Problems with PostgreSQL on Multi-core Systems with Multi-Terabyte Data

Jignesh Shah and
PostgreSQL Performance Team @ Sun
Sun Microsystems Inc

PGCon May 2008 - Ottawa

PostgreSQL Performance Team @Sun

- Staale Smedseng
- Magne Mahre
- Paul van den Bogaard
- Lars-Erik Bjork
- Robert Lor
- Jignesh Shah
- Guided by Josh Berkus

Agenda

- Current market trends
- Impact on workload with PostgreSQL on multicore systems
  - > PGBench Scalability
  - > TPCE- Like Scalability
  - > IGEN Scalability
- Impact with multi-terabyte data running PostgreSQL
- Tools and Utilities for DBA
- Summary Next Steps

Current Market trends

Current Market Trends in Systems

- Quad-core sockets are current market standards
  - Also 8-core sockets available now and could become a standard in next couple of year
- Most common rack servers now have two sockets
  - 8-core (or more) systems are the norm with trend going to 12-16 core systems soon
- Most Servers have internal drives > 146 GB
  - Denser in capacity, smaller in size but essentially same or lower speed
  - More denser in case of SATA-II drives

Current Market Trends in Software

- Software (including Operating Systems) have yet to fully catch up with multi-core systems
  - “tar” still single process utility
- Horizontal Scaling helps a lot but not a good clean solution for multi-core systems
- Virtualization is the new buzzword for Consolidations
  - Hides the fact that the software is not able to fully capitalize the extra cores :-(
- Research being done on new paradigms
  - Complexity of parallelized software is huge

Current Market Trends in Data

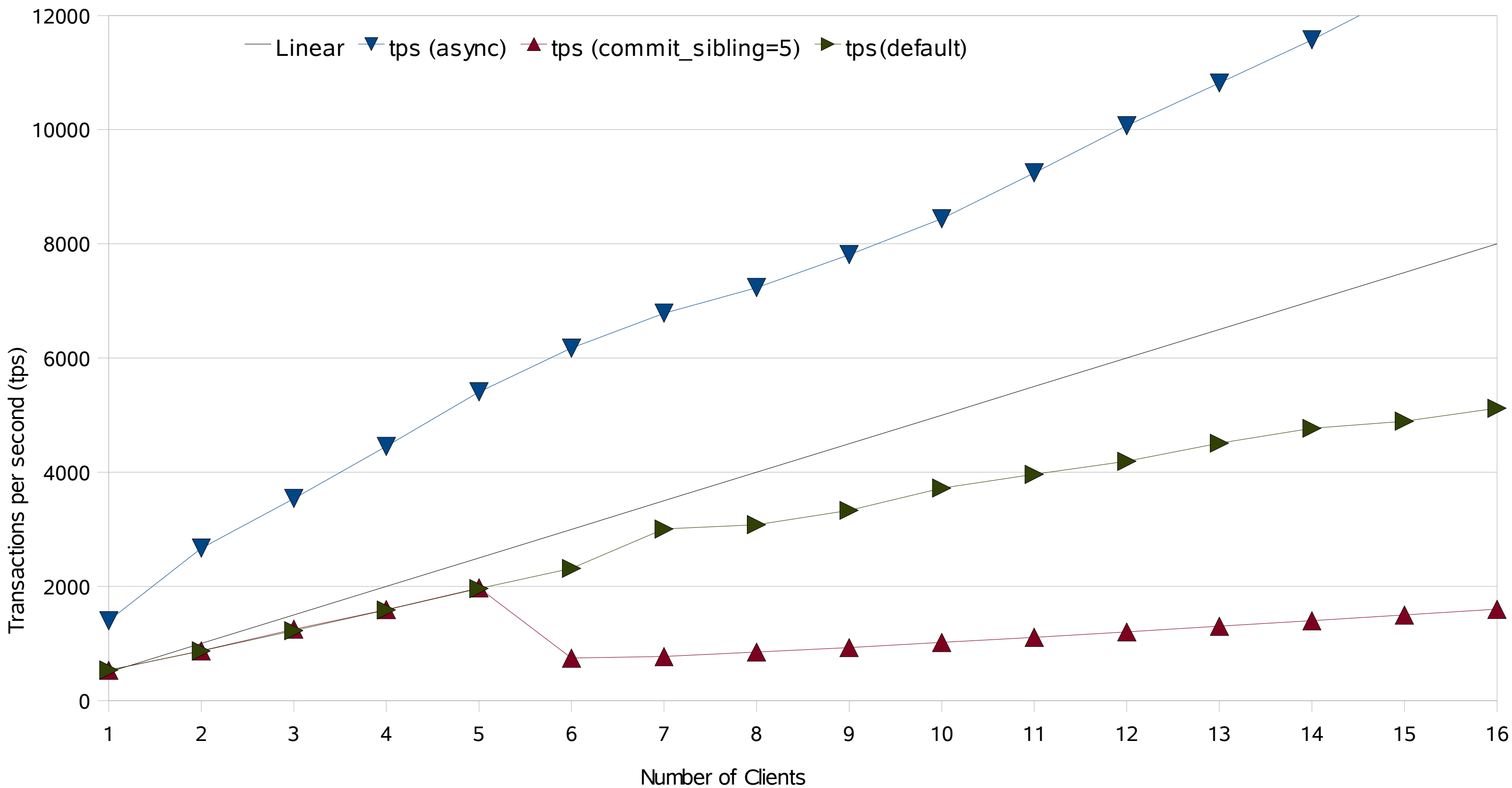
- 12 years ago, a 20GB data warehouse was considered a big database
- Now everybody talks about 200GB-5TB databases
- Some 2005 Survey numbers:
  - Top OLTP DB sizes = 5,973 GB to 23,101 GB
  - Top DW DB Sizes = 17,685 GB to 100,386 GB
  - Source http://www.wintercorp.com/VLDB/2005_TopTen_Survey/TopTenWinners_2005.asp
- Some 2007 Survey numbers:
  - Top DB sizes = 20+ TB to 220 TB (6+ PB on tape)
  - Source http://www.businessintelligencelowdown.com/2007/02/top_10_largest.html

