New transaction’s features and changes in garbage collection in Firebird 4

Firebird Conference 2019

Berlin, 17-19 October

IBPhoenix

YOUR PREMIER SOURCE OF FIREBIRD SUPPORT

IBSurgeon

MOSCOW EXCHANGE

Fast Reports

Reporting must be fast!

IB Expert

REDSOFT

Agenda

- Database snapshot
    - Traditional
    - Commits order
- Intermediate GC
- Read Committed Read Consistency
    - Update conflicts handling
    - Read only transactions
- Shared snapshots

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

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 associated with transaction (i.e. we just defined “Transaction Commit Number“, or Transaction CN)

Database snapshots: commits order

- Commits order:
  - New global per-database counter: Commit Number (CN)
    - Current value could be queried using new context variable *“GLOBAL_CN”* in *“SYSTEM”* context:

```
SELECT RDB$GET_CONTEXT(‘SYSTEM’, ‘GLOBAL_CN’)
  FROM RDB$DATABASE
```

Database snapshots: commits order

- Possible values of transaction Commit Number
  - Transaction is active:
    - **CN_ACTIVE = 0**
  - Transactions committed before database started (i.e. older than OIT):
    - **CN_PREHISTORIC = 1**
  - Transaction is in limbo:
    - **CN_LIMBO = MAX_UINT64 - 1**
  - Dead transaction:
    - **CN_DEAD = MAX_UINT64 - 2**
  - Transactions committed while database works:
    - **CN_PREHISTORIC < CN < CN_DEAD**

Database snapshots: commits order

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

Database snapshots: commits order

- List of “interesting” transactions with its states and commit numbers
  - Array located in shared memory
    - Available for all Firebird processes
  - Item index is transaction’s number
  - Item value is transaction’s CN
  - Whole array split on blocks of fixed size
    - new setting *TipCacheBlockSize* in firebird.conf
    - 4MB by default, fits 512K items
  - Whole array keeps values between OIT and Next
    - blocks dynamically allocated and released

Database snapshots: commits order

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

Database snapshots: commits order

- List of all active database snapshots
  - For garbage collection purposes
  - List of items <attachment_id, snapshot_number>
    - 16 bytes
  - Allocated in shared memory
  - Fixed size
    - new setting *SnapshotsMemSize* in firebird.conf
    - 64KB by default, fits more than 4000 snapshots

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 16-byte items, 64KB by default

