Scalable MVCC solution for Many core machines

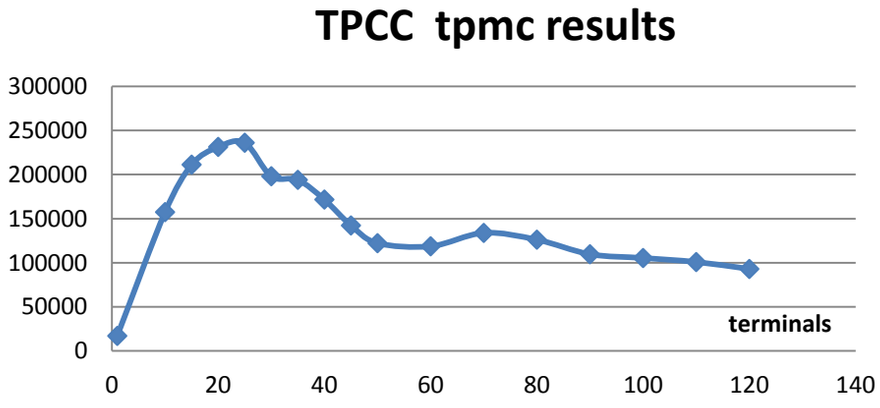
Dilip Kumar (dilip.kumar@huawei.com)
Tao Ye (yetao1@huawei.com)
Nirmala S (nirmalas@huawei.com)
Xiaojin Zheng (zhengxiaojin@huawei.com)

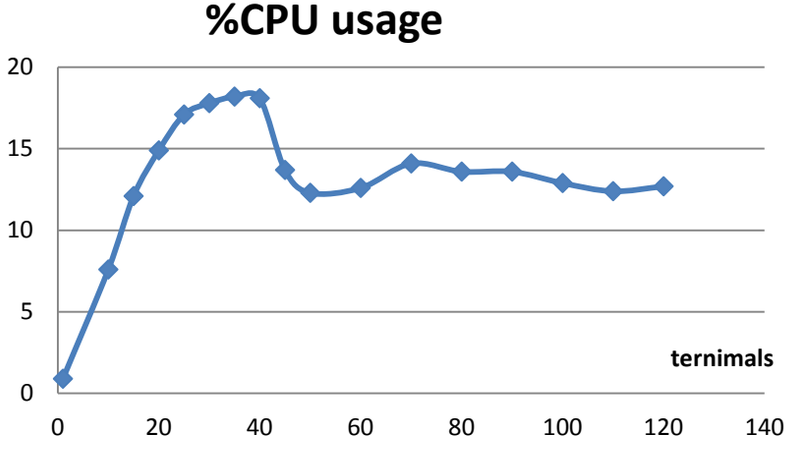
2015/06/04

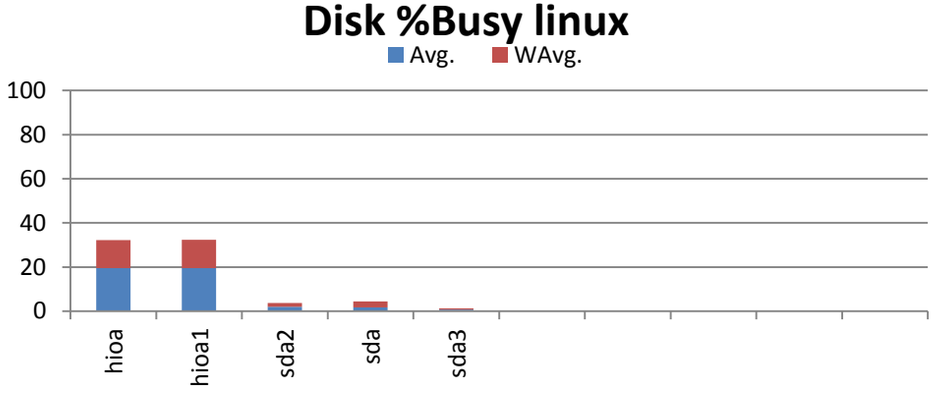
Content

- Multicore Scalability issue
- Concept of MVCC
- MVCC in PG
- Lock Free CSN Solution
- Performance Results
- Conclusion

Multicore Scalability issue – TPCC test







Test Environment:

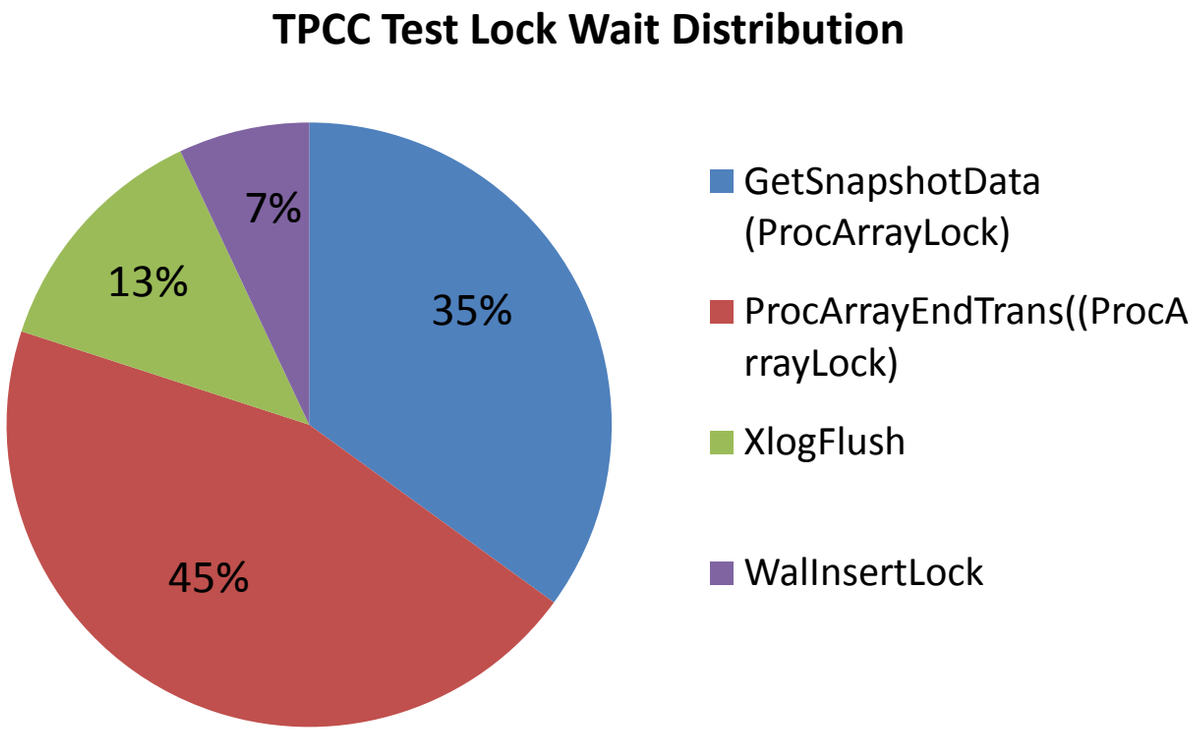
Workload: "TPC-C" the industry standard benchmark for OLTP.