Impact on workloads with multi-cores system running PostgreSQL

PGBench (Modified)

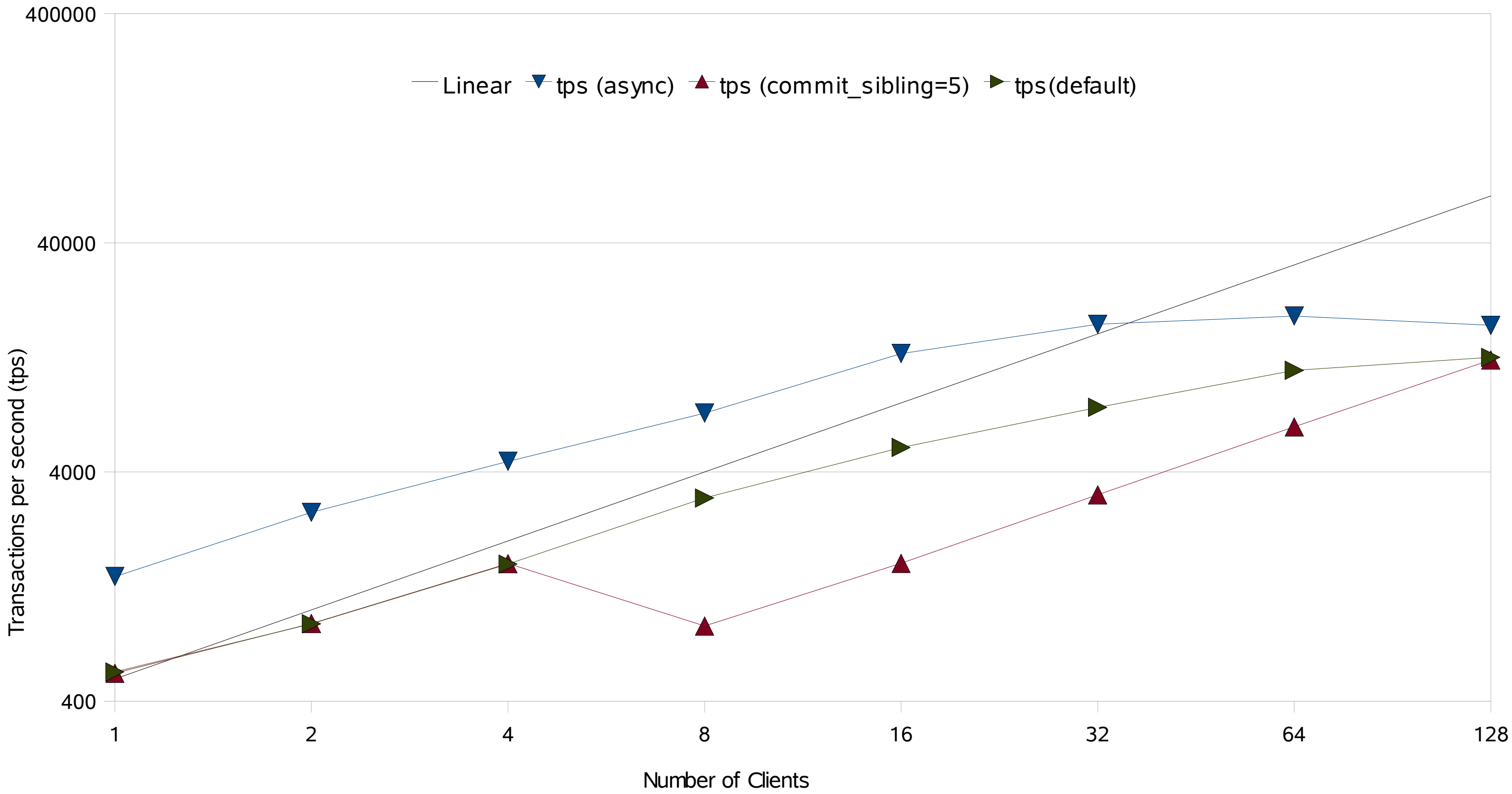
- Custom insert.sql
  - > BEGIN;
  - > INSERT INTO history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
  - > END;
- pgbench -f insert.sql -s 1000 -c 1 -t 10000 pgbench

PGBench (inserts only)