Database snapshots: commits order

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

    CN_REC == CN_ACTIVE,

    CN_REC == CN_LIMBO

    - Invisible

    CN_REC == CN_DEAD

    - Back out 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

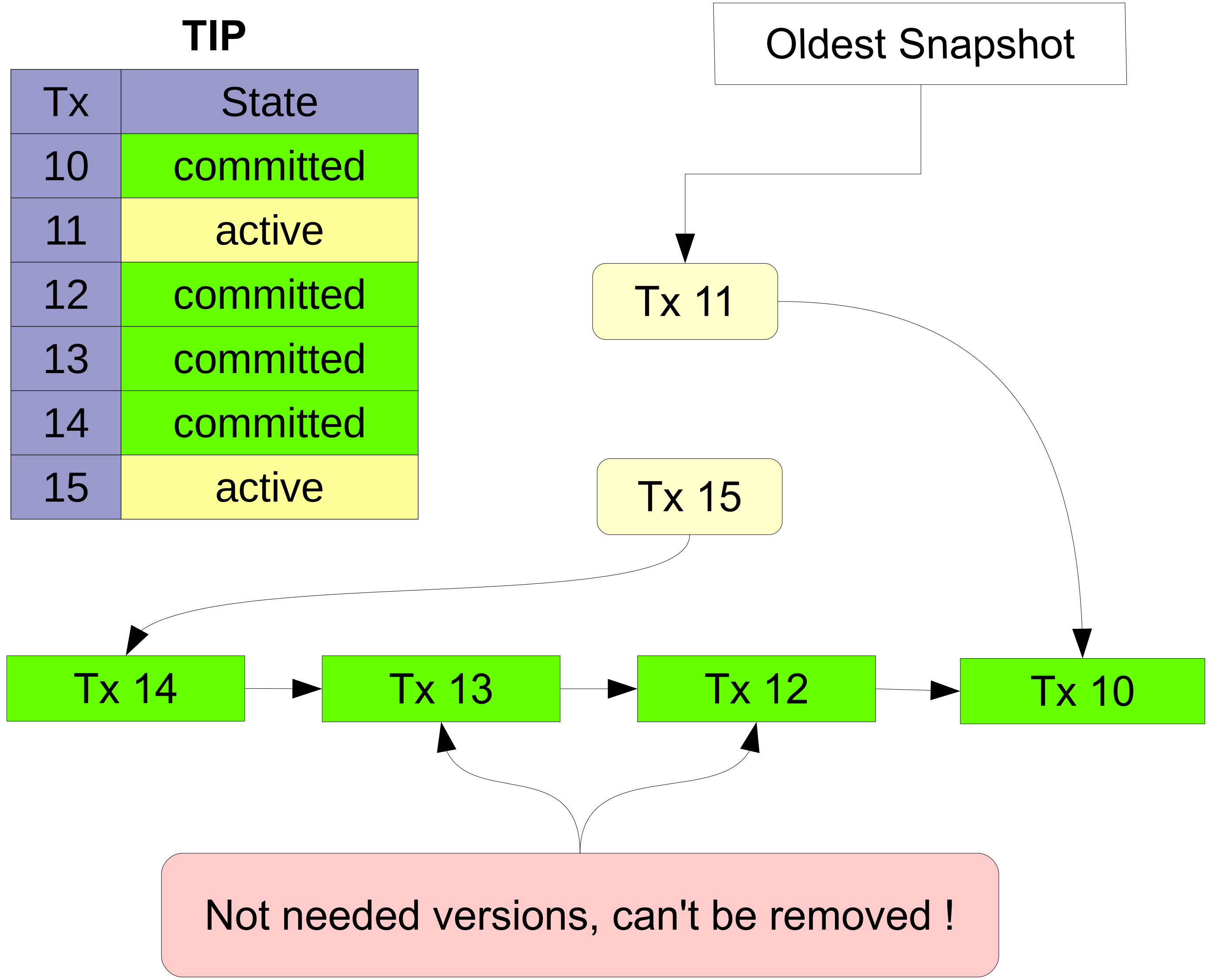
Long running transactions

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



Not needed versions, can't be removed !