Hardware: System with 60 physical cores (120 threads) + SSD +

Results:

Peak TPMC is hit at 25 terminals. After this increase in terminals reduces the performance. I/O and CPU are under utilized and maximum number of process are in idle state.

Multicore Scalability issue - Bottleneck Analysis



On analyzing stack for multiple concurrent terminals, it is evident that

- **ProcArrayLock contention is increasing with number of terminals.**
- **Contention starts increasing steeply after 30 terminals and stays in constant state.**
- **This lock accounts for 80% of contention in the whole system (GetSnaphsotData and ProcArrayEndTransaction)**

Concept of MVCC

MVCC is a concurrency control mechanism commonly used by database implementations to control fast, safe and concurrent access to data.

MVCC is designed to provide following features for concurrent access

- **Readers do not block writers**
- **Writers do not block readers**

When an object has to be overwritten, it is marked obsolete and a new copy of it is written by the writer. Reader reads the current version and writer affects a future version of data. This ensures that readers do not block writers. Since writer is working on a different copy of data it does not block other readers.

MVCC is generally implemented using transactionID or some timestamp mechanism. This is stored in each tuple so at run time, a decision can be made on the visibility of the tuple for any user given SQL.

MVCC in PG

MVCC implemented in PG is using transactionID. Each tuple maintains a xmin(XID of transaction that created it) and a xmax (XID of transaction that deleted it). A snapshot is taken at the beginning of a SQL statement. The main purpose of taking a snapshot is to get

- **highest-numbered committed transaction**
- **Lowest-numbered running transaction**
- **Transactions that are currently Running**

- **Visible tuples must have a creation transaction id that:**

  - is a committed transaction

  - is less than the xmin transaction counter stored at query start or

  - was not in-process at query start

- **Visible tuples must also have an expire transaction id that:**

  - is blank or aborted or

  - is greater than the xmax transaction counter stored at query start or

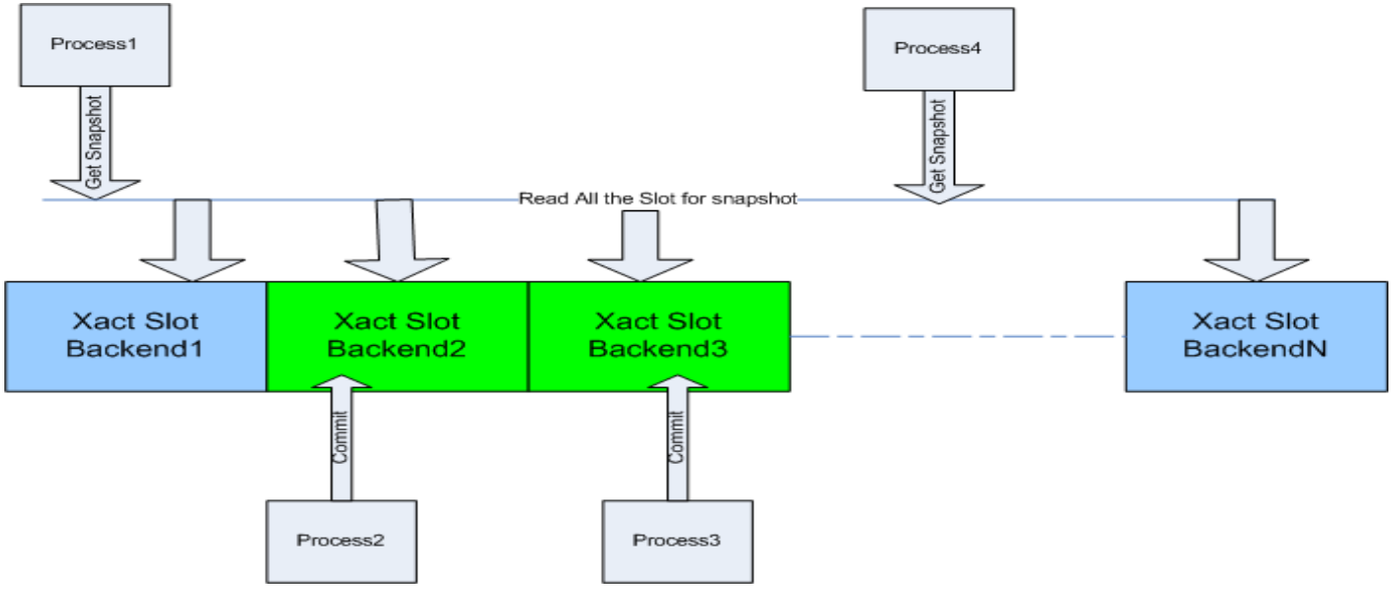
  - was in-process at query start

MVCC in PG

During snapshot creation, for getting all the running transaction information, a process takes Shared latch on ProcArrayLock (GetSnapshotData)

When a transaction is committing, it takes an exclusive latch on ProcArrayLock (ProcArrayEndTransaction) to update shared global variables.

When transactions – starting and ending try to use ProcArrayLock, contention ensues and scalability reduces.



Initial CSN Snapshot Proposal

- **Ant Aasma have proposed a initial solution of the CSN based snapshot.**
- **As per the solution, Snapshot taking cost can be reduced by converting snapshot to as Commit Sequence Number instead of reading transaction id list.**
- **We have used this solution as our initial design idea.**

Lock Free CSN solution

- **Main purpose of fetching all running transaction information is to know, which transaction committed before taking Snapshot and which after taking snapshot.**

- **Instead of getting the list of running transaction while getting the snapshot we can depend upon some timeline variable.**

- **Each transaction when committing, can increment a global number – Commit Sequence Number (CSN). Any transaction which is starting, will note down the current CSN number of the system during its snapshot creation – this can be termed as Snapshot CSN.**

- **Now to check the visibility of any tuple, it is enough to see if CSN associated with its xmin and xmax is less than Snapshot CSN.**

Lock Free CSN solution

By effectively having an integer(CSN) which is shared across all backend, we can remove traversing Pgxact structure and thereby removing the need to acquire ProcArrayLock in shared mode.