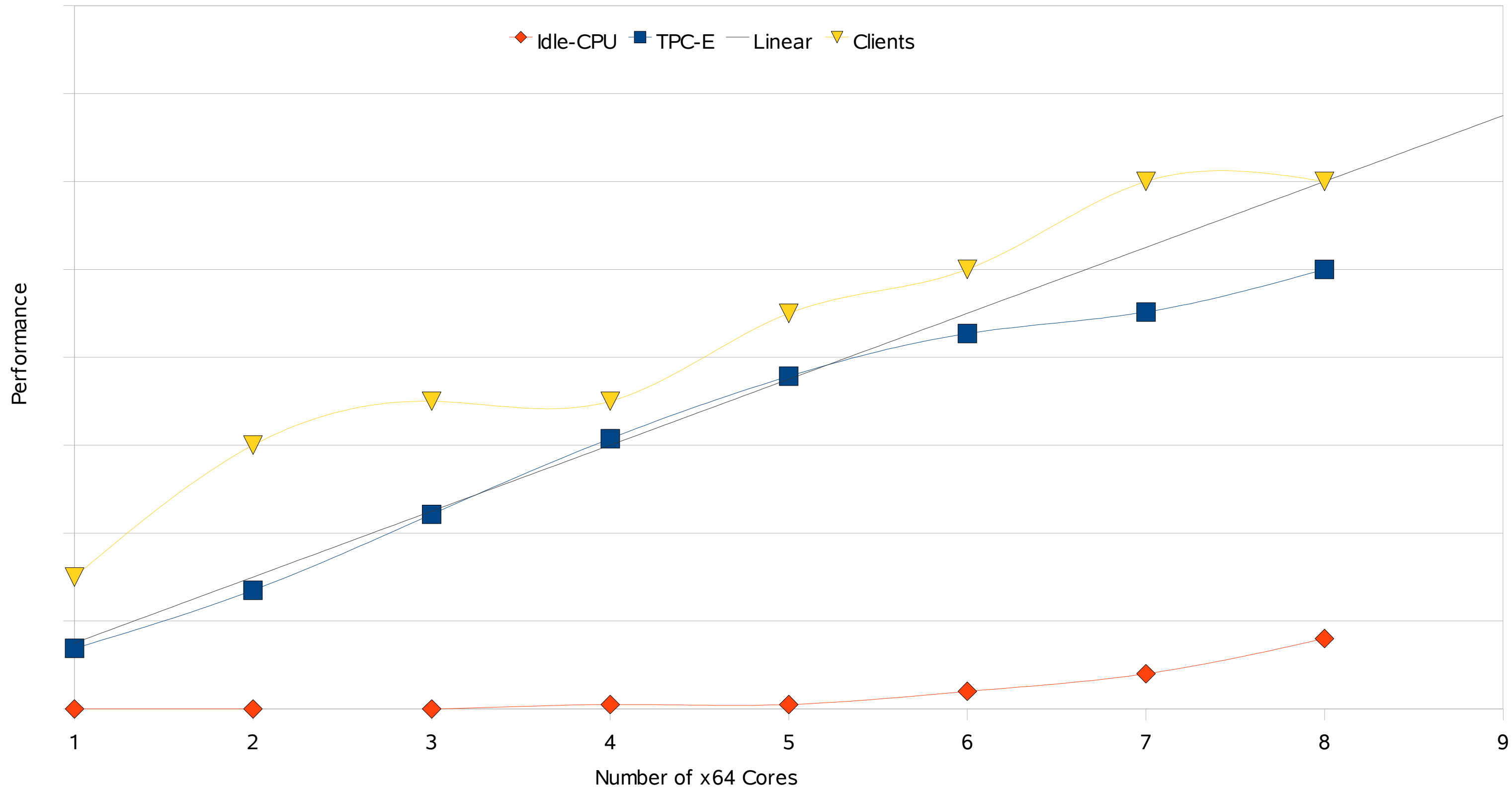
- IOPS on logs during regular pgbench run is around 800 w/s which means transaction optimizations happening somewhere
- With commit_delay (sibling =5) at 16 clients IOPS on logs is 102 w/sec which means quite a bit of capacity on logs yet
- With synchronous_commit=off wal_writer_delay=100ms, the iops on the log devices is 10 w/sec
- Same performance with wal_writer_delay=10ms (50w/sec on logs) and wal_writer_delay=1ms (100w/sec on logs)

PGBench (inserts only) – Take II



- At 128 clients system (16cores) was 58% idle (commit) , 56% idle (async)
- As more and more clients are added eventually performance seems to converge
- Runs with 256 clients and beyond using pgbench Clients pgbench running on a different server becomes cpu core limited and hence those results are not useful

