Vacuum More Efficient Than Ever

Masahiko Sawada
NTT Open Source Software Center

@PGCon 2018

Who Am I?

- Masahiko Sawada
  - from Tokyo, Japan

- PostgreSQL contributor
  - Multiple synchronous replication: *FIRST* and *ANY* methods (9.6 and 10)
  - Freeze map (9.6)
  - Skipping cleanup index vacuum (11)

Agenda

- **What Is Vacuum?**
- **Three Vacuum Improvements**
  - Problems
  - Solutions
  - Challenges
  - Evaluations
- **Conclusion**

What Is Vacuum?

- **PostgreSQL garbage collection feature**
- **Recover or reuse disk space occupied**

- **VACUUM command**
  - =# VACUUM tbl1, tbl2;
  - =# VACUUM (ANALYZE, VERBOSE) tbl1;

- **Auto vacuum**
  - autovacuum launcher process
  - autovacuum worker processes

History of Vacuum Evolution

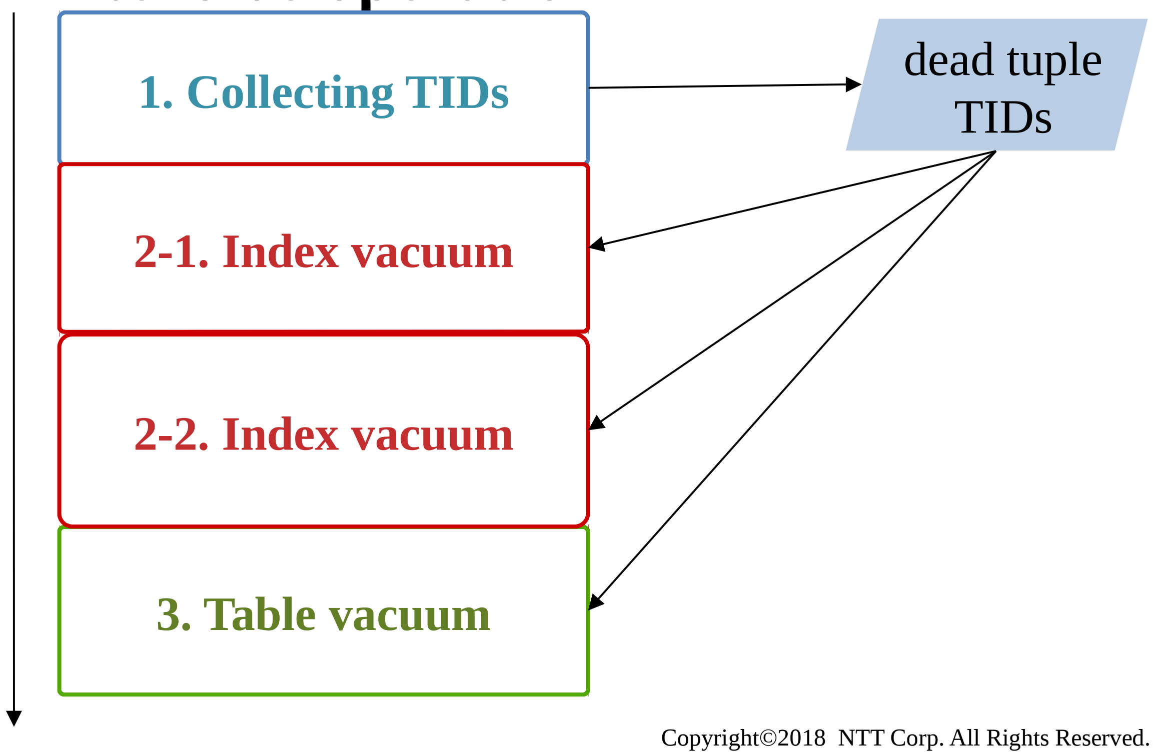
- **Auto Vacuum (8.1~)**
- **Vacuum Delay (8.1~)**
- **Visibility Map (8.4~)**
- **Freeze Map (part of visibility map) (9.6~)**
- **Skipping index cleanup (11~)**

What's Needed For "Good Vacuum"?

