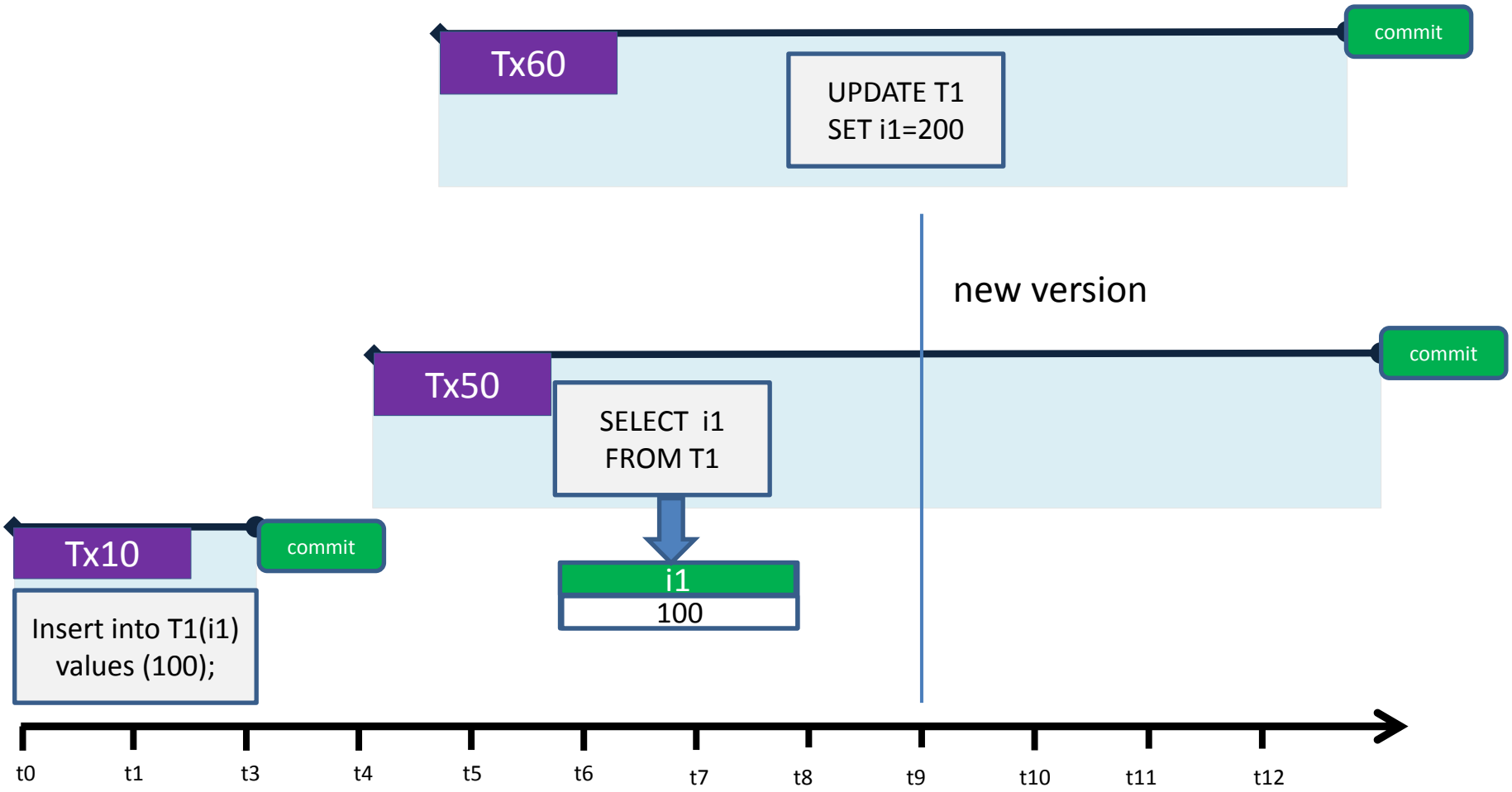
How versions appear



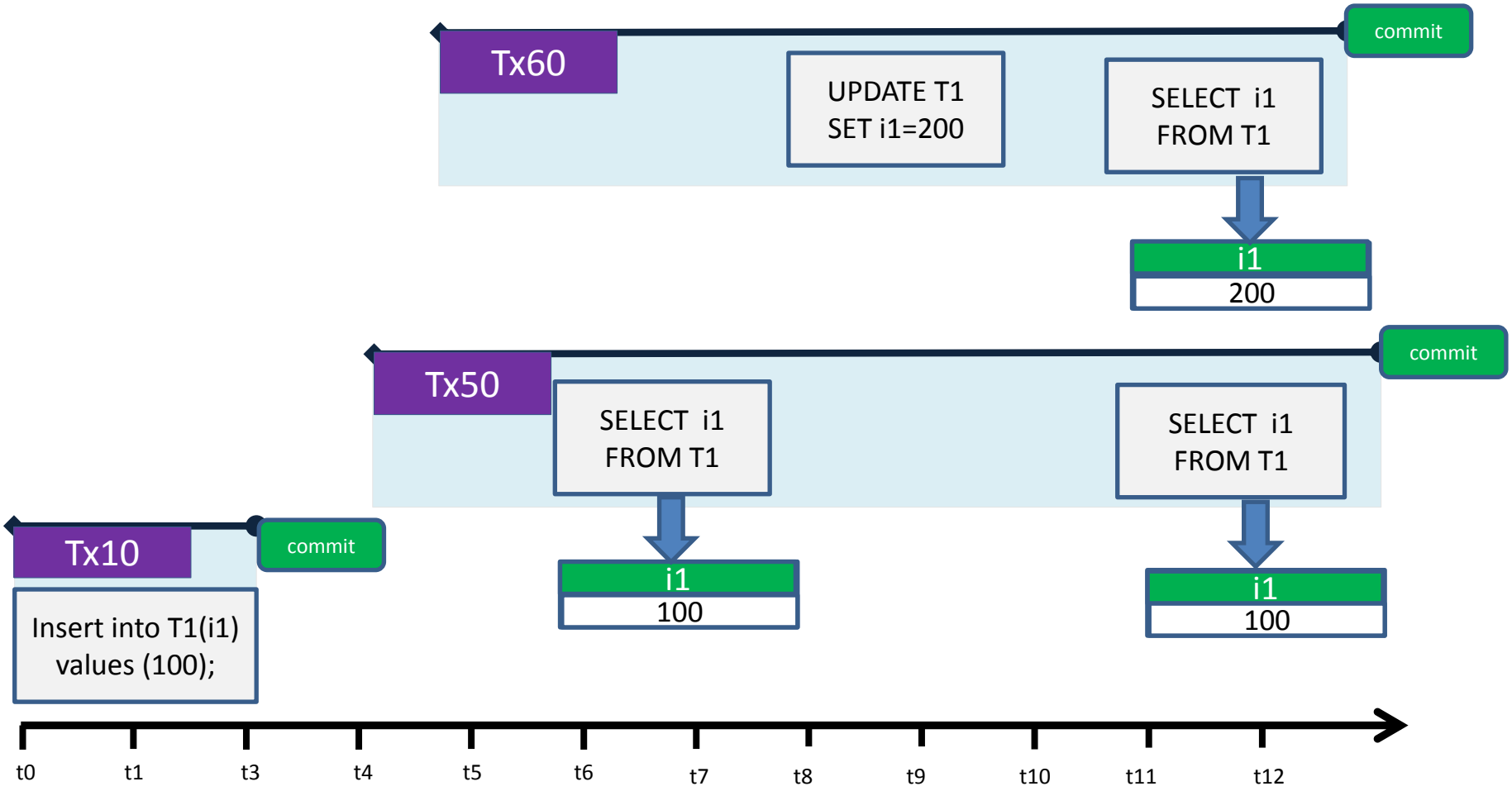