TPC-E Like Workload with PG 8.3

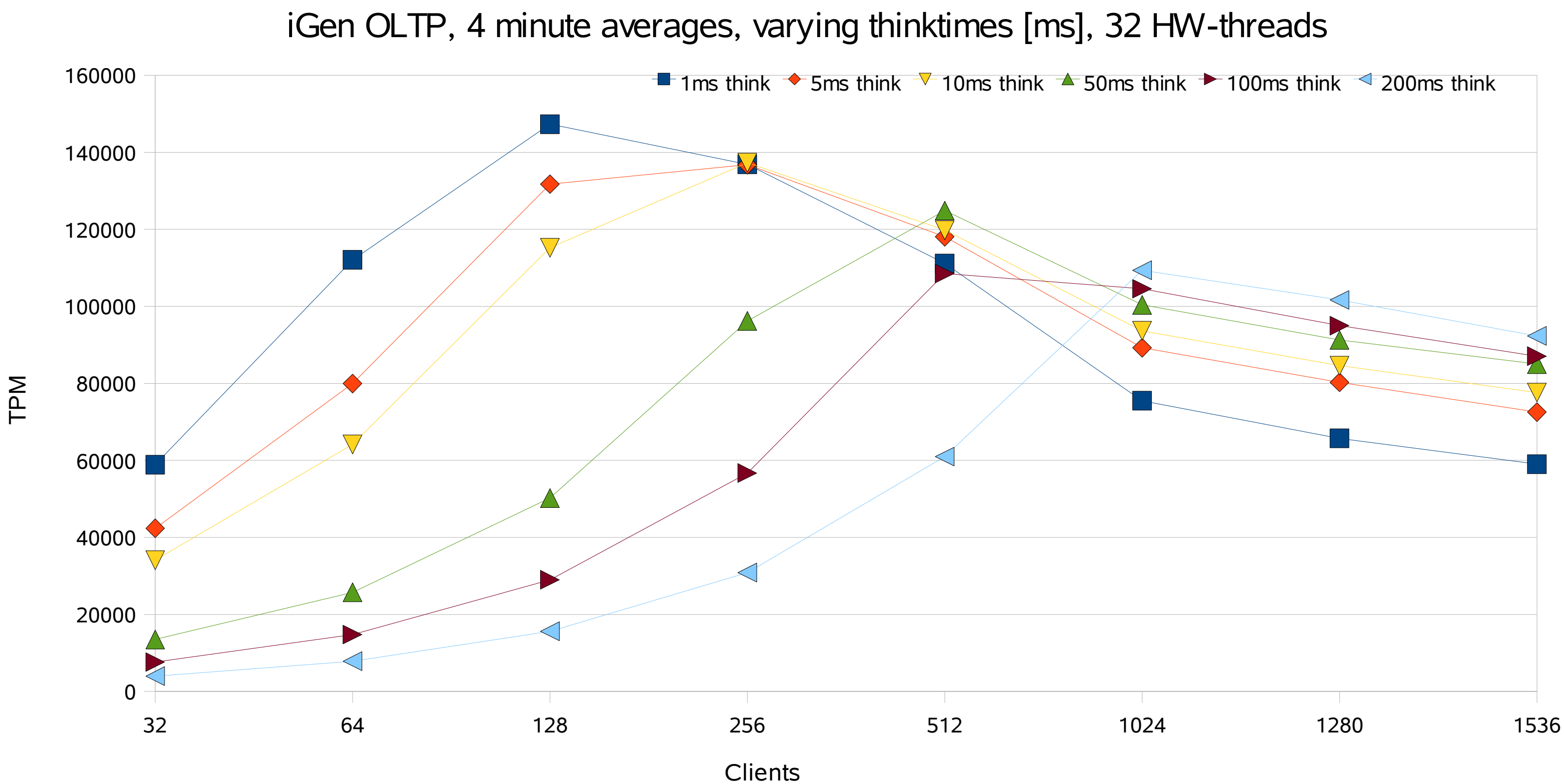


- At 7 cores number of clients increases beyond 110
- Increasing more cores beyond 8 doesn't seem to help much in terms of performance
- Quick dtrace shows ProcArrayLock Exclusive waits increasing while committing transactions at high loads