1. **Shorten the vacuum execution time**
   - Use resource as much as possible
   - Reduce the amount of work
   - Work in parallel
2. **But, reduce impact on transaction processing**
   - Work lazily

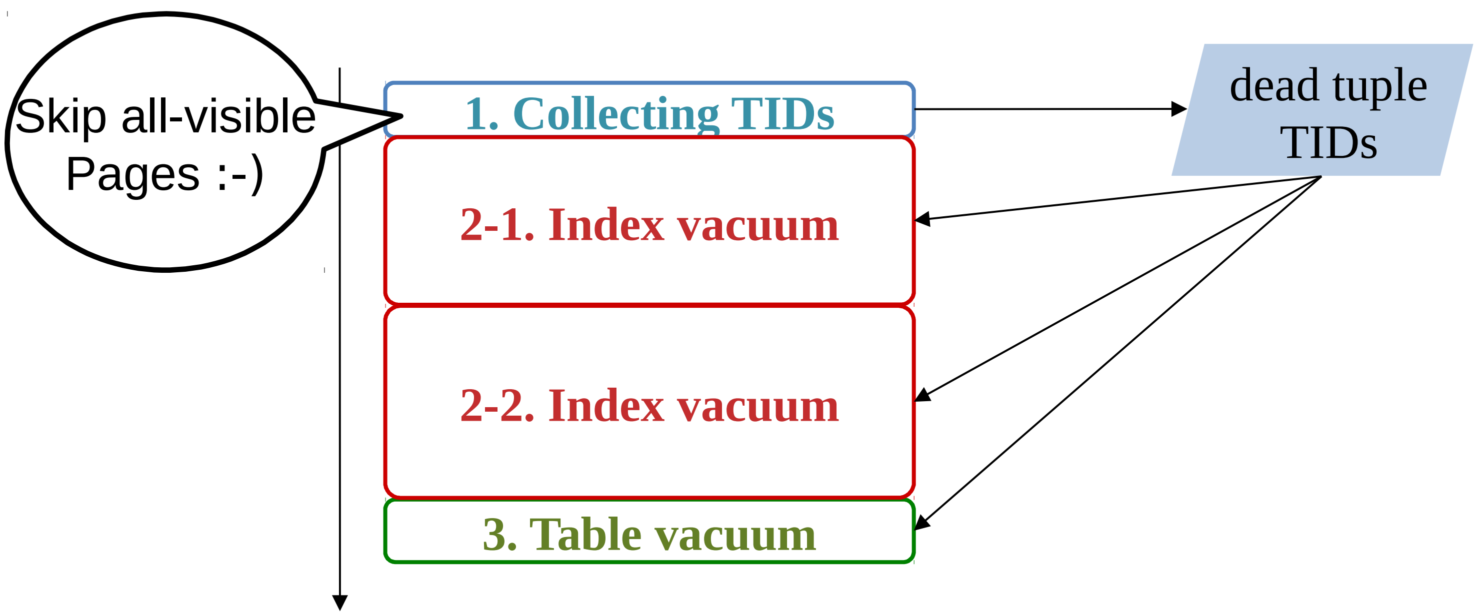
How Vacuum Actually Works

- **Vacuum works with three phases:**
  1. Collecting dead tuple TIDs till maintenance_work_mem amount of memory is consumed
  2. Vacuum indexes
  3. Vacuum table
- **Vacuum is a disk-intensive operation**



Vacuum With Visibility Map

- **2 bits per block: all-visible and all-frozen**
- **Track which pages "might" have garbage**
  - all-visible bit = 1 means the corresponding page has only visible tuples so we don't need to vacuum it



Factors Of Vacuum Performance

- Table size
- Number of indexes
- Resources

- Visibility map
- Vacuum delays (make lazy)
- Skipping index cleanup

Factors Of Vacuum Performance

- Table size
- Number of indexes
- Resources

Today's talk

- Parallel vacuum
- Deferring index vacuums
- Range vacuum

- Visibility map
- Vacuum delays (make lazy)
- Skipping index cleanup

Factors Of Vacuum Performance

Today's talk

- **Table size**
- **Number of indexes**
- **Resources**

- **Parallel vacuum**
- Deferring index vacuums
- Range vacuum
- Visibility map
- Vacuum delays (make lazy)
- Skipping index cleanup

PARALLEL VACUUM

On Very Large Table

- **Vacuum is performed by single process**
- **Vacuum could take a very long time**
  - Over days or even more!
- **Taking longer time if table has multiple indexes**