How versions appear



How versions appear

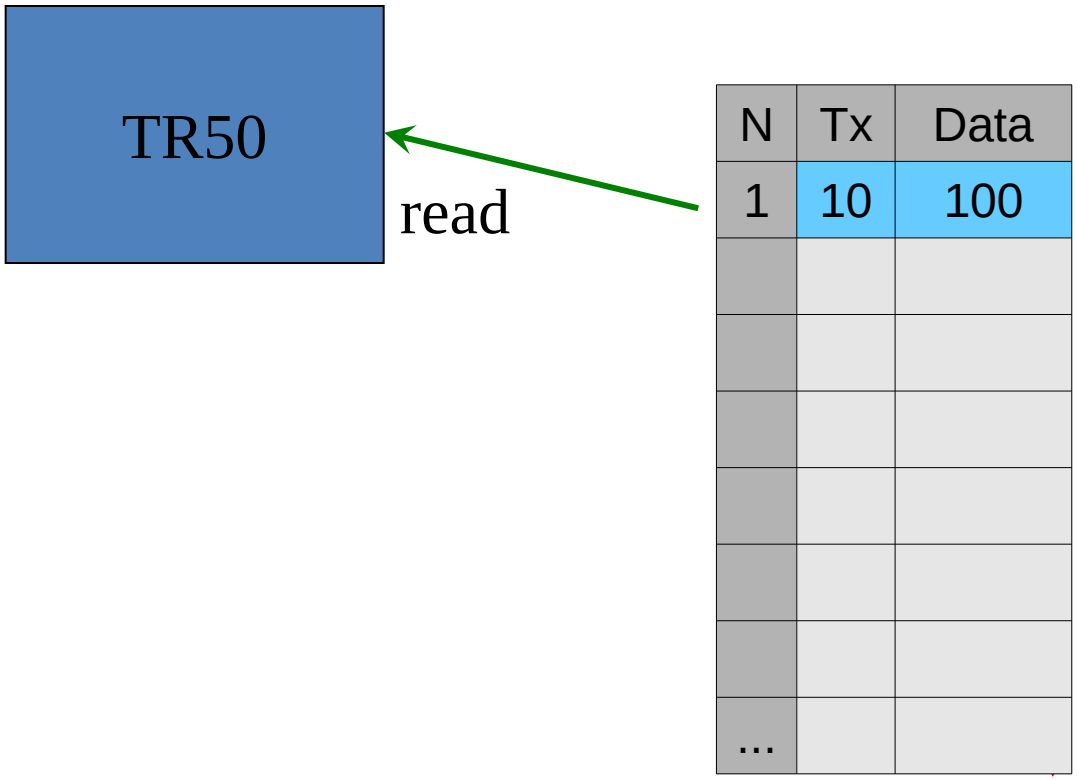