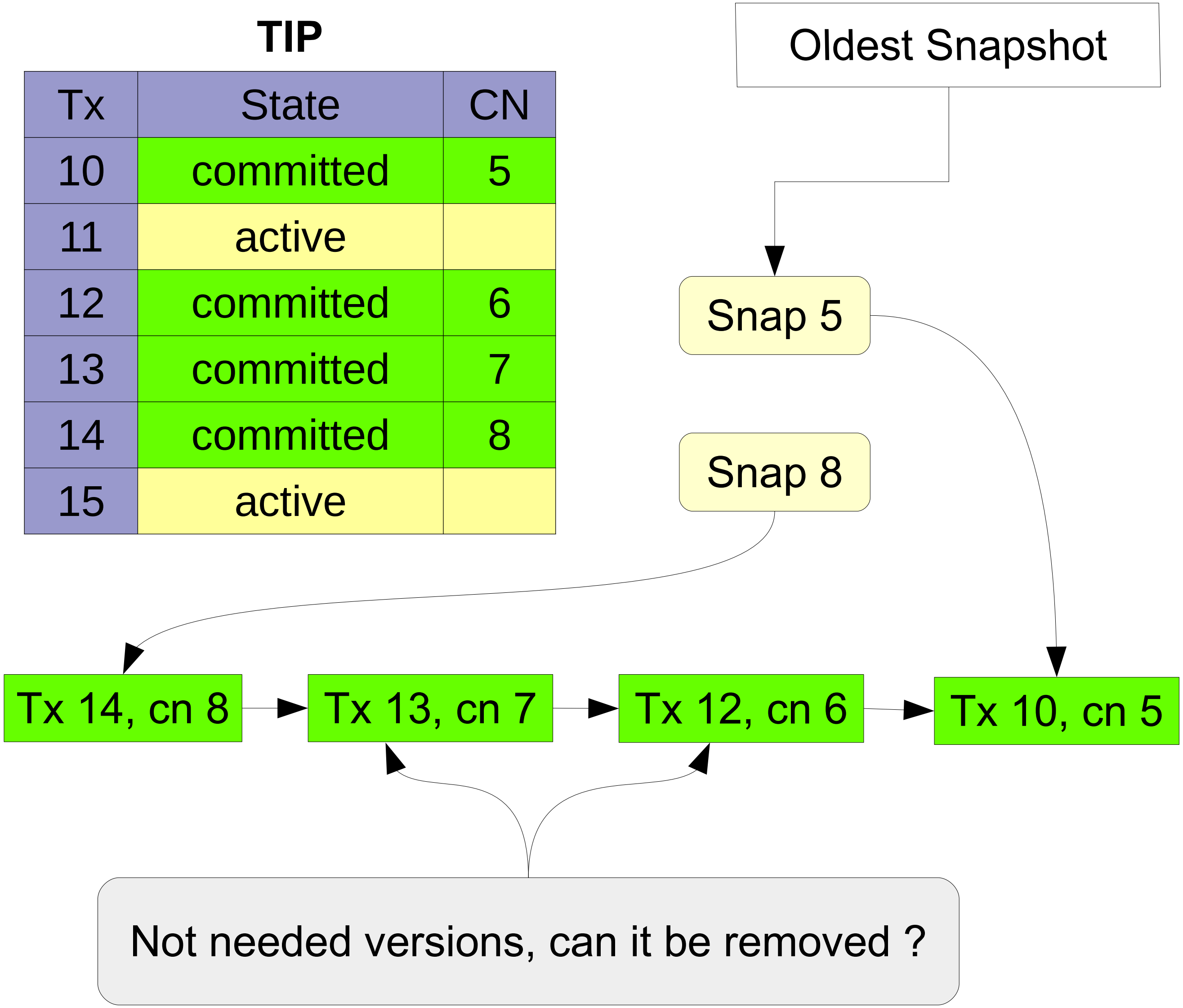
Long running transactions

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	



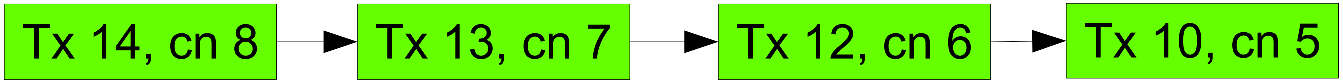
Long running transactions

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
...



Tx 14, cn 8 → Tx 13, cn 7 → Tx 12, cn 6 → Tx 10, cn 5

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

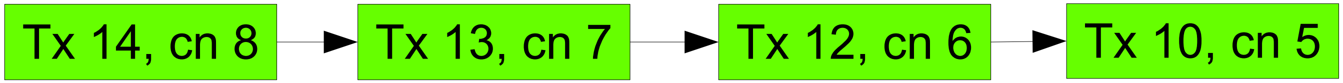
Long running transactions

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
...


Tx 14, cn 8 → Tx 13, cn 7 → Tx 12, cn 6 → Tx 10, cn 5

- 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 !

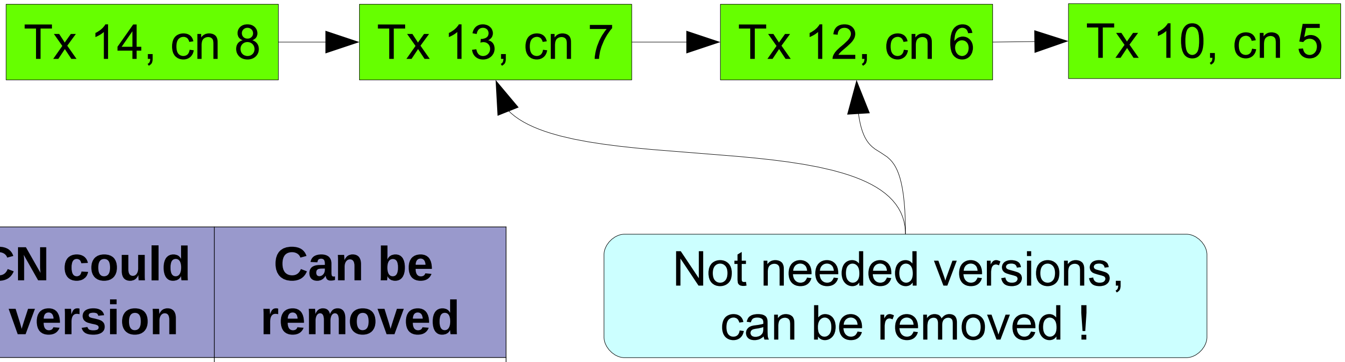
Long running transactions

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
...