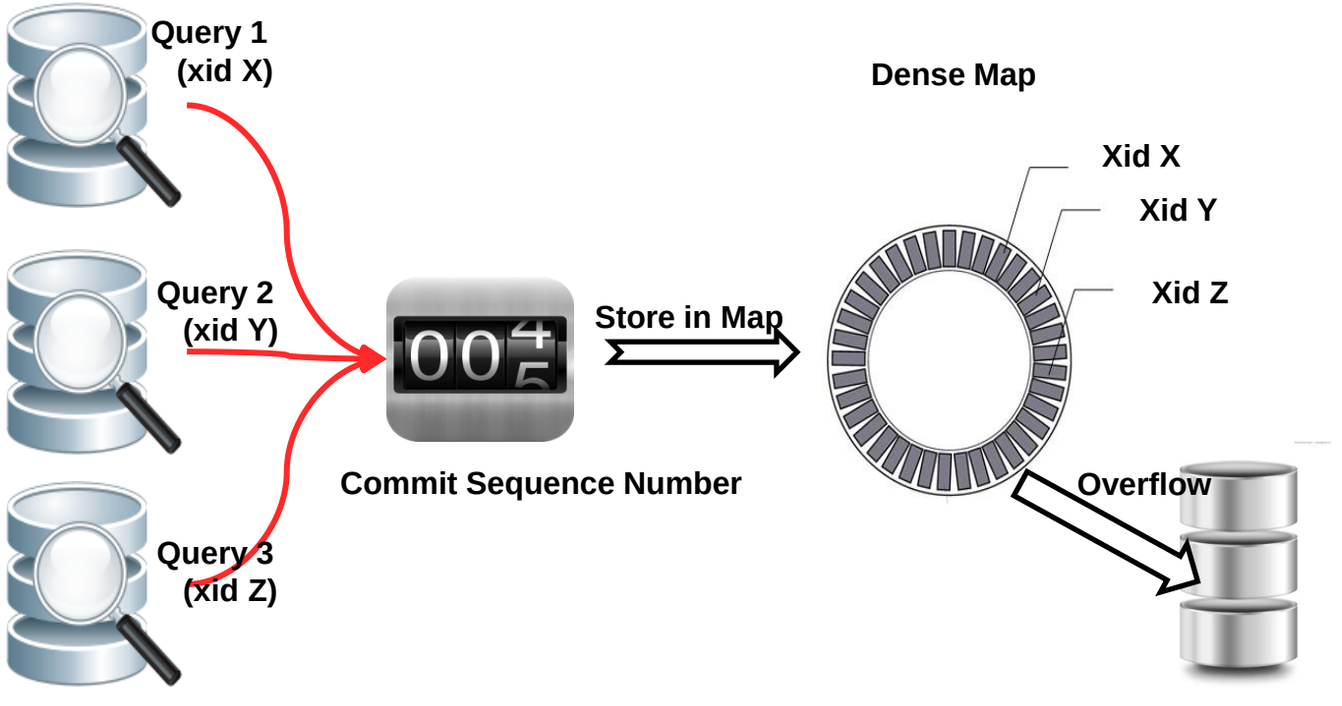
One of the obvious problems for the CSN based solution is how to maintain a map between CSN and XID. This data structure should have the following properties

- **Searching should be efficient**
- **Read/Write should be lock-free**

We decided to select a circular array, in which we can directly get the XID slot using XID%ARRAY_SIZE. This ensures that there is no extra searching cost.

Lock Free CSN solution

- **XID to CSN mapping is stored in circular array called Dense Map. So that most of the operation can be atomic without any lock.**
- **Dense Map size selected >4MB (with experiment on TPCC, beyond this size performance is peak).**
- **Old Transactions are moved to secondary MAP, called sparse MAP**



Lock Free operations

Now major challenge is to keep the access to dense map lock free, b/w the reader (getting the CSN value for a transaction) and writer(reusing the slot and changing the transaction ID).

Start Transaction (Writer)

- **If Slot is empty start transaction can simply use with transaction ID. (multiple XID entering together is protected using XIDGenLock)**
- **If Slot is not Empty but XID is very OLD (not concurrent to any snapshot), just flush it out**
- **If XID is concurrent then move it to secondary MAP.**

Commit Transaction (Writer)

- **If XID found in dense map assign the CSN.**
- **If XID not found in dense map, search the slot in secondary map and assign the CSN**

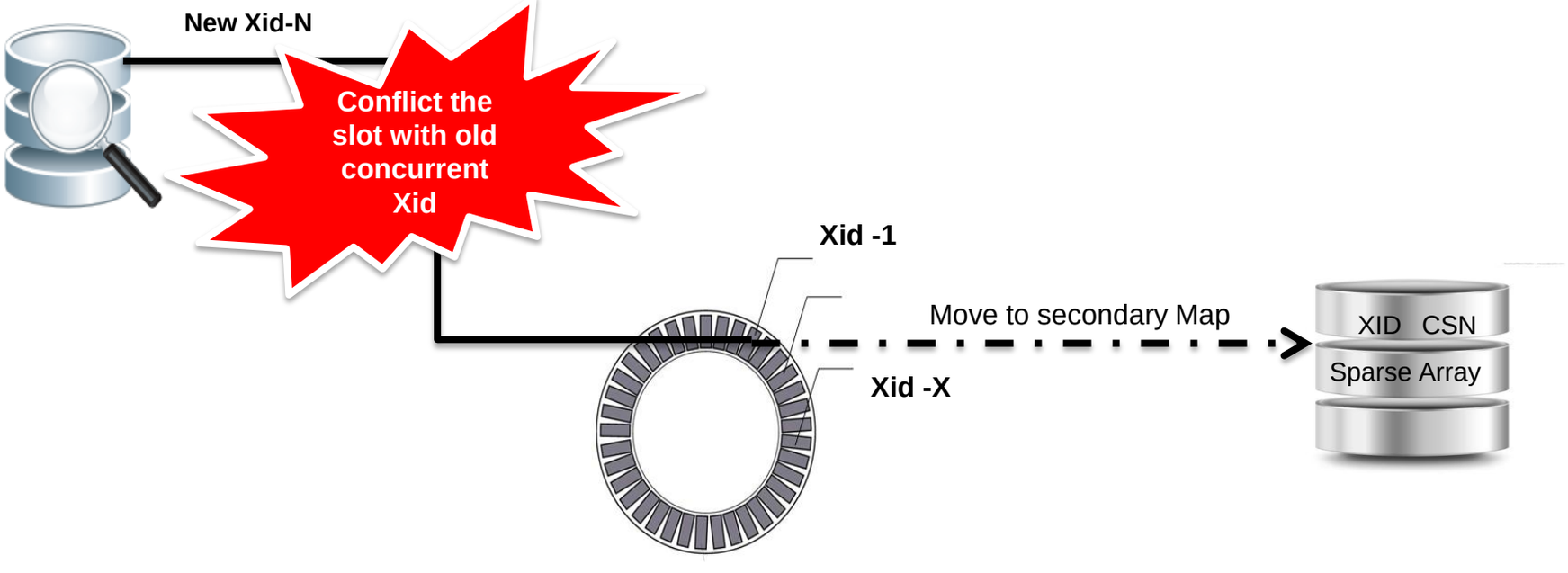
Lock Free operations cont..