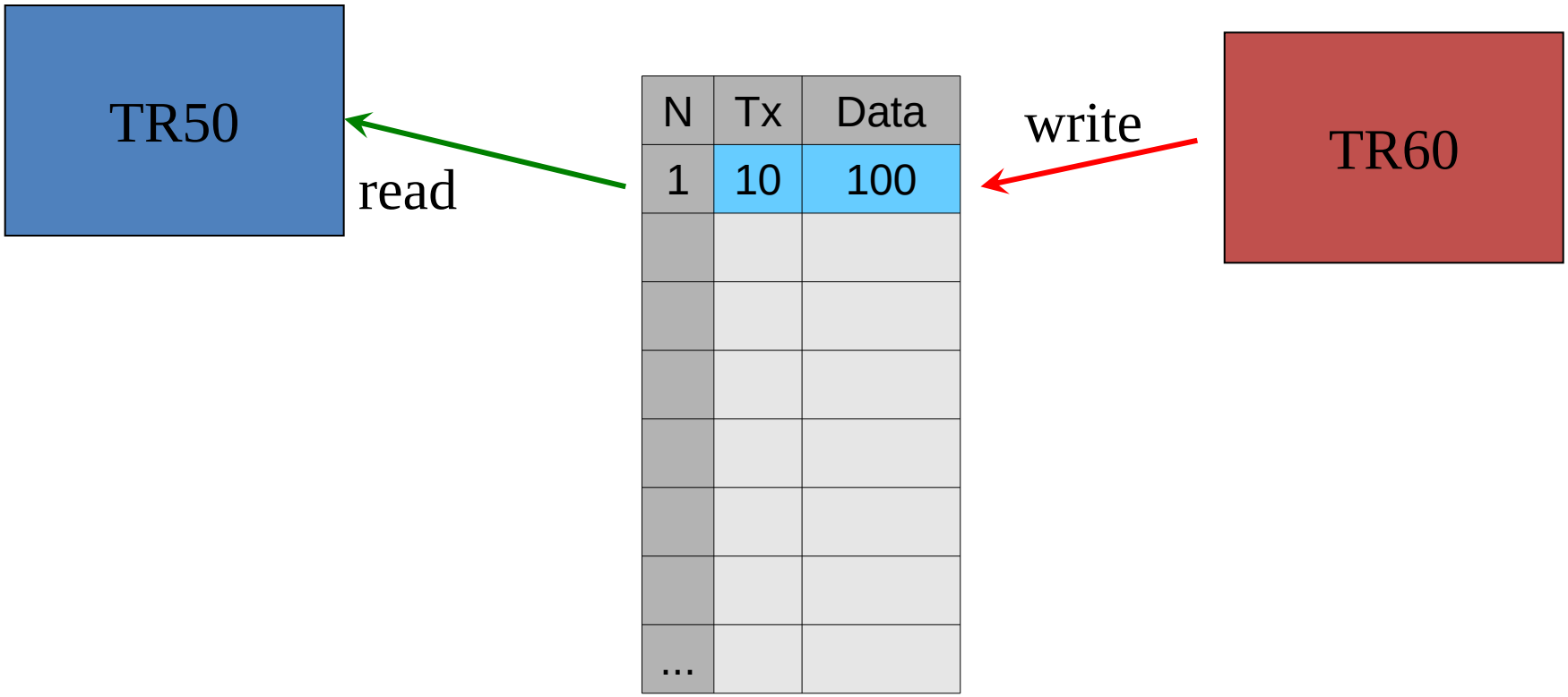
How versions appear

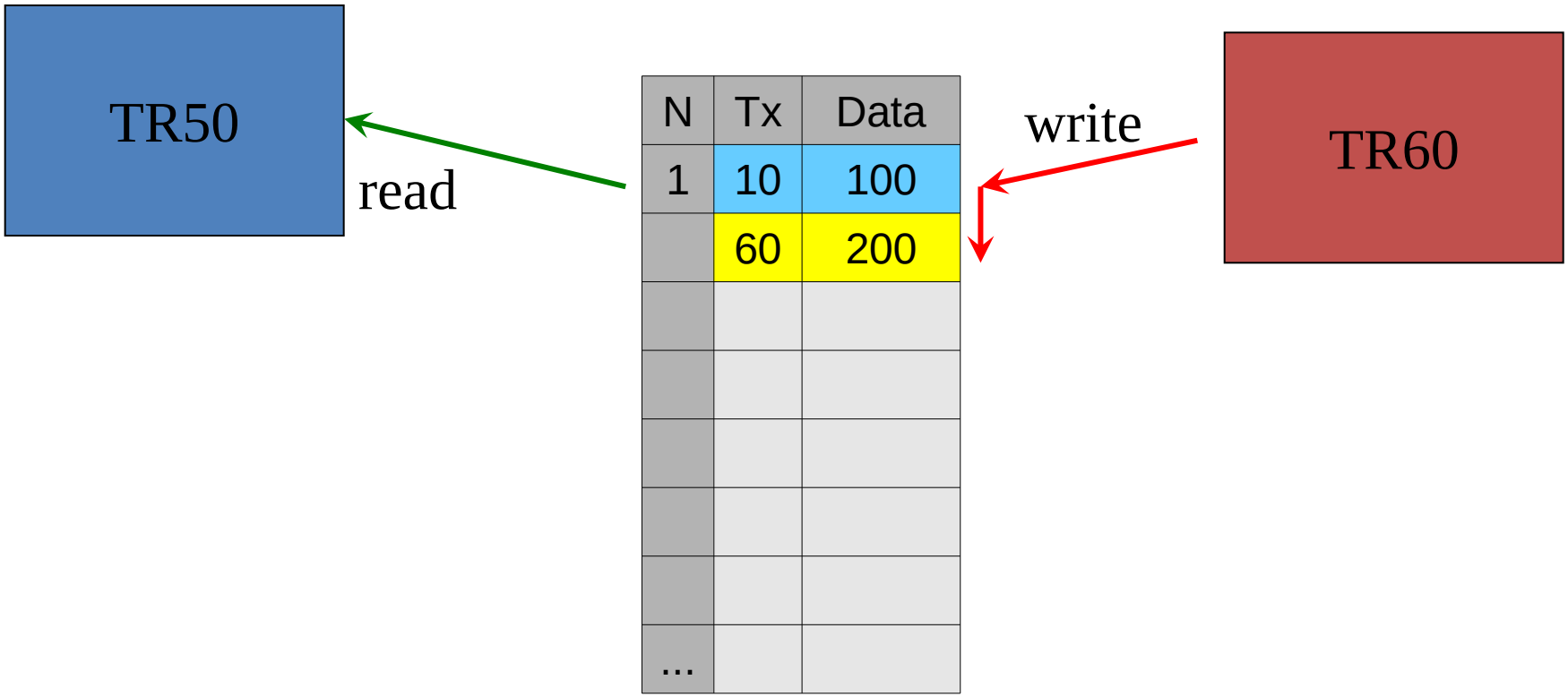