Current Solutions

- ✓ **Divide a large table**
- ✓ **Reduce autovacuum_delay_cost/limit**
  - Additional burden on the disk I/O instead

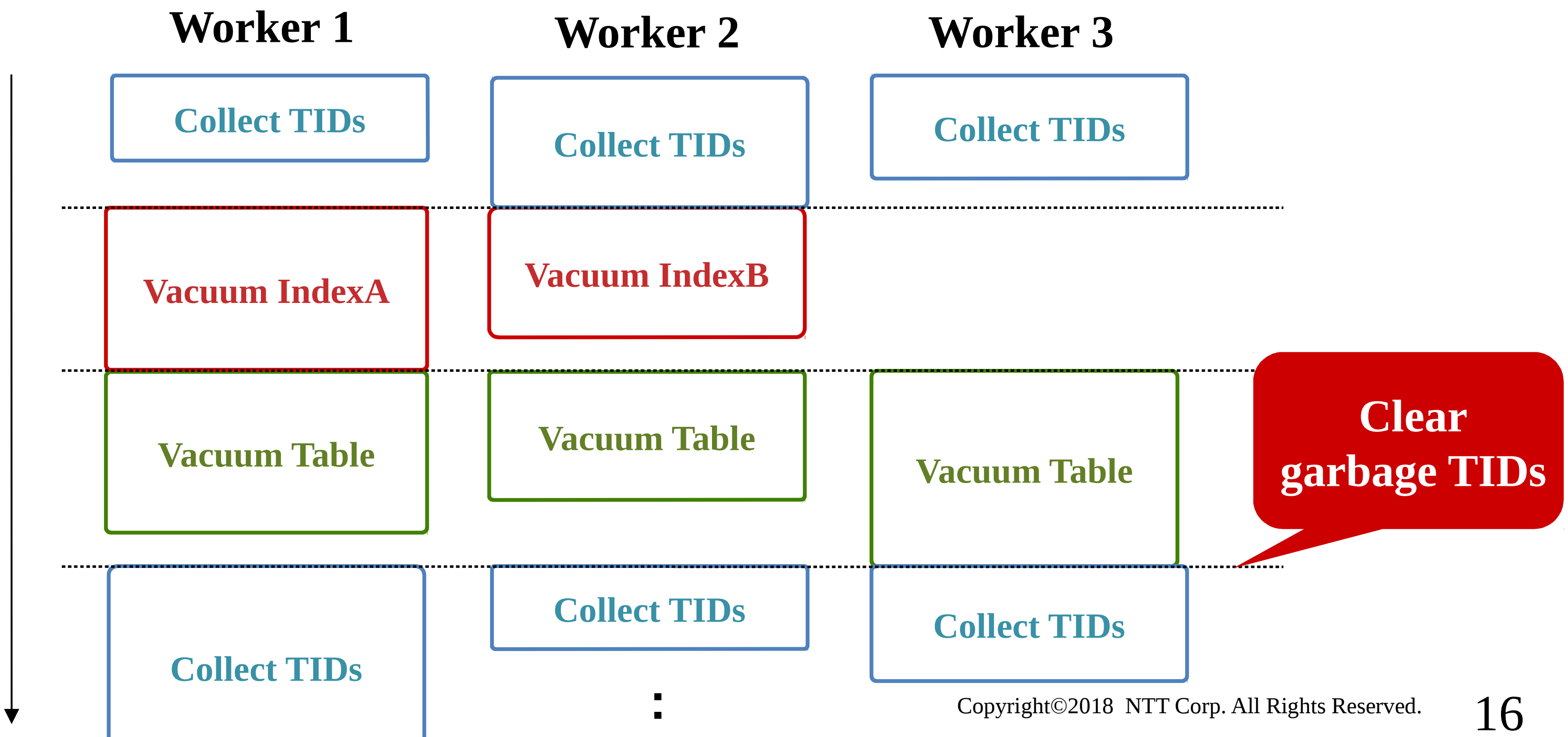
Idea: Parallel Vacuum

- **Execute vacuum with parallel workers**
- **Shorten the execution time of vacuum**
- **Note that this will consume more disk I/O**

- **A patch has been proposed**
  - “Block level Parallel Vacuum” (2016)
  - However, must resolve RelExt lock issue first
    - Please refer to “Moving relation extension locks out of heavyweight lock manager” (2016)