Visibility Check of the tuple (Reader)

- **Read Slot XID (Match XID with dense map Slot XID)**
- **Read Slot CSN**
- **Read Slot XID again (Rematch the XID, if matches than CSN value can be used. Otherwise some write has overwritten the XID, then start by reading CSN again)**

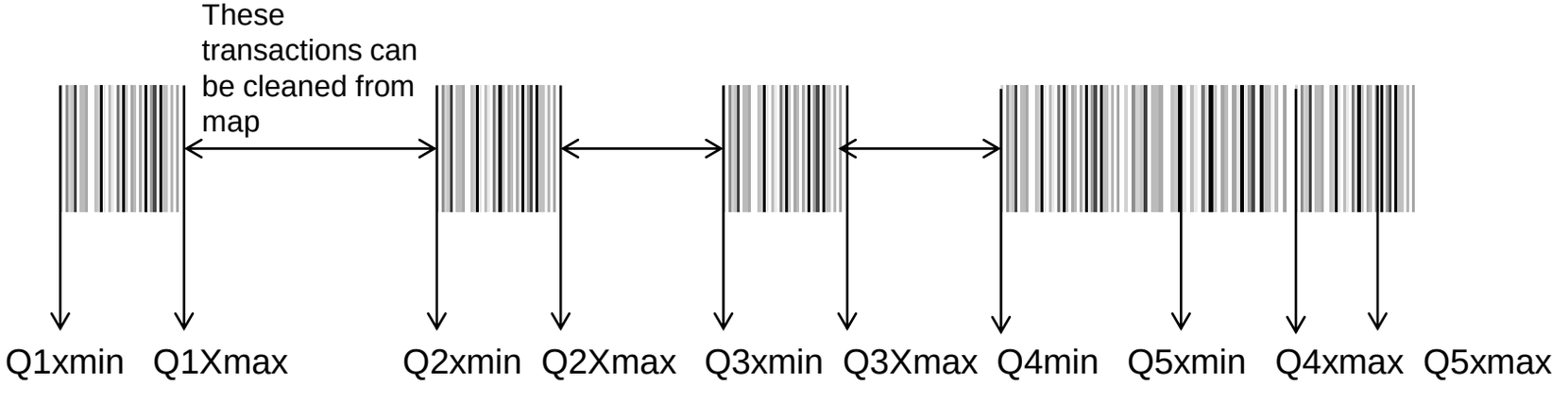
Conflict handling

- **While starting the new transaction if there is concurrent transaction in Slot,.**
- **We move such transaction to a secondary map, called sparse map. Sparse map is implemented using Array of XID-CSN pair (sorted in XID order)**
- **Writers takes exclusive locks on sparse map, and reader takes shared locks.**
- **If a transaction is active, its XID should fall in either of dense map or sparse map. If it is not found in both, then the transaction is not concurrent to any snapshot**



