Statement-level read consistency in read-committed transactions

Non-consistent reads problem

- Attachment 1

```
INSERT INTO T …
INSERT INTO T …
--  1000 recs
COMMIT;
INSERT INTO T …
INSERT INTO T …
--  1000 recs
COMMIT;
...
```

- Attachment 2

```
SELECT COUNT(*) FROM T;
SELECT COUNT(*) FROM T;
...
```

Non-consistent reads problem

- Expected results

```
1000
2000
…
N * 1000
```

- Actual results

```
0
200
1000
1823
3000
4028
…
```

Non-consistent reads problem

- Attachment 1

```
INSERT recs 0...199
INSERT recs 200...399
INSERT recs 400...599
INSERT recs 600...799
INSERT recs 800...999
COMMIT;
```

- Attachment 2

```
READ recs 0...
   ...
   ...
   799
   not committed – ignored

READ recs 800...999
   committed – counted
RESULT = 200
```

Non-consistent reads problem

- What is required to solve the issue ?
  - Every top-level statement in read-committed transaction require own stable view of database (snapshot)
  - Every user cursor in read-committed transaction also require own snapshot (until closed)

Database snapshots: traditional

- Database snapshot allows to know state of any transaction when snapshot created
  - All transaction states are recorded at Transaction Inventory (TIP)
  - Copy of TIP created at some moment allows later to know state of any given transaction at that moment
- If some transaction state is known as “active” in any used snapshot, there should be guarantee that engine could read records committed before this transaction changed it.
  - Special database marker OST used as garbage collection threshold

Non-consistent reads : solution

- Obvious solution: use private TIP copy
  - Every top-level statement should take copy of active part of TIP and use it instead of live TIP data
    - same as snapshot transactions

Non-consistent reads : solution

- Obvious solution: drawbacks
  - To make it work at read-committed read-only transactions it is necessary to not allow OST to move above RC RO transaction number
    - Long lived RC RO transaction will block GC
  - Cost of creating private TIP copy for every statement is non-zero
    - Could affect performance
  - RC transactions will use private copy of TIP instead of shared TIP cache
    - Memory usage will grow

Database snapshots: commits order

- It is enough to know order of commits to know state of any transaction when snapshot created:
  - If other tx is active (dead) in TIP, consider it as active (dead), obviously
  - If other tx is committed in TIP - we should know when it was committed:
    - before our snapshot created – consider it as committed
    - after our snapshot created – consider it as active

Database snapshots: commits order

- Commits order:
  - New global per-database counter: Commit Number (CN)
    - In-memory only, no need to store in database
    - Initialized when database is started
    - When any transaction is committed, global Commit Number is incremented and its value is assotiated with transaction (i.e. we just defined “transaction commit number“, or transaction CN)

Database snapshots: commits order

- Possible values of transaction Commit Number
  - Transaction is active : CN = 0
    - **CN_ACTIVE**
  - Transaction is in limbo: CN = MAX_TRA_NUM - 1
    - **CN_LIMBO**
  - Dead transaction: CN = MAX_TRA_NUM - 2
    - **CN_DEAD**
  - Transactions committed before database started (i.e. older than OIT) : CN = 1
    - **CN_PREHISTORIC**
  - Transactions committed while database works:
    - **CN_PREHISTORIC < CN < CN_DEAD**

Database snapshots: commits order

- Database snapshot is defined as
  - Private value of global Commit Number at moment when database snapshot is created, and
  - Common list of all transactions with assotiated CN's
    - Transactions older than OIT are known to be committed thus not included in this list

Database snapshots: commits order

- Database snapshot could be created
  - For every transaction
    - Useful for snapshot (concurrency) transactions
  - For every statement and for every cursor
    - Useful for read-committed transactions
    - Allows to solve statement-level read consistency probem

Database snapshots: commits order

- In-memory cost of database snapshot:
  - Per-snapshot - just a number (64 bit)
  - Per-database
    - List of all active snapshots
    - List of all transactions between OIT and Next with assotiated commit numbers