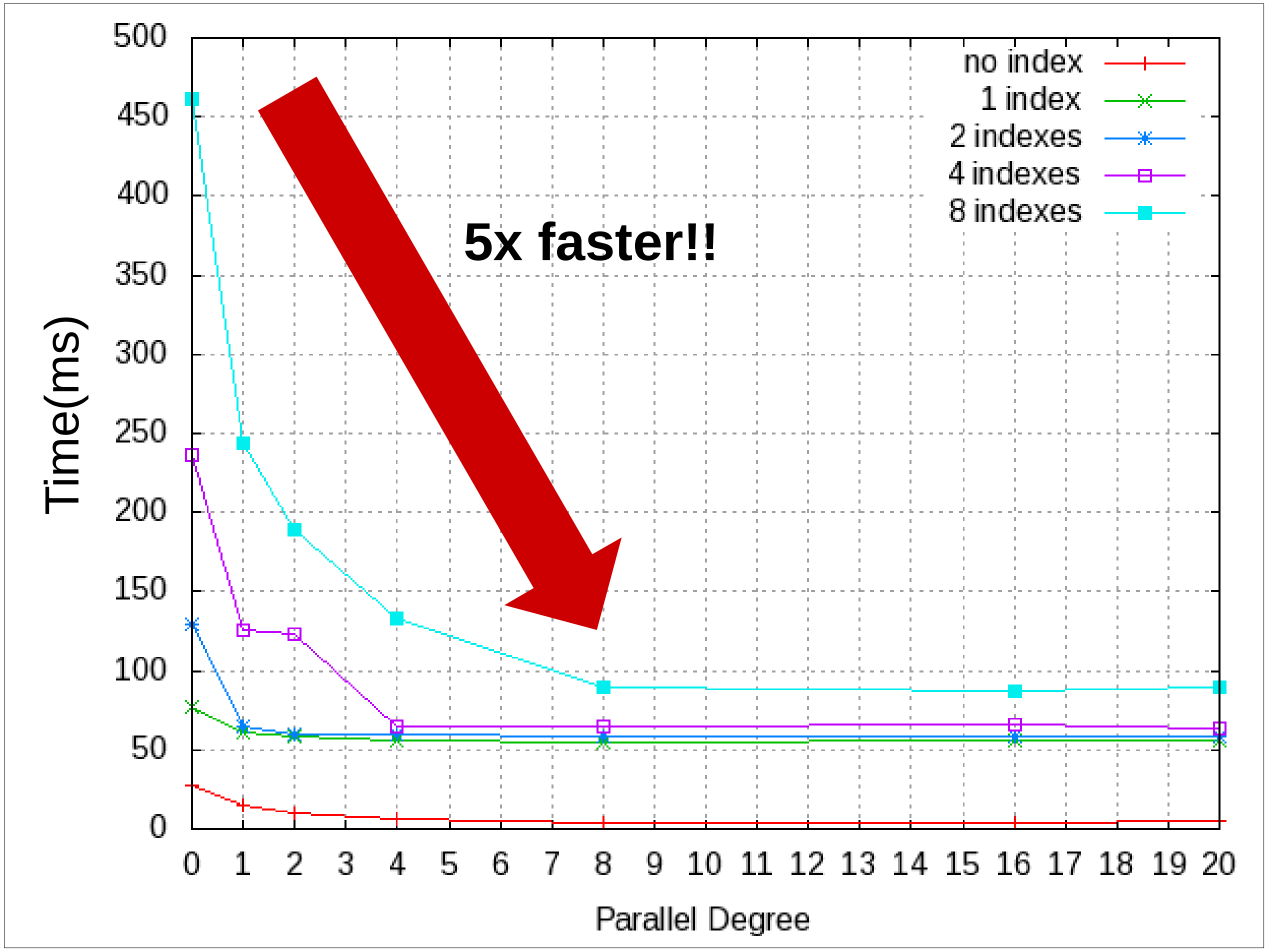
How Does It Work?

- Perform both TID collection and table vacuum with parallel workers
- Dead tuple TIDs are shared on the shared memory(DSM)
- Each index is assigned to a worker
- Make some synchronizations among workers



Evaluation (~8 indexes)

RAM : 32GB
shared buffers: 512 MB
Table : 4GB
Index vacuum : 1
ioDrive SSD : 256GB



Summary

- Parallel vacuum makes vacuums significantly faster
- This consume more CPUs and disk I/O
- Patch has been proposed
- Relation extension lock issue must be solved first!