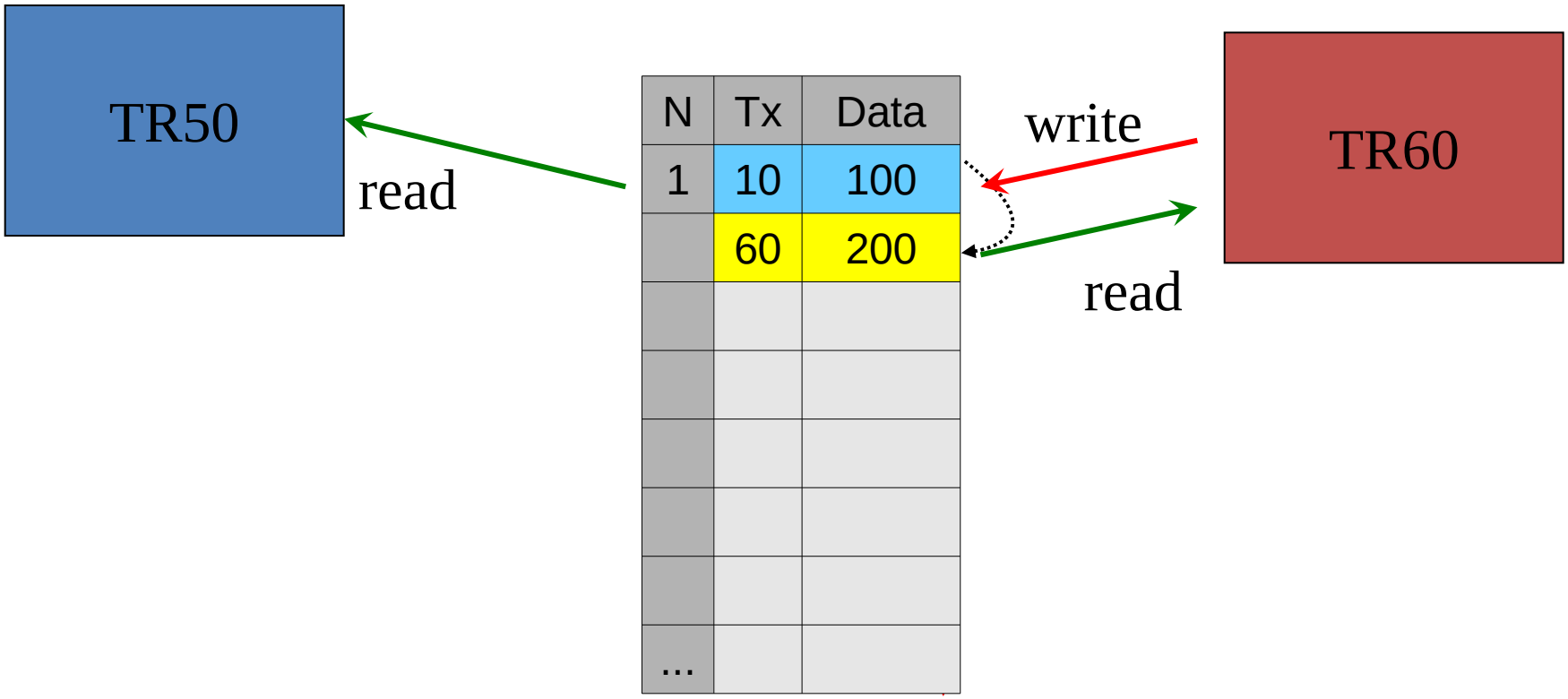
What is stored on data page?