Database snapshots: commits order

Memory usage comparison		
	Traditional	**Commits Order**
TIP on disk	Array of 2-bit states for every transaction	Array of 2-bit states for every transaction
TIP cache in memory	Array of 2-bit states for every transaction since OIT	Array of 64-bit Commit Numbers of every transaction since OIT
Private snapshot	Array of 2-bit states of transactions between OIT and Next	Single 64-bit Commit Number
List of active snapshots		Array of 64-bit Commit Numbers

Database snapshots: commits order

- Record visibility rule
  - Compare CN of our snapshot (CN_SNAP) and CN of transaction which created record (CN_REC):

    CN_REC == CN_ACTIVE,

    CN_REC == CN_LIMBO

    - Invisible

    CN_REC == CN_DEAD

    - Backout dead version (or read back version) and repeat

    CN_REC > CN_SNAP

    - Invisible

    CN_REC <= CN_SNAP

    - Visible

Database snapshots: commits order

- Record visibility rule: consequence
  - If some snapshot CN could see some record version then all snapshots with numbers > CN also could see same record version
- Garbage collection rule
  - If all existing snapshots could see some record version then all it backversions could be removed, or
  - If oldest active snapshot could see some record version then all it backversions could be removed

Long running transactions

Sequence of actions

1	Tx 10 start
2	Tx 10 insert
3	Tx 10 commit
4	Tx 11 start
5	Tx 12 start
6	Tx 12 update
7	Tx 12 commit
8	Tx 13 start
9	Tx 13 update
10	Tx 13 commit
11	Tx 14 start
12	Tx 14 update
13	Tx 14 commit
14	Tx 15 start

TIP

Tx	State
10	committed
11	active
12	committed
13	committed
14	committed
15	active

Oldest Snapshot

Tx 11

Tx 15

Tx 14 → Tx 13 → Tx 12 → Tx 10

Not needed versions, can't be removed !

Long running transactions

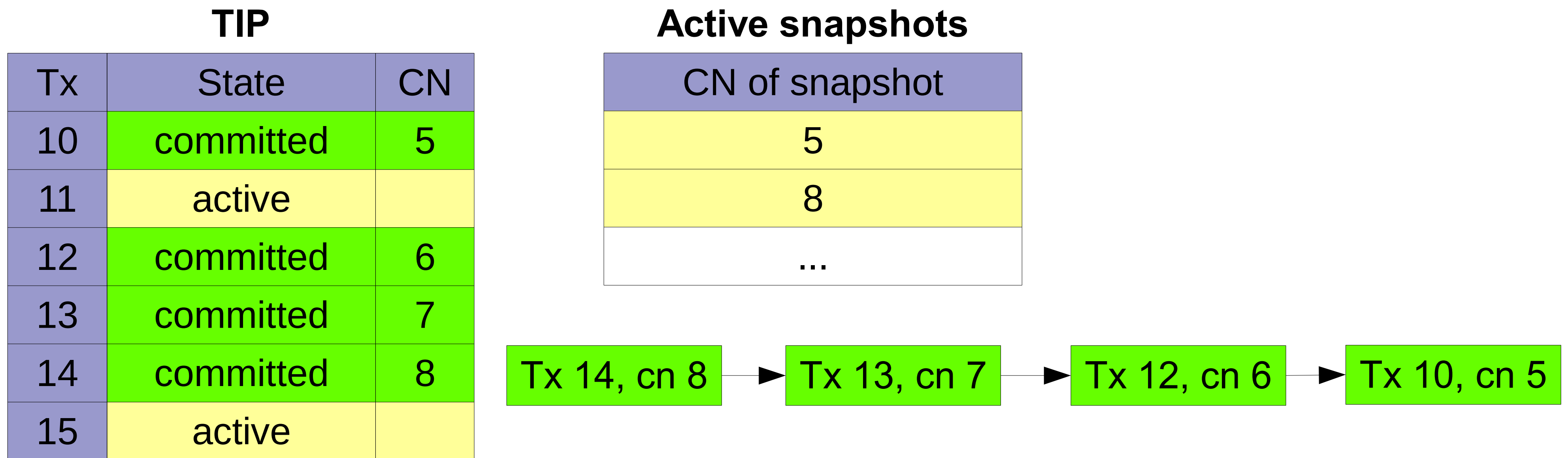
Sequence of actions

3	Tx 10 commit, CN = 5
4	Tx 11 start
	create snapshot 5
5	Tx 12 start
6	Tx 12 update
7	Tx 12 commit, CN = 6
8	Tx 13 start
9	Tx 13 update
10	Tx 13 commit, CN = 7
11	Tx 14 start
12	Tx 14 update
13	Tx 14 commit, CN = 8
14	Tx 15 start
	create snapshot 8