Factors of Vacuum Performance

Today's talk

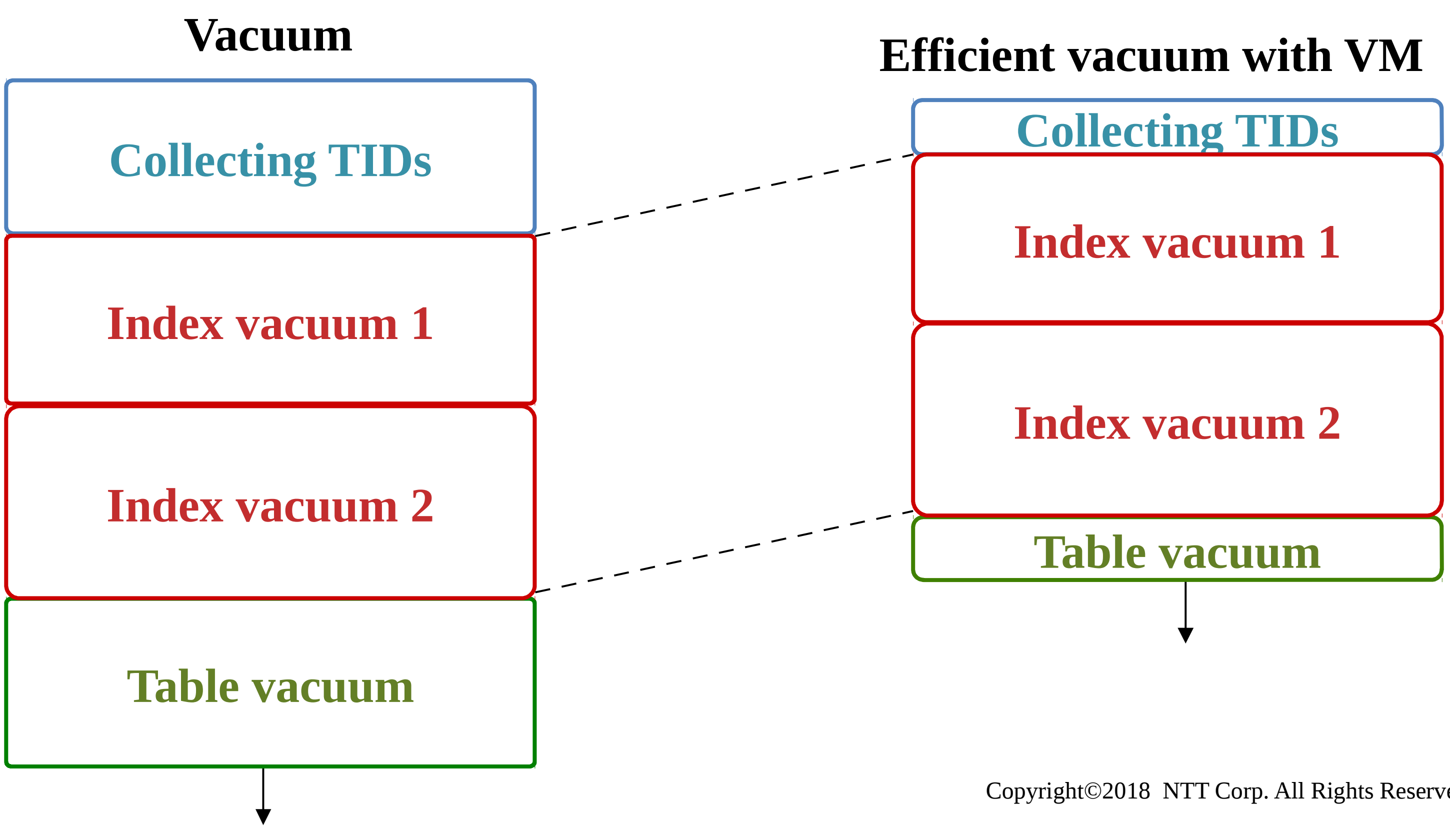
- Table size
- Number of indexes
- Resources

- Parallel vacuum
- **Deferring index vacuums**
- Range vacuum
- Visibility map
- Vacuum delays (make lazy)
- Skipping index cleanup

DEFERRING INDEX VACUUM

Looking Back To Analysis of Vacuum

- **Index vacuums could still be very long**
  - Table vacuum can be skipped by Visibility Map but index vacuum doesn't have such facility
  - Index vacuum could be invoked N times in a vacuum processing
- **Almost all index AMs require a full scanning on index**
  - Only 10 dead tuples in 1TB table requires whole index scans!