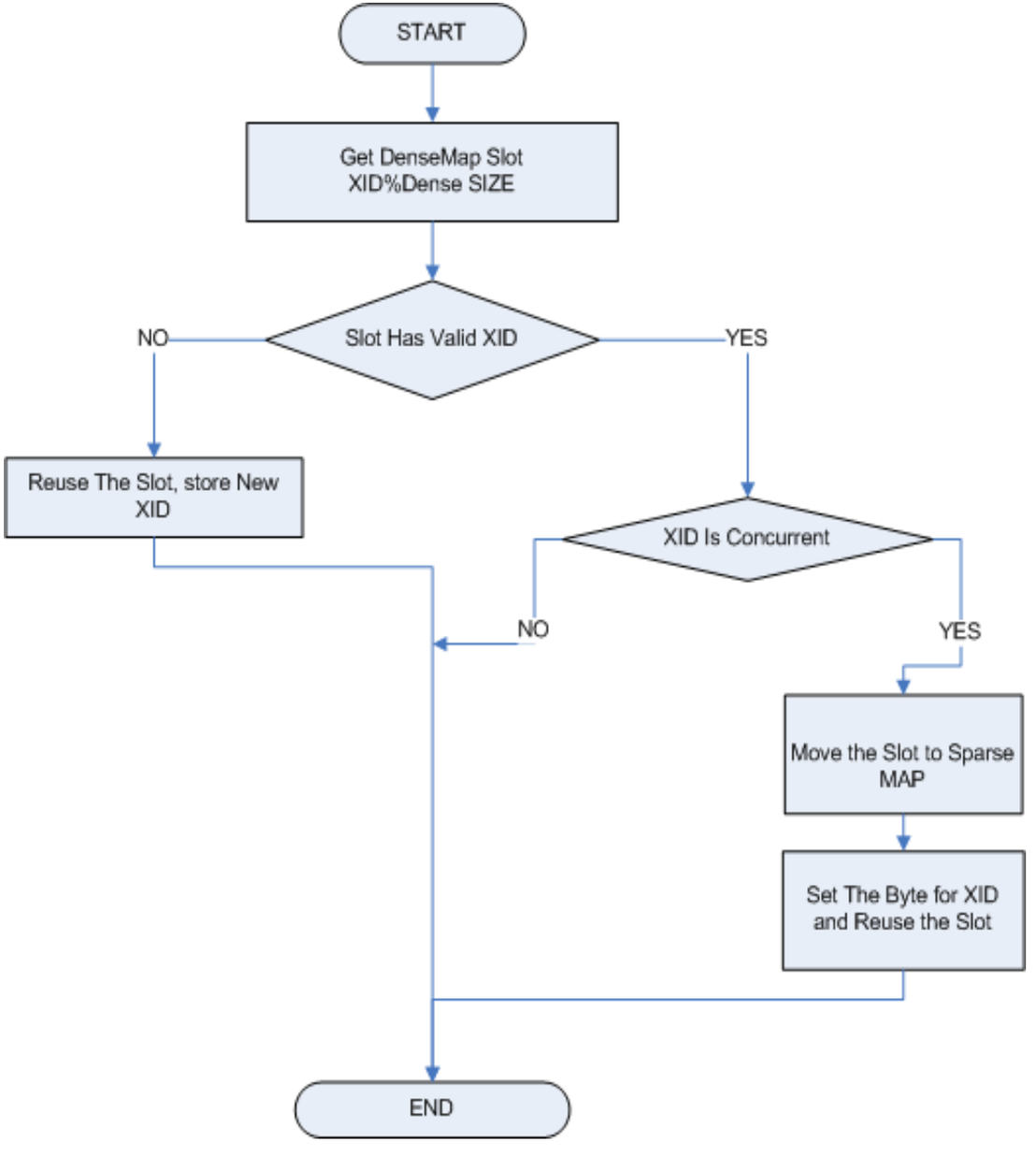
Map Cleanup

- **Sometime because of few long running query, many transaction stays in the map. And these should be clean up.**
  - **For cleaning up the long running transactions, long running transactions are identified and there snapshot is marked to be converted to XID snapshot.**
  - **Conversion from CSN to XID snapshot happen when long running query try to do the visibility check.**
  - **Since conversion from csn to xid is asynchronous, we have taken extra cleanup action to remove unwanted slots, As per this clean all the XID which is not overlapping b/w two transaction snapshots.**



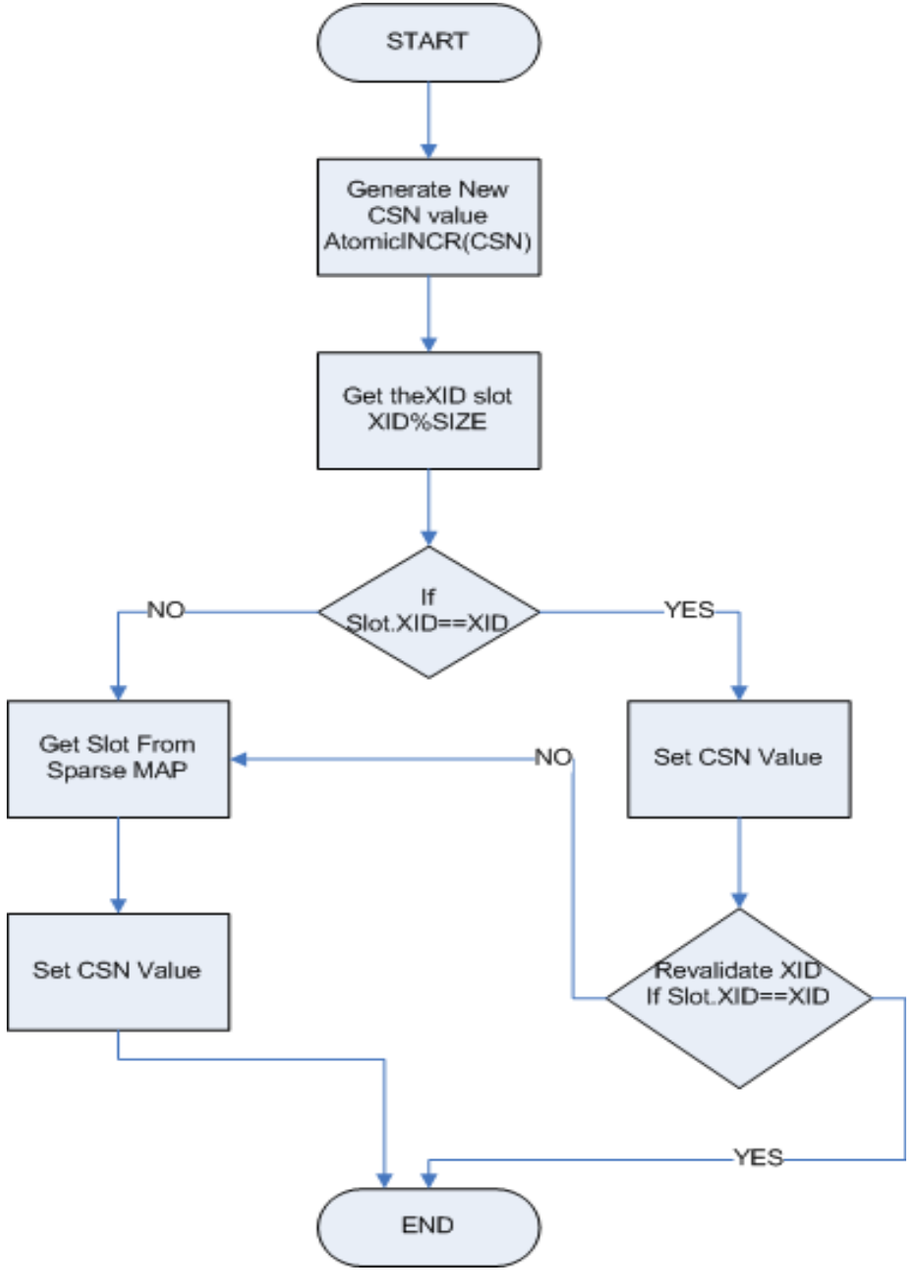
Start transaction

- **Get a slot for this Transaction from the dense MAP using XID % SIZE**
- **If there is Old XID in Slot and its concurrent (either running or CSN is not smallest than global snapshot CSN min)**
- **Move to Sparse Map**
  - ✓ **Take Sparse Map Lock**
  - ✓ **Copy Slot to Sparse Map**
  - ✓ **Release Lock**
  - ✓ **Set Bit for XID in XID Map**
- **Reuse the Slot.**
- **Assign the XID in slot and initialize CSN to INVALID_CSN.**