Not needed versions, can be removed !

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

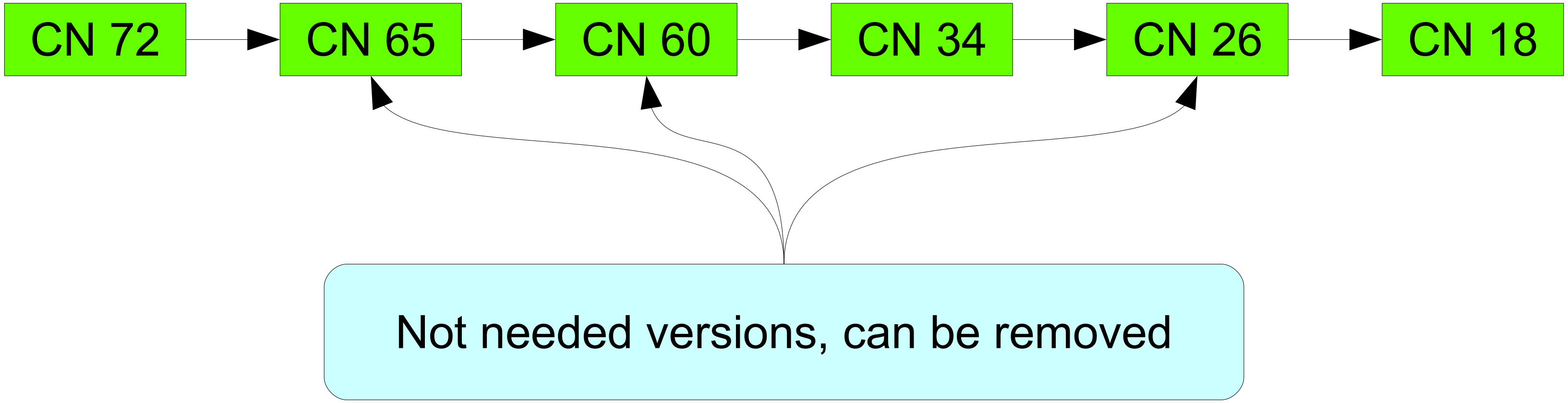
Intermediate record versions

Active snapshots

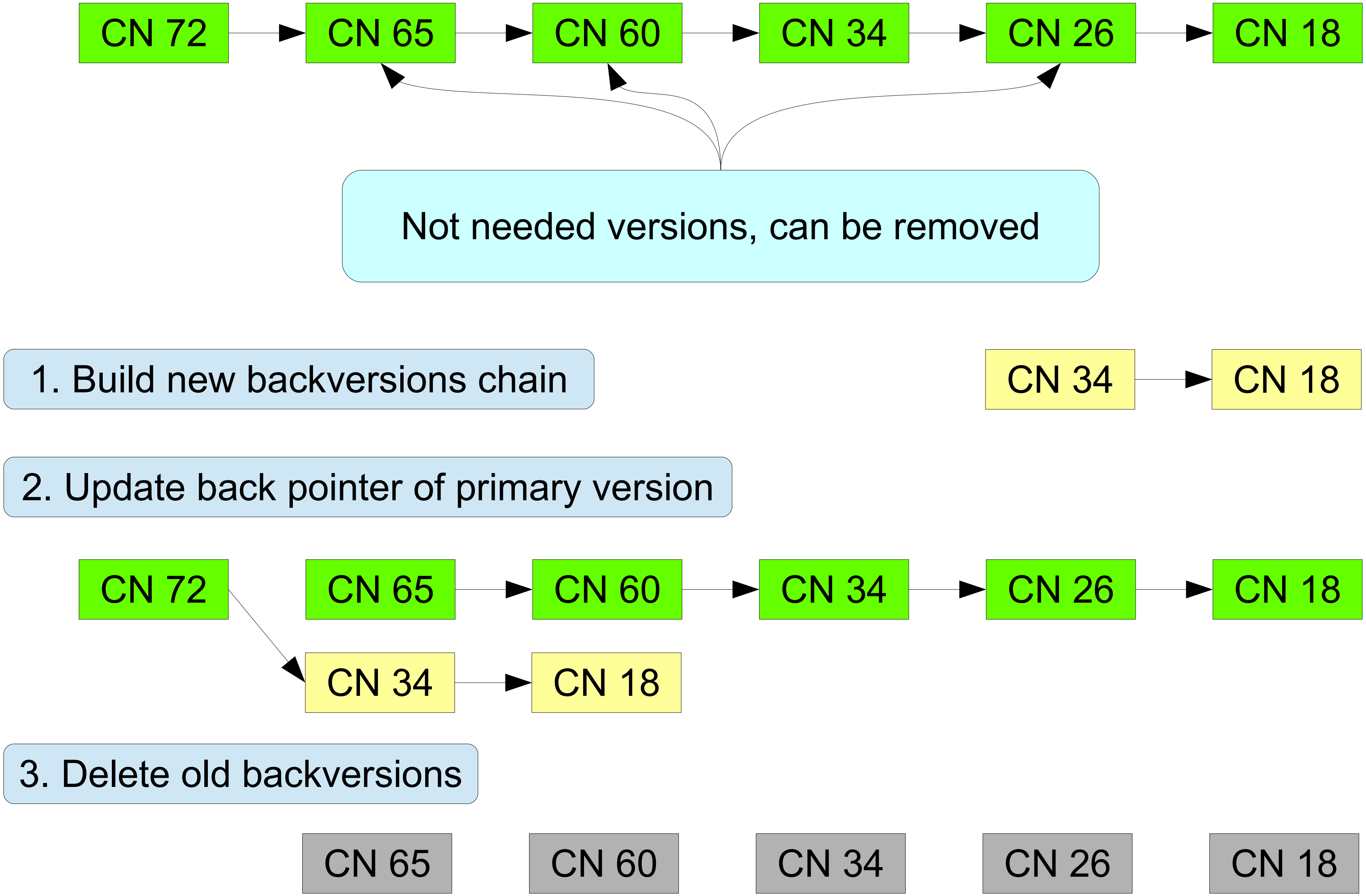
CN of snapshot
23
48
54
57
78
...