Current Solutions

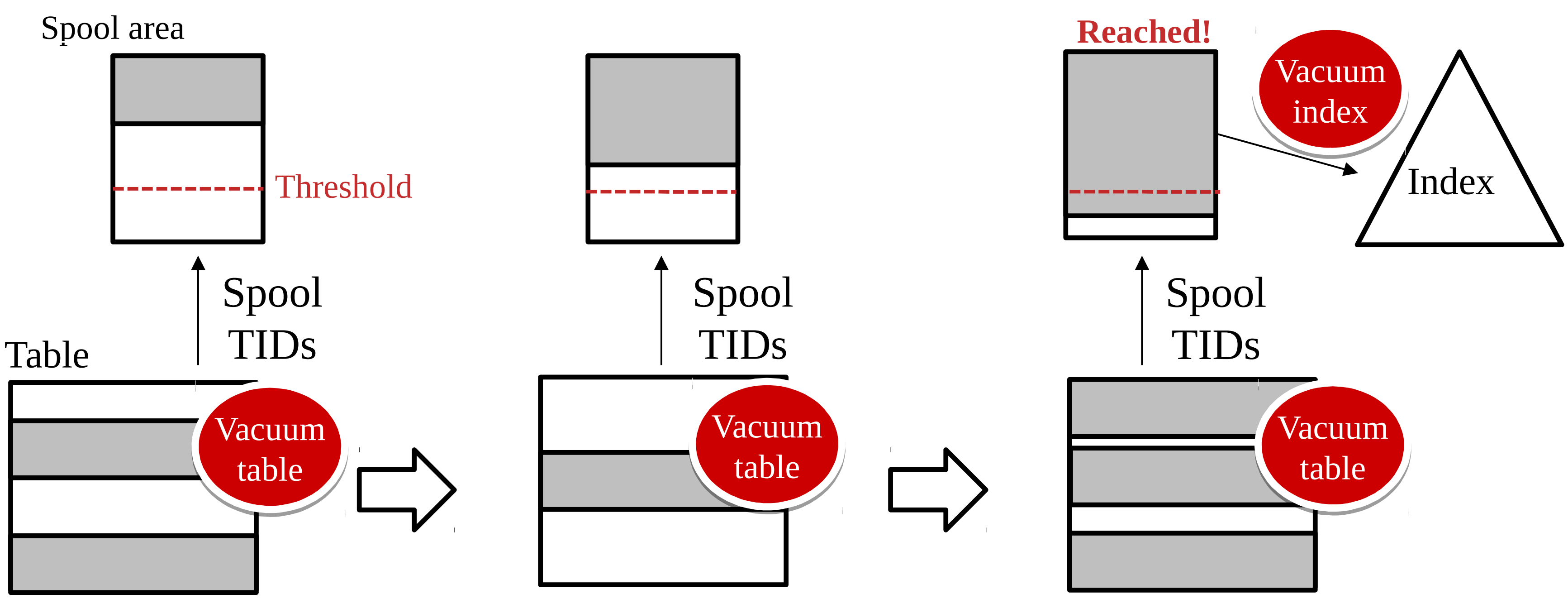
- ✓ **Don’t trigger auto-vacuum with a small threshold**
  - What about manual vacuum?
  - Indexes are not easy to bloat than tables
- ✓ **Increase maintenance_work_mem to avoid calling index vacuuming multiple times**
  - However, still requires index vacuum at least once

Idea: Deferring Index Vacuum

- **Spool garbage TIDs**
- **Don't trigger index vacuum unless the amount of spooled garbage TIDs reached to the threshold**
- **Reduce the number of index vacuum**

How Does It Work?