Transaction marks own changes by its number




TR50
read
N
Tx
Data
1
10
100
...
write
TR60


TR50
read
N
Tx
Data
1
10
100
60
200
...
write
TR60


TR50
read
N
Tx
Data
1
10
100
60
200
...
write
read
TR60

Some facts

- No “locks” to lock the record
- Only one non-committed version can exist for the record
  - 2 transactions can’t update same record
- There can be lot of committed versions for one record – up to 1.5 million and more
  - If version size is 20 bytes, 1.5mln versions will occupy 30mb
- Performance degrade is not proportional to the number of versions
- Versions may be needed or not. If not, they can be considered as “garbage”.

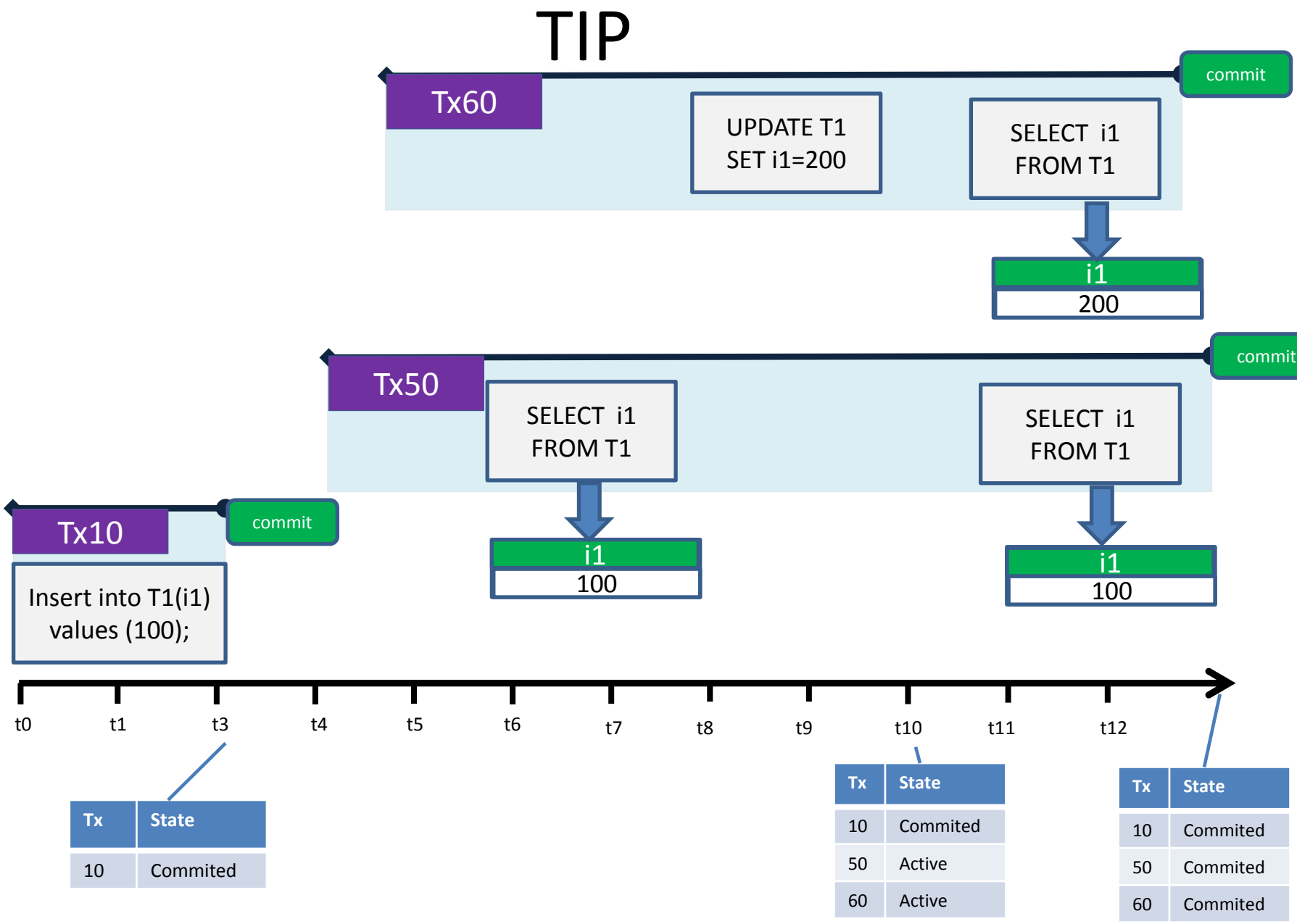
How server knows about transaction states?