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



Intermediate record versions



CN 72 → CN 65 → CN 60 → CN 34 → CN 26 → CN 18

Not needed versions, can be removed

1. Build new backversions chain

CN 34 → CN 18

2. Update back pointer of primary version

CN 72 → CN 34 → CN 18

CN 65 → CN 60 → CN 34 → CN 26 → CN 18

3. Delete old backversions

CN 65 CN 60 CN 34 CN 26 CN 18

Intermediate record versions

- Intermediate GC cost is not zero
  - Avoid concurrent Intermediate GC of the same record
- When it happens
  - After UPDATE, DELETE, SELECT WITH LOCK
    - record is “owned” by current active transactions
    - no concurrency with other user attachments
    - GCPolicy = Cooperative or Combined
  - Sweep, background GC thread
    - trying to avoid concurrency with user attachments
      - only if primary record version is committed
      - only if traditional GC is not possible (tx > OST)

Transactions

- *Concurrency* and *Consistency* isolation modes now uses private database snapshot, based on new “Commit Order” feature
  - No more private copies of TIP
  - Private snapshot
    - created - when transaction started
    - released – when transaction finished
    - Current value could be queried using new context variable *“SNAPSHOT_NUMBER”* in *“SYSTEM”* context

Transactions

- New sub-level for *Read Committed* transactions:
  Read Committed Read Consistency
  - Allows to solve problem with non-consistent reads at the statement level
  - Uses private database snapshot while statement executed
  - Similar to concurrency transactions but for the single statement

Transactions

- Read Committed Read Consistency
  - Create private database snapshot when statement started execution (cursor opened)
  - Release snapshot when statement execution finished (cursor fetched to eof or closed)
  - Same snapshot is used for all called sub-statements, including triggers, stored procedures, dynamic statements (in the same transaction context)
  - Autonomous transaction uses own private snapshot

Update conflicts

- How Read Consistency interacts with active concurrent writers
  - Reader <-> Writer
  - Writer <-> Writer

Update conflicts

- When Read Consistency transaction read record, updated by concurrent active transaction
  - No sense to wait for commit\rollback of concurrent transaction – our snapshot not allows us to detect it
  - Read backversion, if it is exists
  - Similar to Record Version transactions

Update conflicts

- When Read Consistency transaction going to update record, updated by concurrent active transaction
  - Update conflict !