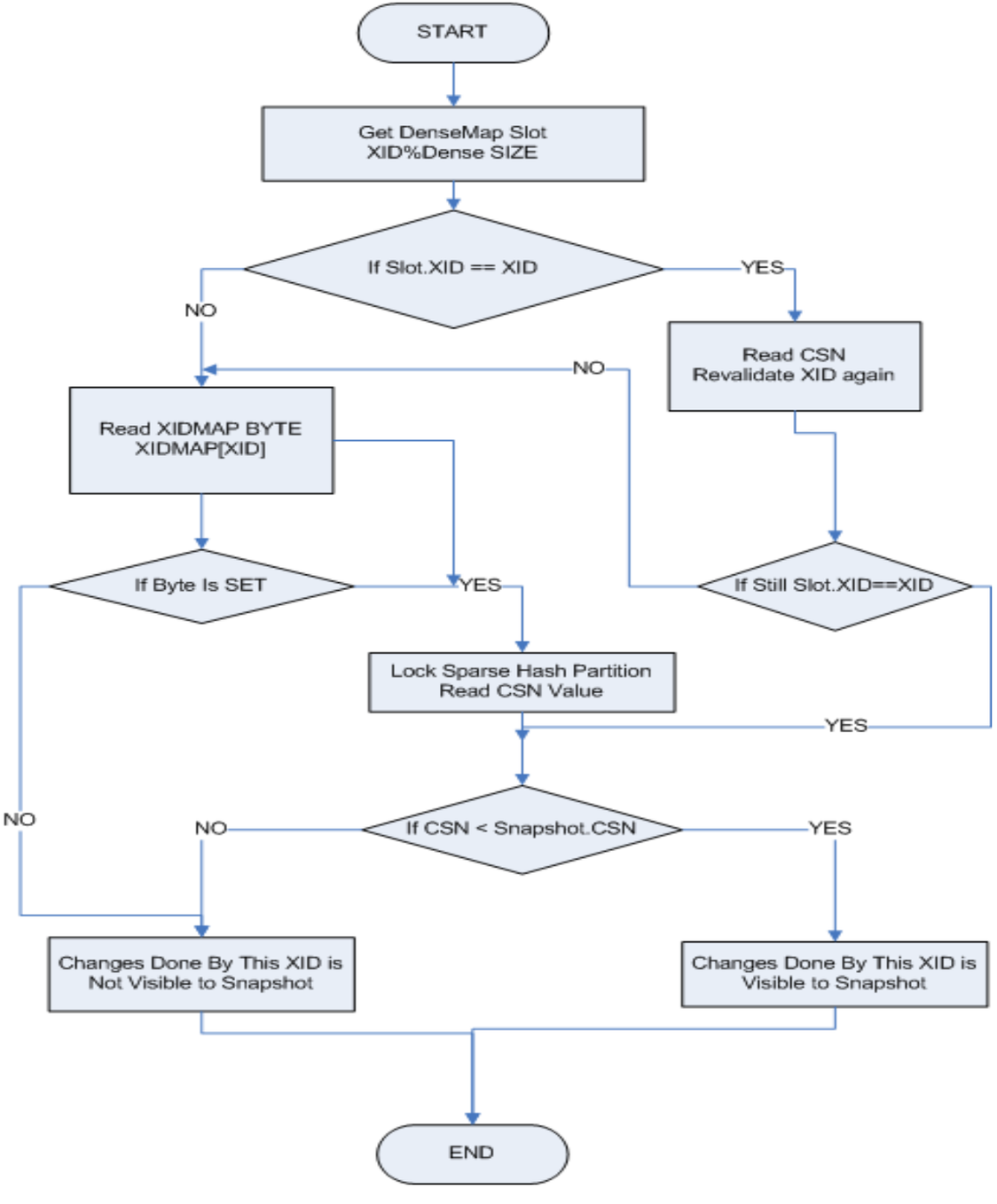
End Transaction

- **Get a slot for this Transaction from the dense MAP using XID % SIZE**
- **Generate New CSN(Increment Global CSN value)**
- **Assign SUB_CSN value to all sub transactions.**
- **Assign CSN value to main transaction.**
- **Assign CSN value to all sub transactions.**
- **In case of abort Assign special CSN called ABORT_CSN.**



Visibility Check

- For checking a visibility of a operation done by transaction (XID), first find the slot for the XID, using XID%DenseMapSize.
- If XID matches, read CSN value and check the XID again.
- If XID not match then check the ByteMAP, if set than search in Sparse Map.
- Compare the Slot CSN with Snapshot CSN
- If Slot CSN > Snapshot CSN changes are not visible
- Else Changes are visible.



GlobalXmin and Vacuum

Though GetSnapshotData is simplified and now its just reading CSN, But many operations like vacuum still depends upon GlobalXmin. So we still need to support calculating the GlobalXmin.