- TIP – Transaction Inventory Pages
  - Linear list of transaction states, from 0 to last transaction number
  - Stored in the database

Transaction's states

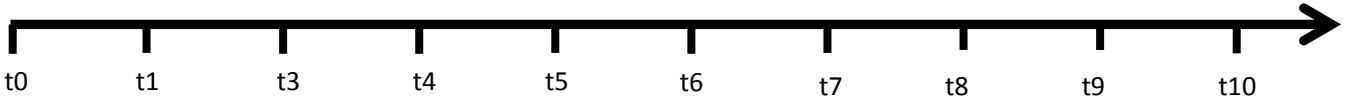
- Each transaction is represented by it's state
  - 00 – Active
  - 01 – Committed
  - 10 – Rolled back
  - 11 – Limbo (distributed 2-phase transactions)
  - 2 bits for each transaction state

TIP contents	
Tx №	Tx state
	…
10	committed
11	committed
12	committed
13	rolled back
14	committed
15	committed
16	committed
17	rolled back
18	active
19	committed
20	active

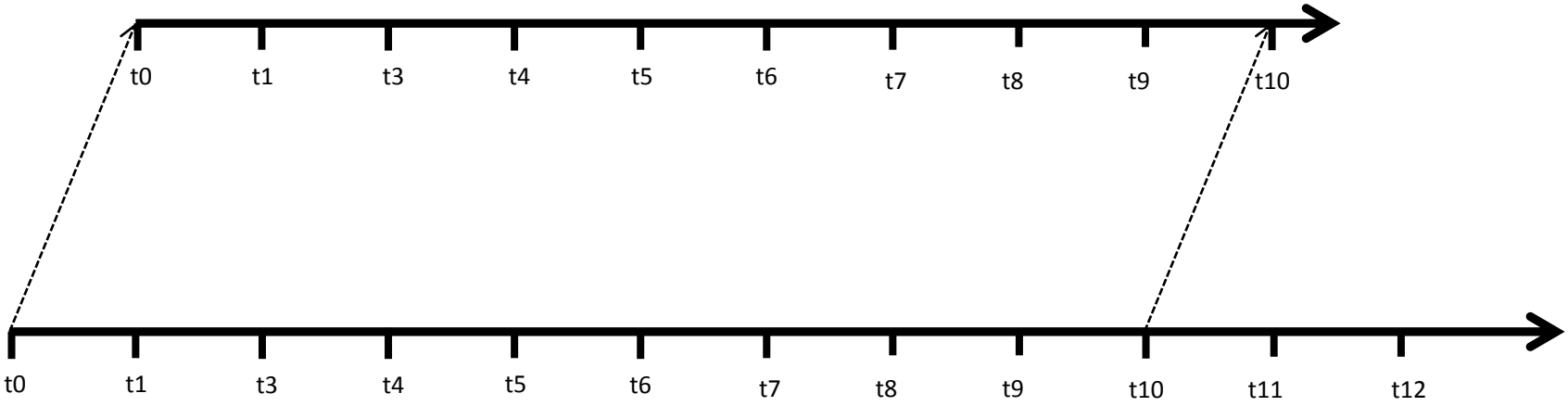

TIP
Tx60
commit
UPDATE T1
SET i1=200
SELECT i1
FROM T1
i1
200
Tx50
commit
SELECT i1
FROM T1
SELECT i1
FROM T1
i1
100
i1
100
Tx10
commit
Insert into T1(i1)
values (100);
t0
t1
t3
t4
t5
t6
t7
t8
t9
t10
t11
t12
Tx
State
10
Commited
Tx
State
10
Commited
50
Active
60
Active
Tx
State
10
Commited
50
Commited
60
Commited