- **Amount of garbage TID < threshold**
  - Vacuum only table and spool dead tuple TIDs
- **Amount of garbage TID >= threshold**
  - Vacuum indexes



Related Discussions

- **There are related discussions**
  - “Proposal: Another attempt at vacuum improvements” (2011)
  - “Single pass vacuum – take1” (2011)
  - But it breaks on-disk format

Evaluation

- **Evaluate the performance improvement by reducing the number of index vacuums**
  - Spool garbage TIDs to DSM
  - When bulk-deletion we look up both collected TIDs and spooled TIDs
- **Introduce new storage parameter vacuum_index_defer_size which controls how much dead tuples can be spilled out**

- **However, don’t care about concurrent update and durability :(**

Evaluation

```
=# \dt+
                    List of relations
 Schema |     Name     | Type  |  Owner   |  Size   | Description
--------+--------------+-------+----------+-------- +------------
 public | defer_table  | table | masahiko | 3458 MB |
 public | normal_table | table | masahiko | 3458 MB |
(2 rows)


-- Spool size is 100kB
=# ALTER TABLE defer_table SET (vacuum_index_defer_size = 100);

-- Disable deferring index vacuum
=# ALTER TABLE normal_table SET (vacuum_index_defer_size = 0);
```

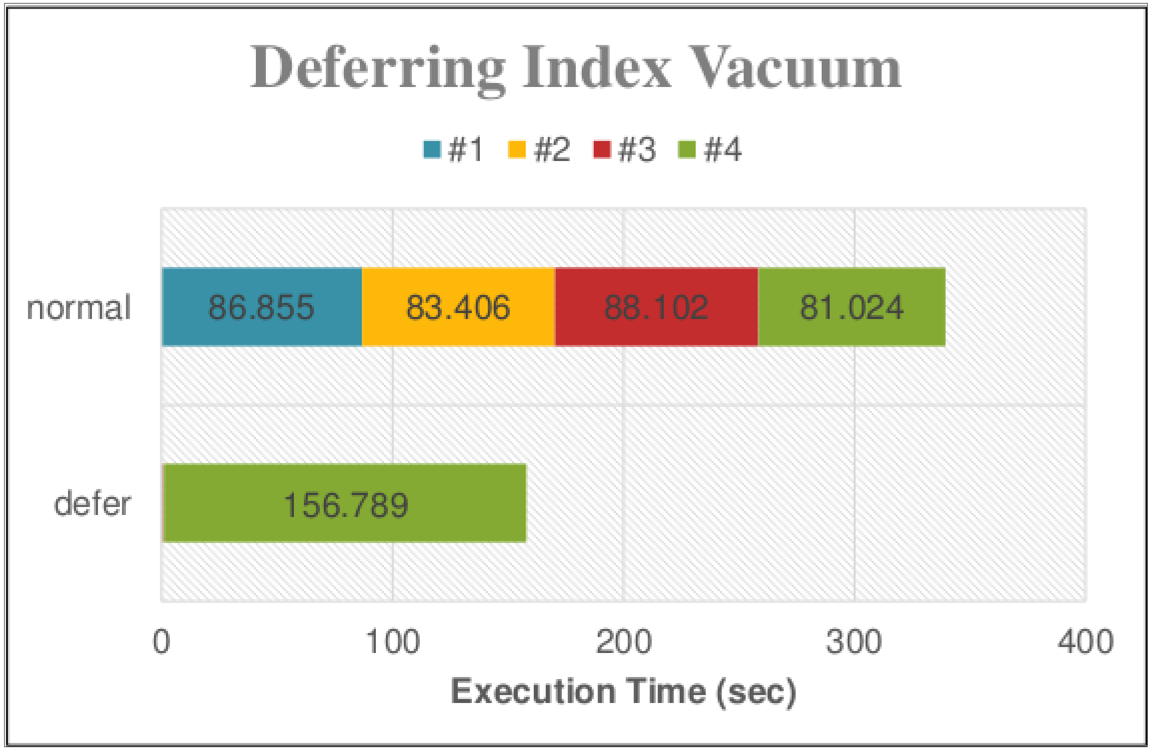
Evaluation

1. **Load data**
2. **Vacuum table to make VM**
3. **Loop until the amount of garbage reached to the threshold (= 17000 tuples)**
   1. Delete 5000 tuples to make garbage
   2. Vacuum

Vacuum will be performed 4 times, and index vacuum will be executed at only the 4th vacuum