TIP

Tx	State	CN
10	committed	5
11	active	
12	committed	6
13	committed	7
14	committed	8
15	active	

Long running transactions



TIP

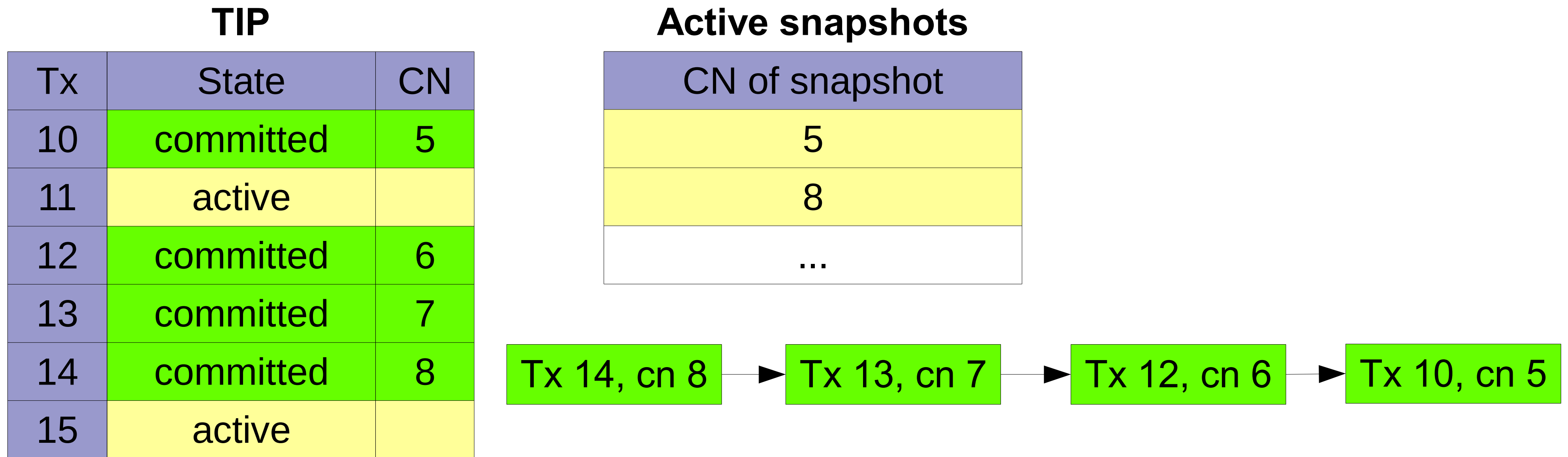
Tx	State	CN
10	committed	5
11	active	
12	committed	6
13	committed	7
14	committed	8
15	active	

Active snapshots

CN of snapshot
5
8
...

- Snapshots list is sorted
  - First entry is oldest snapshot
- Which snapshot could see which record version ?
  - CN_REC <= CN_SNAP

Long running transactions



TIP

Tx	State	CN
10	committed	5
11	active	
12	committed	6
13	committed	7
14	committed	8
15	active	

Active snapshots

CN of snapshot
5
8
...

- Interesting value: oldest active snapshot which could see given record version
- If few versions in a chain have the same (see above) then all versions except of first one could be removed !