Read Committed and Snapshot

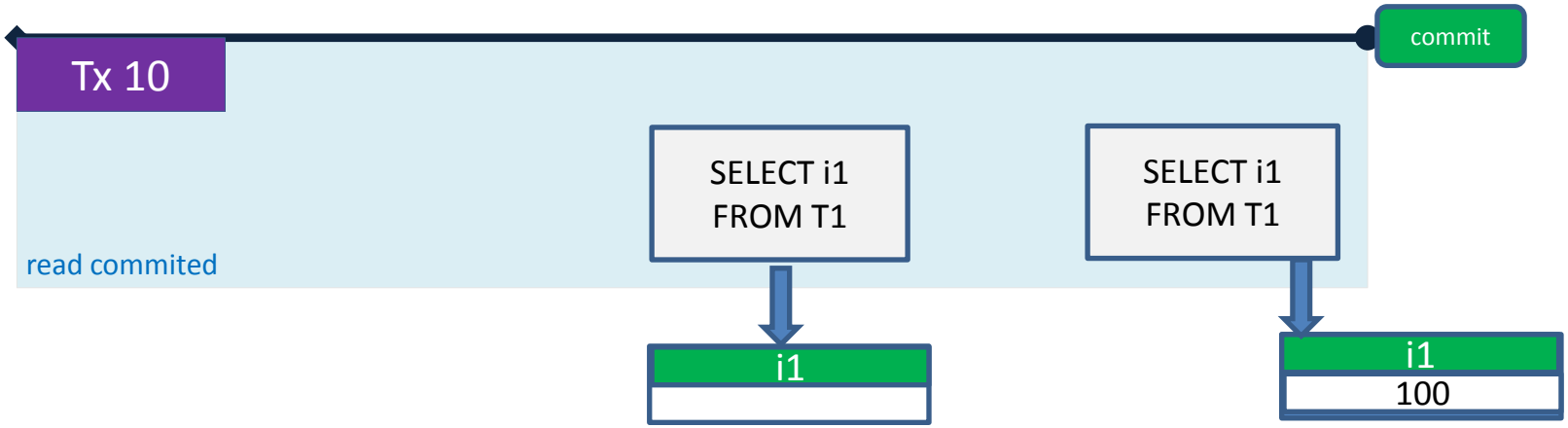
Read Committed transactions “see” global TIP.
That’s why they can read other committed changes

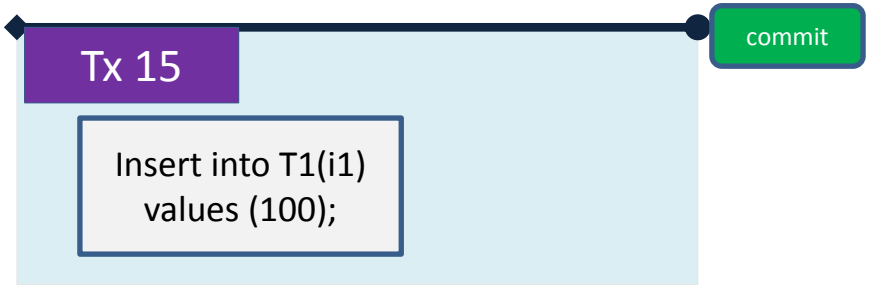


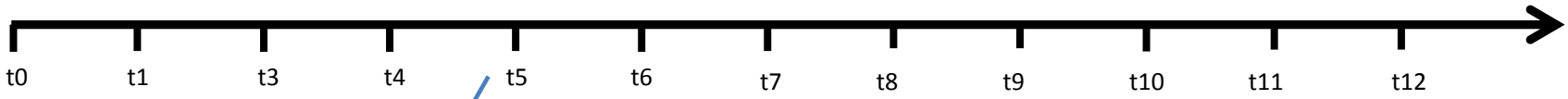
Snapshot copies TIP on it’s start. Thus it does not see any changes made by other committed transactions after snapshot start