Update conflict

- Traditional handling of update conflicts by applications
  - Try to update record
  - If conflict happens
    - Rollback work
    - Start new transaction
    - Repeat from start

Update conflict

- Restart request algorithm
  - Try to update record
  - If conflict happens
    - Wait for commit\rollback of concurrent transaction
      - On wait timeout return update conflict error
    - If concurrent is rolled back
      - Remove dead record version and try to update same record again
    - If concurrent is committed
      - Undo all actions of current statement
      - Release statement snapshot
      - Create new statement snapshot
      - Repeat from start

Update conflict

- Restart request algorithm
  - More efficient than application-level restart
    - No need to restart transaction
    - Save network round-trips
  - Number of restarts is limited by hard coded value (10)
  - Could have some side effects
    - Triggers are fired multiply times
  - Not applied if statement already returns records to the client application before update conflict happens

Update conflict

- Restart request algorithm
  - Does not work when there is big contention on the same record !

Update conflict

- Better handling of update conflicts by applications
  - Try to SELECT WITH LOCK
  - If conflict happens
    - Rollback work
    - Start new transaction
    - Repeat from start
  - Update record

Update conflict

- New restart request algorithm
  - Try to update record
  - If conflict happens
    - … *same actions* ...
    - If concurrent is committed
      - Undo all actions of current statement, but
        - Leave write locks on all changed records, including conflicted one
        - Same as SELECT WITH LOCK
      - Release statement snapshot
      - Create new statement snapshot
      - Repeat from start

Update conflict

- New restart request algorithm
  - Code exists as pull request and is not merged into master branch yet
  - Code is currently evaluated and tested by team
  - So far results is good

Transactions

- Read Committed Read Only
  - *Read Consistency* transactions still committed at start, but keeps own lock with own transaction number at its data – same as any *Read Committed Write* transaction
    - Necessary to keep statement-level snapshot stability
    - Not delays garbage collection thanks to Intermediate GC
  - *Record Version* and *No Record Version* transactions
    - No changes, works as before

Read Committed Read Consistency

- Support at SQL level
  - *SET TRANSACTION READ COMMITTED READ CONSISTENCY*

Read Committed Read Consistency

- Support at SQL level
  - New value (4) at MON$TRANSACTIONS.MON$ISOLATION_MODE
    - Description available in RDB$TYPES, as usual

```
SELECT RDB$TYPE, RDB$TYPE_NAME FROM RDB$TYPES
 WHERE RDB$FIELD_NAME = “MON$ISOLATION_MODE”;

RDB$TYPE RDB$TYPE_NAME
======== ==================================
       0 CONSISTENCY
       1 CONCURRENCY
       2 READ_COMMITTED_VERSION
       3 READ_COMMITTED_NO_VERSION
       4 READ_COMMITTED_READ_CONSISTENCY
```

Read Committed Read Consistency

- Support at API level
  - New TPB tag
    - `isc_tpb_read_consistency`
  - Sample TPB
    - `isc_tpb_read_committed, isc_tpb_read_consistency, isc_tpb_write`

Read Committed Read Consistency

- New per-database configuration setting
  - ReadConsistency
- ReadConsistency = 1 (default)
  - Force engine to make any read committed transaction mode to be read committed read consistency
  - For brave developers who want to avoid inconsistencies once and forever ;)
- ReadConsistency = 0
  - Allows to use all three kind of read committed mode with no limitations

Shared snapshots

- It is easy now to implement snapshots sharing
  - Allows for many independent transactions to see the same stable data set
    - Concurrency transactions, of course
  - Useful to handle some big task by parallel connections

Shared snapshots

- Snapshots sharing
  - Start some concurrency transaction
  - Query its snapshot number
    - RDB$GET_CONTEXT(‘*SYSTEM*’, ‘*SNAPSHOT_NUMBER*’), or
    - isc_transaction_info(… fb_info_tra_snapshot_number …)
  - Start new concurrency transaction(s) using existing snapshot number
    - SET TRANSACTION SNAPSHOT *AT NUMBER <number>*, or
    - new TPB tag

      isc_tpb_at_snapshot_number, <length>, <number>

Summary

- Statement-level read consistency problem is solved
- Long running transactions not blocks garbage collection
- Attempt to handle update conflicts efficiently and automatically
- Very easy way to share same snapshot by many independent transactions

THANK YOU FOR ATTENTION

Questions ?

Firebird official web site

Firebird tracker

hvlad@users.sf.net