Some thoughts

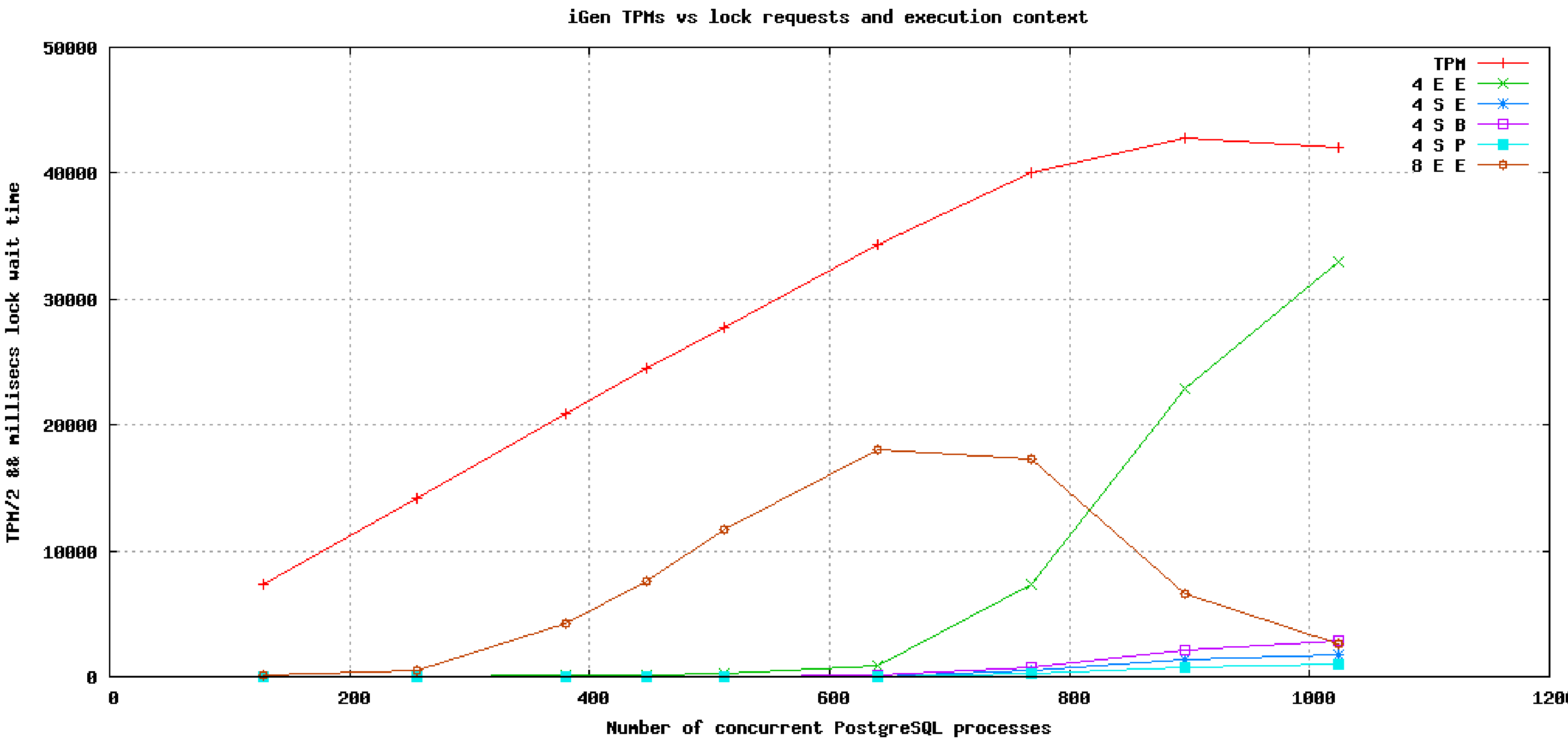
- Two main Blocks for PostgreSQL Backend:
  - READS
  - LOCK WAITS (Top few):
    - ProcArray Exclusive
    - Dynamic Locks (Shared) IndexScan
    - Dynamic Locks (Exclusive) InsertIndexTuples
- Reducing and modifying various indexes increased performance more than 2-3X but still limited due to core indexes required
- Haven't filtered out lock spins which keeps CPU busy (results in higher CPU utilization with more core without appropriate increase in throughput rates)

IGEN with PostgreSQL on T2000



- Sun Enterprise T2000 has 32 hardware threads of CPU
- Data and log files on RAM (/tmp)
- Database reloaded everytime before the run