TIP for Read commited





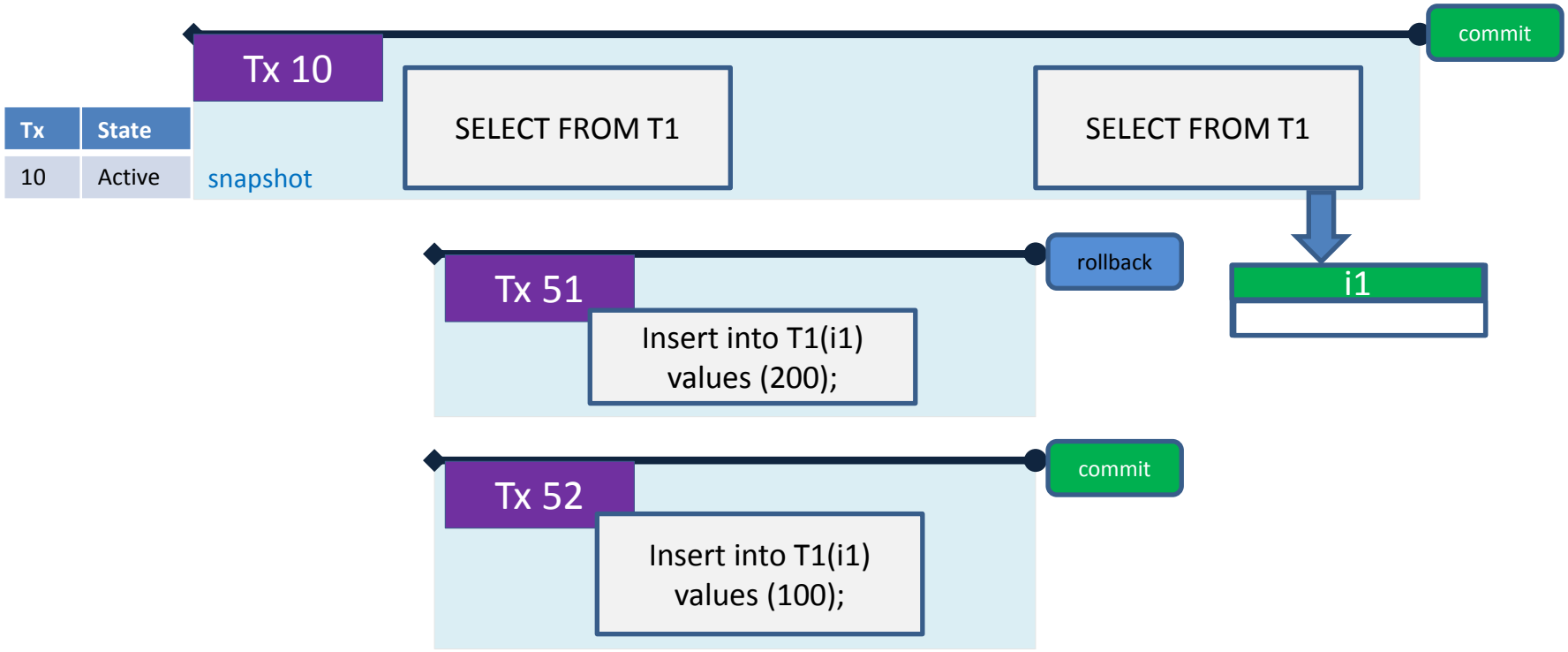


Tx	State
10	Active

Tx	State
10	Active
15	Active

Tx	State
10	Active
15	Commited

TIP for Snapshot



Tx	State
10	Active

Tx	State
10	Active
51	Active
52	Active

Tx	State
10	Active
51	Rollback
52	Commited

What transaction can see?

- Of course, it's own created records and versions
  - Insert, Update, Delete
- If it is **Read Committed**, it can see every changes that was made by committed transactions, because it checks global TIP
- If it is **Snapshot**, it can see own changes and record versions commited to the moment of its start, because it checks it's own copy of TIP

Example

- TABLE have 100 records, committed
- start Read Committed transaction 1
- start transaction 2
- tr2: delete 5 records
- start Snapshot transaction 3
- tr1: insert 4 records

Commit

100

95

100

- How many records will see select count in each transaction?

100+ 5 + 4 = 109

How many versions?

WHERE DO THE TRANSACTIONS COME FROM

Where transactions can be initiated

- In application
- System transaction № 0 –
  - used for DDL changes
  - Reading system queries
- Garbage collector's transaction
  - Read commited read-only
- Transactions for triggers ON CONNECT/ON DISCONNECT
  - concurrency write wait (default snapshot)

Autonomous transactions

- PSQL (triggers, procedures, blocks)
- create trigger tr_connect on connect
  as
  begin
    in **autonomous transaction** do
      insert into log (msg) values ('User ' || current_user || ' connects.');
    if (current_user in (select username from blocked_users)) then
    begin
      in **autonomous transaction** do
      begin
        insert into log (msg) values ('User ' || current_user || ' refused.');
        post_event 'Connection attempt by blocked user.';
      end
      exception ex_baduser;
    end
  end

- Code running in an autonomous transaction **will be committed immediately** upon successful completion, regardless of how the parent transaction finishes.
- Autonomous transactions have the **same isolation level** (and other parameters) as their parent transaction
- Because the autonomous transaction is completely independent of its parent, care must be taken to avoid **deadlocks**
- If an **exception** occurs *within* the autonomous transaction, the work in autonomous transaction **will be rolled back**

Triggers on transaction start/end

- CREATE TRIGGER ABC
  ON TRANSACTION START |
  TRANSACTION COMMIT |
  TRANSACTION ROLLBACK
  as
  begin
  …
  end

- Fired on user and autonomous transactions

- TRANSACTION triggers are executed within the same transaction. The actions taken after an uncaught exception depend on the type:
  - In a START trigger, the exception is reported to the client and the transaction is rolled back.
  - In a COMMIT trigger, the exception is reported, the trigger's actions so far are undone and the commit is canceled.
  - In a ROLLBACK trigger, the exception is not reported and the transaction is rolled back as foreseen.
- you can't start any transaction if a TRANSACTION START trigger causes an exception, so, you may lock DB completely. In this case use isql –nodbtriggers and fix or drop the wrong trigger.

Context

- What is the context?
- 3 namespaces of context
  - SYSTEM – read-only
    - Info about connection and engine
  - USER_SESSION
    - For storing user variables at connection level
  - USER_TRANSACTION
    - For storing user variables at transaction level
- rdb$get_context
- rdb$set_context

Transaction context variables

- rdb$set_context(‘USER_TRANSACTION, ‘abc’, 555)
- …
- Myvar = rdb$get_context(‘USER_TRANSACTION’, ‘abc’)

- select rdb$get_context('SYSTEM', ‘ISOLATION_LEVEL') from rdb$database
  - READ COMMITED
  - SNAPSHOT
  - CONSISTENCY
- select rdb$get_context('SYSTEM', ‘TRANSACTION_ID') from rdb$database
  - CURRENT_TRANSACTION

CURRENT_TRANSACTION usage

- SELECT FROM MON$STATEMENTS

  WHERE MON$TRANSACTION_ID = CURRENT_TRANSACTION

NEXT …