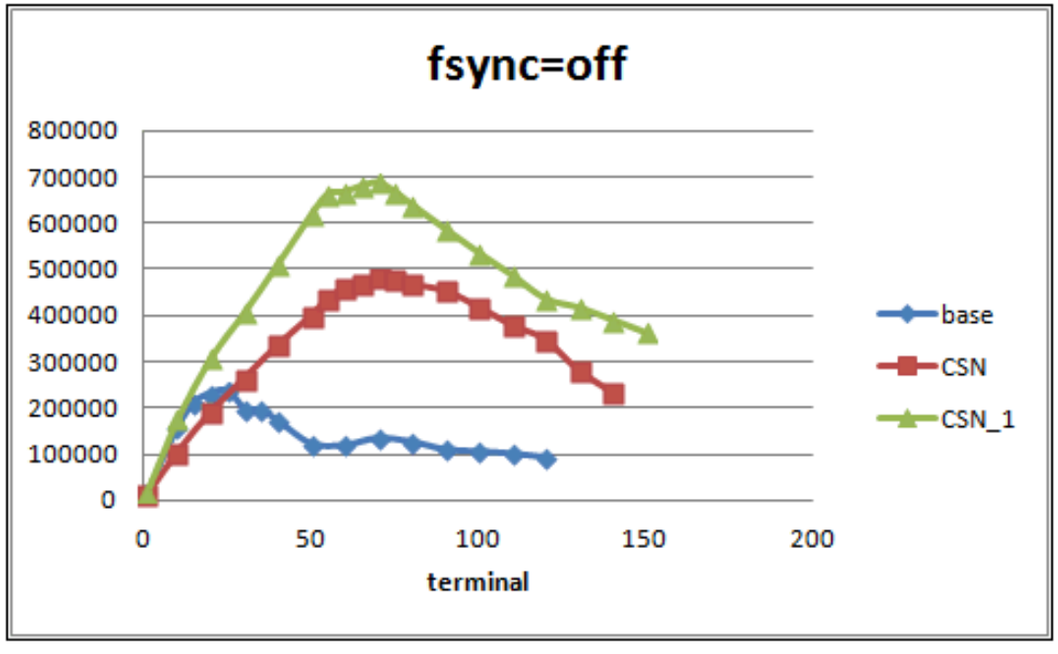
GlobalXmin Solution

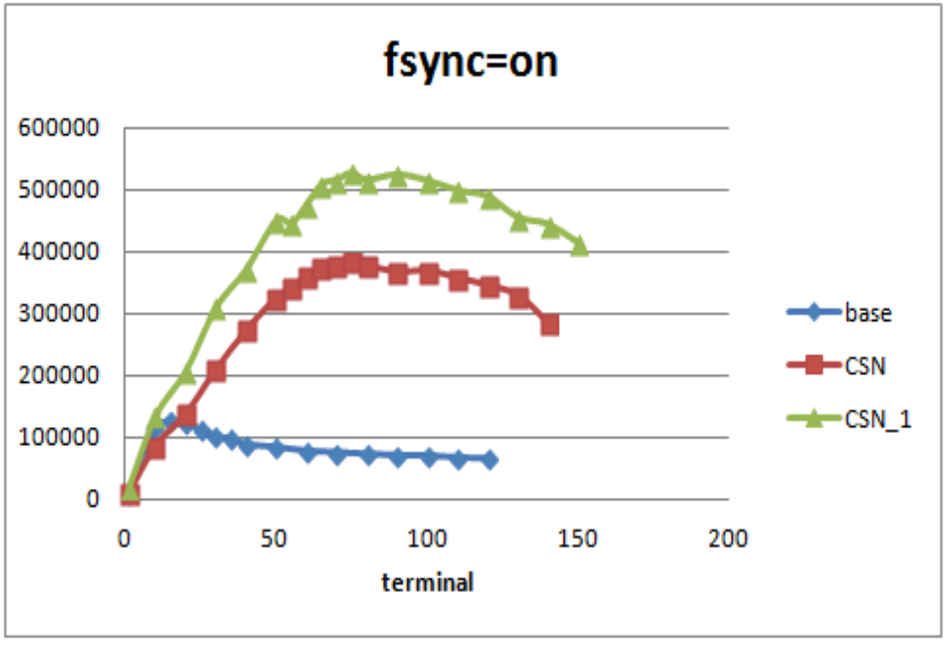
- **Instead of calculating the Accurate Xmin it calculated delayed while Committing the transaction.**
- **While committing the transaction Xmin is stored in pgxact in separate variable.**
- **While taking the snapshot same variable is read from the self slot of pgxact and stored as xmin.**
- **Now Vacuum can work as it is without any change.**

GlobalCSNMin Solution

- **At end transaction we need to calculate the GlobalCSNMin for CSN map cleanup.**
- **Instead of maintaining Two variable GlobalXmin and GlobalCSNMin we can implement vacuum logic based on GlobalCSNMin.**

Performance Test





Performance CSN VS Base Pg9.4

Test Environment:

- **Workload: "TPC-C" the industry standard benchmark for OLTP.**
- **Hardware: System with 60 physical cores (120 threads) + SSD +**

Results

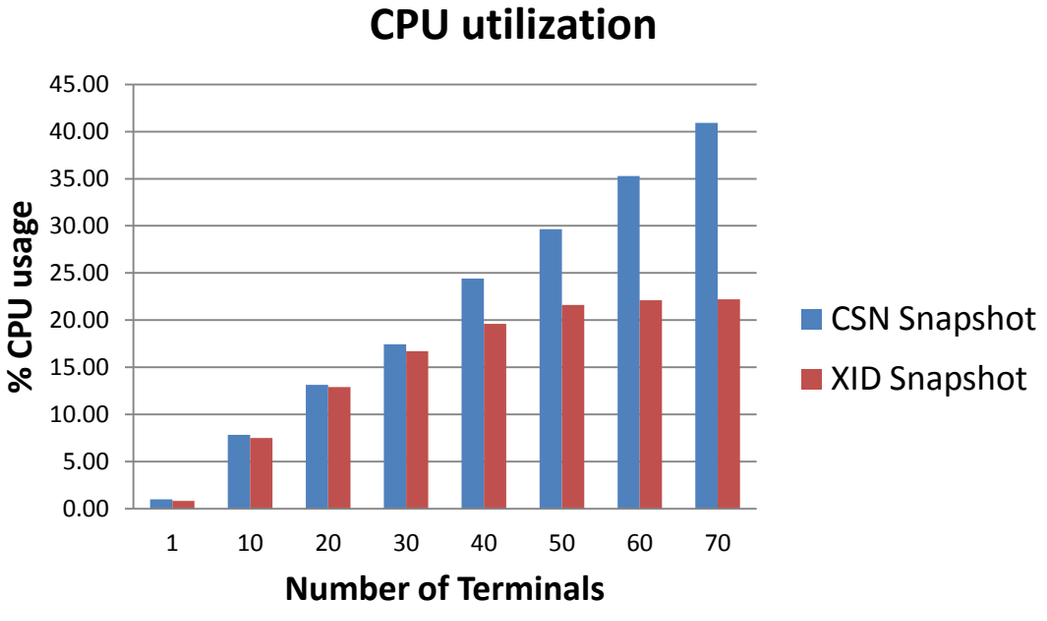
- **fsync=off, peak terminal=70 and peak tpmC=687,325, improve 191%**
- **fsync=on, peak terminal=75 and peak tpmC=525,862, improve 316%**

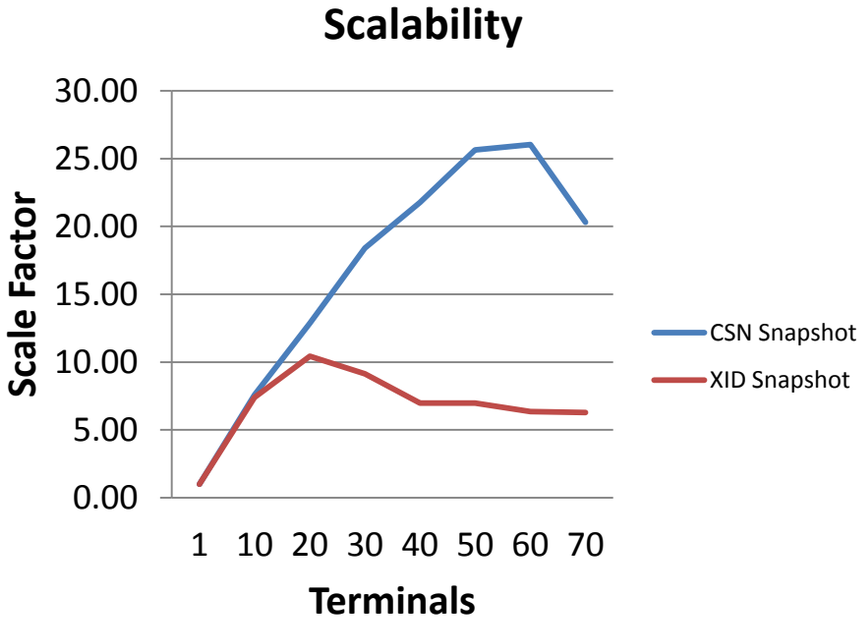
base : Base code performance of PG9.4 with best configuration

CSN : Lock Free CSN code performance

CSN_1 : Lock Free CSN performance by configuration change

Performance Test cont..