IGEN with PostgreSQL on T2000



- First number is LockID, (Only 2 different locks pop up: ProcArrayLock == 4; WALWriteLock == 8)
- Second is Mode: S(hared)or E(xclusive) mode
- Third is Function P(arse), B(ind), E(xecute).
- Example: procarraylock in S(hared) mode while doing a B(ind) operation will be reflected in the graph 4SB

OLTP Workload on PostgreSQL

- Top Light Weight Locks having increasing wait times as connections increases
  - ProcArrayLock
    - EXCLUSIVE - Called from ProcArrayEndTransaction() from CommitTransaction()
    - SHARED - Called from GetSnapShotData()
  - WALWriteLock
    - XlogFlush()
  - WALInsertLock
    - XLogInsert()
-

ProcArray LWLock Thoughts

- ProcArray Lock currently has one wait list
  - If it encounters SHARED, it looks if the following wait-listed process is SHARED or not if it is wakes them up together
  - If it Encounters EXCLUSIVE, it just wakes up that process
- Multiple SHARED process can execute simultaneously on multi-core,
  - maybe a two wait-list (SHARED, EXCLUSIVE) or something schedule SHARED requests together might improve utilization
- Won't help EXCLUSIVE (which is the main problem)
  - Reduce Code Path
  - Use some sort of Messaging to synchronize

WALWriteLock & WALInsertLock

- WALWriteLock can be controlled
  - commit_delay=10 (at expense of latency of individual commit)
  - synchronous_commit = off (for non-mission critical types)
- WALInsertLock (writing into the WAL buffers) eventually still is a problem
  - Even after increasing WAL Buffers, its single write lock architecture makes it a contention point
- Making WALInsertLock more granular will certainly help scalability
  - Some discussion on reserving WAL Buffer space and releasing locks earlier

Impact on workloads with multi-terabyte data running PostgreSQL

Some Observations

- Its easy to reach a terabyte even with OLTP environments
- Even a single socket run for TPC-E could result close to about 1 TB data population
  - > http://tpc.org/tpce/tpce_price_perf_results.asp
- In some sense you can work around “writes” but “read” will block and random read can have real poor response time bigger the database size
- Blocking Reads specially while holding locks is detrimental to performance and scaling
-

Impact on Sequential Scan

- Sequential Scan rate impact depends on not only on storage hardware but also CPU intense functions which depends on updates done to table since last vacuum
  - Types of functions with high CPU usage during sequential reads: HeapTupleSatisfiesMVCC (needs Vacuum to avoid this CPU cost), heapgettup_pagemode, advance_* (count() fuction)
  - Blocking reads and then CPU intense functions results in inefficient usage of system resources which should be separated in two separate processes if possible
  - Hard to predict rate of scan during Sequential scan with PostgreSQL
    - Example: Before Vacuum: Sequential scan takes 216.8sec
    - After Vacuum: Same sequential scan takes 120.2sec

Impact on Index Range Scan

- Similar to sequential scan except still slower
- High CPU usage functions include index_getnext(), _bt_checkkeys, HeapTupleSatisfiesMVCC, pg_atomic_cas (apart from BLOCKS happening with read)
- Slow enough to cause performance problems
  - 26 reads/sec on index and 1409 reads/sec on table during a sample index range scan (with file system buffer on) Its really reads on tables that kills the range scan even when SQL only refers to columns in the index
  - 205 seconds Vs 102 seconds (via sequential) while doing primary key range scan
-

Tools Utilities for DBA

Think about the DBA

- Multicore systems means more end users using the database
- More pressure on DBA to keep the scheduled downtime window small
- Keeping DBA's guessing ("is it done yet?") while running maintenance commands is like testing the breaking point of his patience
- Example: VACUUM FULL -
  - Customer (DBA) reported it took 18 hrs to vacuum 3.4TB
  - VACUUM is just an example, all maintenance commands need to be multi-core aware designed to handle multi-terabyte data efficiently

Tools Utilities

- Tools are generally used more as a single task at a time
- Problems with Tools using a Single Process approach

Compute Intensive	IO Intensive
Maxes out 1 cpu/core/thread at a time Wasted CPU Resources Wated IO Resources	Uses 1 cpu/core/thread at a time Wasted CPU Resources