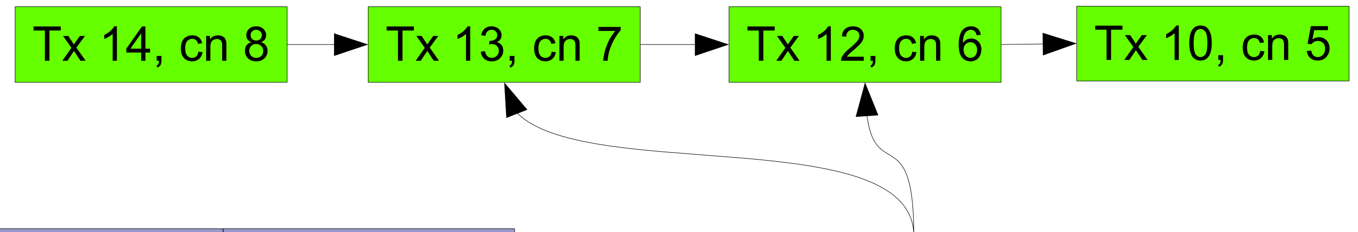
Long running transactions

TIP

Tx	State	CN
10	committed	5
11	active	
12	committed	6
13	committed	7
14	committed	8
15	active	

Active snapshots

CN of snapshot
5
8
...



Record versions chain	Oldest CN could see the version	Can be removed
Tx 14, cn 8	8	No
Tx 13, cn 7	8	Yes
Tx 12, cn 6	8	Yes
Tx 10, cn 5	5	No

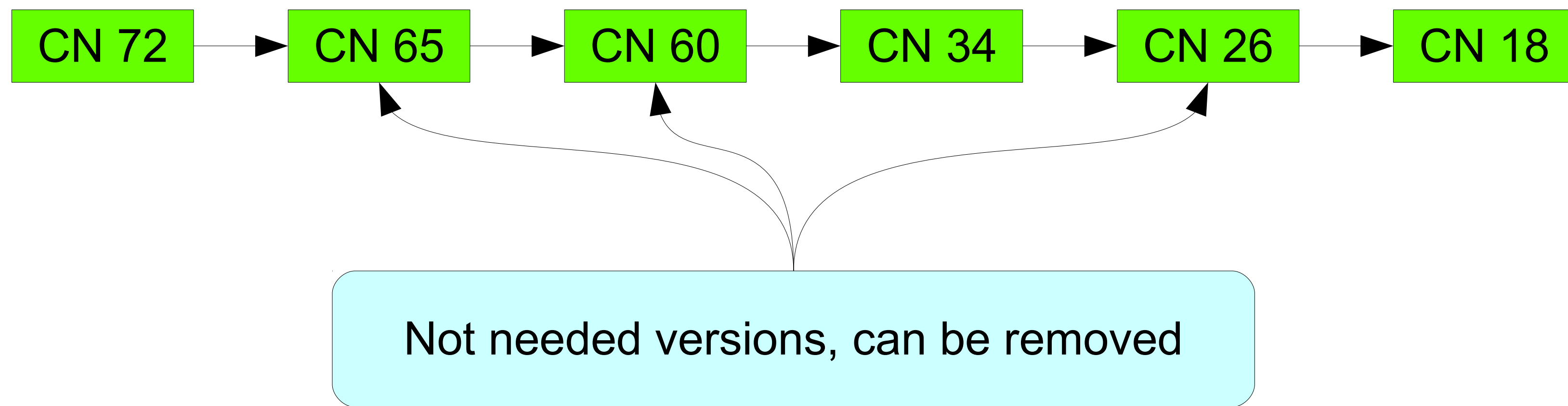
Intermediate record versions

Active snapshots

CN of snapshot
23
48
54
57
78
...

Visibility of record versions

Record versions chain	Oldest CN could see version	Could be removed ?
Tx 345, cn 72	78	No
Tx 256, cn 65	78	Yes
Tx 287, cn 60	78	Yes
Tx 148, cn 34	48	No
Tx 124, cn 26	48	Yes
Tx 103, cn 18	23	No



Database snapshots: commits order

- Conclusions
  - Statement-level read consistency problem will be solved
  - Long running read-committed transactions will not block garbage collection at all
  - Long running snapshot transactions allow GC to clean really unneeded versions early
  - Long running statements (open cursors) in read-committed transactions – same as snapshot transactions
  - No need to mark read-committed read-only transaction as committed at start

THANK YOU FOR ATTENTION

Questions ?

[Firebird official web site]()

[Firebird tracker]()

[hvlad@users.sf.net]()