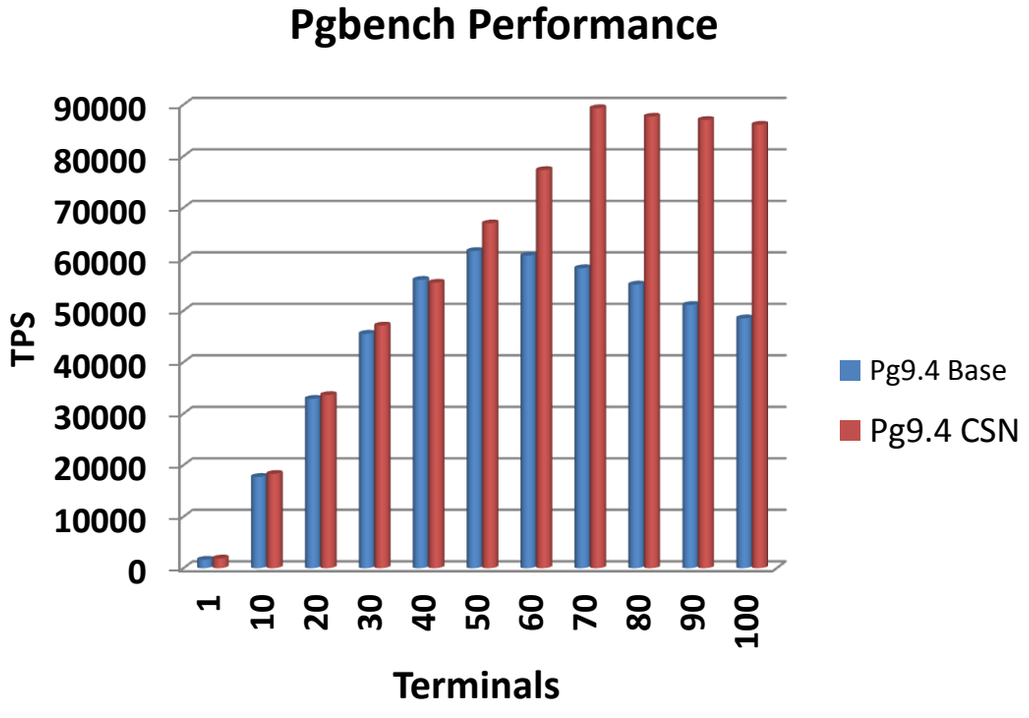


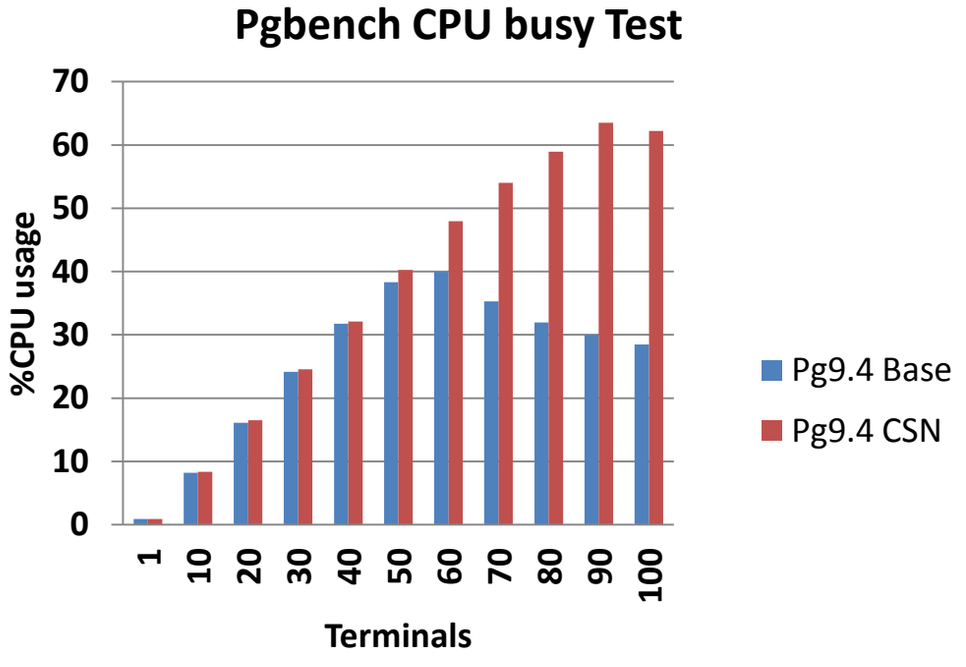
CPU and Scalability (TPCC)

Results:

- **CSN snapshot is able to utilize all the cores of 60 cores machines.**
- **CSN snapshot could scale beyond 60 cores wherein XID snapshot could not scale beyond 25 cores.**

Performance Test cont..





Performance CSN VS Base Pg9.4

Test Environment:

- **Workload: "PGBench (sync commit = off)".**
- **Hardware: System with 60 physical cores (120 threads) + SSD +**

Results

- **fsync=off, peak terminal=70 and peak TPC=89,321, improve 45%**

Conclusion

- **From last two experiment we can observe that now ProcArrayLock Bottleneck is completely removed from the system.**
- **Now new bottlenecks are getting uncovered like XlogFlush Lock, Clog Partition Locks.**
- **This is clearly visible from TPCC experiment, where by changing some configuration , it can scale further, but same was not possible with base code as it was already hitting earlier bottleneck (ProcArrayLock).**

Acknowledgement

Thanks for supporting in design and discussion

Prasanna,

Huijun Liu,

Guogen Zhang,

Qingqing Zhou,

Yuanyuan Nie

Kumar Rajeev Rastogi

PG On BIG Machine

Redo ?
Clog ?
??

What is the Next Bottleneck

Q & A

Thank You