Resulting System Utilization very poor
Most people do not run other tasks while doing maintenance jobs

No indication when it will finish

BACKUP Performance

- pg_dump dbname
  - CPU limited with hot functions _ndoprnt, CopyAttributeOut, CopyOneRowTo, memcpy
  - Processing about 36MB/sec when CPU is saturated
  - Multiple pg_dump process could give about 91MB/sec which means if additional cores are used it could effectively help speed up backup
- Same goes for pg_recovery

VACUUM Performance

- We saw earlier state of last VACUUM is important for performance which means VACUUM is needed (apart from XID rollover)
- However VACUUM itself is very inefficient if there are cost_delays set
  - > Sample run on about 15GB table with vacuum_cost_delay=50:
  - > CPU utilization : 2% avg
  - > Took 3:03:39 @ 1.39 MB/sec
  - > Hot functions: heap_prune_chain(30%), lazy_scan_heap(14%), HeapTupleSatisfiesVacuum(14%)
- A heavily updated table can result in a bigger downtime just to get VACUUM completed on it

VACUUM Performance

- If costs for auto_vacuum are controlled and let DBA initiated VACUUM go full speed then (cost_limit=1, cost_delay=0)
- Hot functions include bsearch
  - Sample run on about 15GB table:
  - CPU utilization : 0-55% avg core
  - Took 0:18:8 @ 14.11 MB/sec
  - Hot functions: heap_prune_chain, hash_search_with_hash_value, heap_freeze_tuple
- Even with this a 1TB table could take about 20 hours
- Maybe help with some sort of pipelining reads through one process while processing it with another

CREATE INDEX Performance

- Dropping Index takes about 10 seconds
- However index creation is much longer
  - Depending on type of columns, the backend can process about 18MB/sec before its limited by core performance
  - Hot functions are btint8cmp (in this case) 50%, dumptuples (25%), comparetup_index (9.1)%, timestamp_cmp(3%)
- In this particular index it was index on an id and a timestamp field.
- On a table that takes about 105 second to do a full sequential scan, it takes about 1078 seconds to create an index (10X)

Summary/Next Step

- Propose more projects in making DBA utilities multi-process capable to spread up the work (eg VACUUM)
- Propose a separate background reader for sequential scans so that it can do processing more efficiently without blocking for read
- Propose re-thinking INDEX in multi-terabyte world

More Acknowledgements

- Greg Smith, Truviso - Guidance on PGBench
- Masood Mortazavi – PostgreSQL Manager @ Sun
- PostgreSQL Team @ Sun

More Information

- Blogs on PostgreSQL
  - > Josh Berkus: http://blogs.ittoolbox.com/database/soup
  - > Jignesh Shah: http://blogs.sun.com/jkshah/
  - > Paul van den Bogaard: http://blogs.sun.com/paulvandenbogaard/
  - > Robert Lor: http://blogs.sun.com/robertlor/
  - > Tom Daly: http://blogs.sun.com/tomdaly/
- PostgreSQL on Solaris Wiki:
  - > http://wikis.sun.com/display/DBonSolaris/PostgreSQL
- PostgreSQL Questions:
  - > postgresql-questions@sun.com
  - > databases-discuss@opensolaris.org

Q & A

Backup Slides/ Additional Information

TPC-E Characteristics

- Brokerage House workload
- Scale factor in terms of active customers to be used dependent on target performance (roughly Every 1K customer = 7.1GB raw data to be loaded)
- Lots of Constraints and Foreign keys
- Business logic (part of system) can be implemented via Stored Procedures or other mechanisms
- Can be used to stress multiple features of database: Random IO reads/writes, Index performance, stored procedure performance, response times, etc

TPC-E Highlights

- Complex schema
- Referential Integrity
- Less partitionable
- Increase # of trans
- Transaction Frames
- Non-primary key access to data
- Data access requirements (RAID)
- Complex transaction queries
- Extensive foreign key relationships
- TPC provided core components