Evaluation

- **Skipped index vacuum at 1st, 2nd and 3rd vacuum**
- **Deferring index vacuum made vacuum 2.1x faster**
- **At the 4th vacuum, deferring index vacuum took twice time than the normal**
  - Looking up the collected TIDs as well as the spooled TIDs



Summary

- **Deferring index vacuum have potentials of speed up vacuums much**
  - In this evaluation, it speeds up 2.1x faster
- **More tricks are required for the correct implementation**
  - To prevent vacuumed item pointers from being reused before index vacuum
  - To avoid breaking on-disk format

Factors Of Vacuum Performance

Today's talk

- **Table size**
- **Number of indexes**
- **Resources**

- **Parallel vacuum**
- **Deferring index vacuums**
- **Range vacuum**

- **Visibility map**
- **Vacuum delays (make lazy)**
- **Skipping index cleanup**

RANGE VACUUM

Dilemma

DBA wants to complete vacuum as quickly as possible

DBA wants to avoid both disk I/O bursts and affecting to TPS by vacuum as much as possible

Long-running Vacuum Problems

- Long-running vacuum likely to be canceled
- Restart vacuum from the beginning of the table again
- Cannot reclaim garbage that is made since the vacuum started

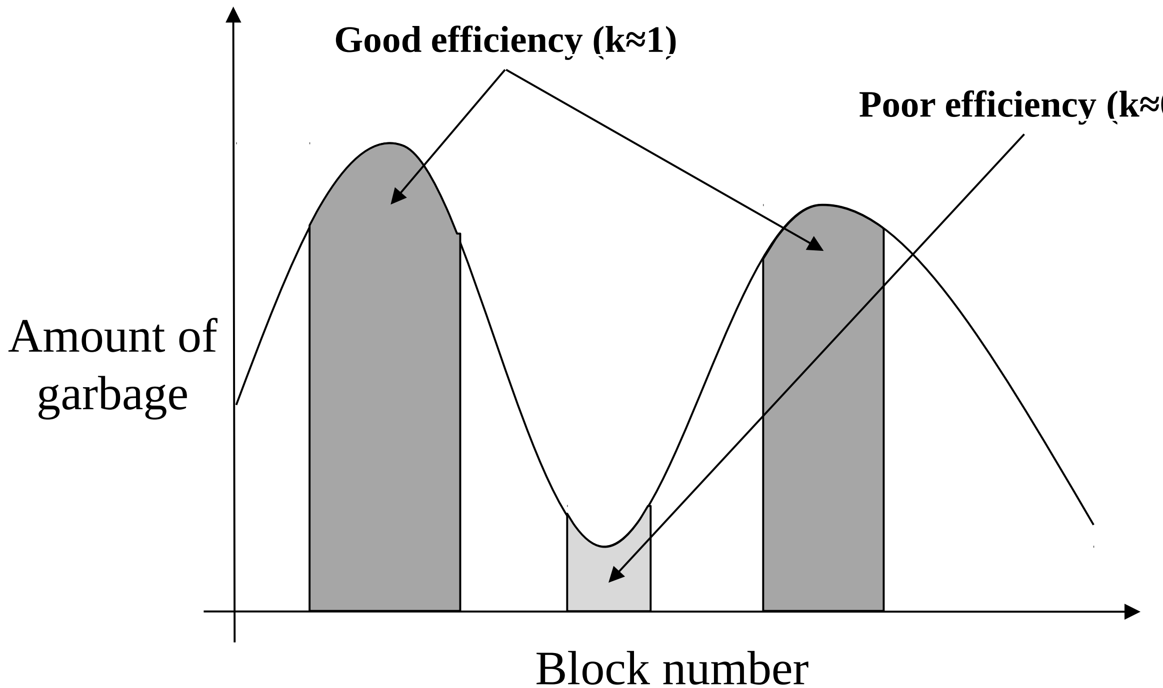
Is it possible to use vacuum delays and to complete vacuum in a short time?

Efficiency Analysis of Vacuum

- **The cost of vacuum a block can be regard as almost constant**
  - The most spent time is disk I/O (read buffer, write WAL)
- **Garbage on table might have locality**
- **Even though vacuum reclaims a block the new free space got by vacuum depends on how much garbage exists on the block**

Efficiency Analysis of Vacuum

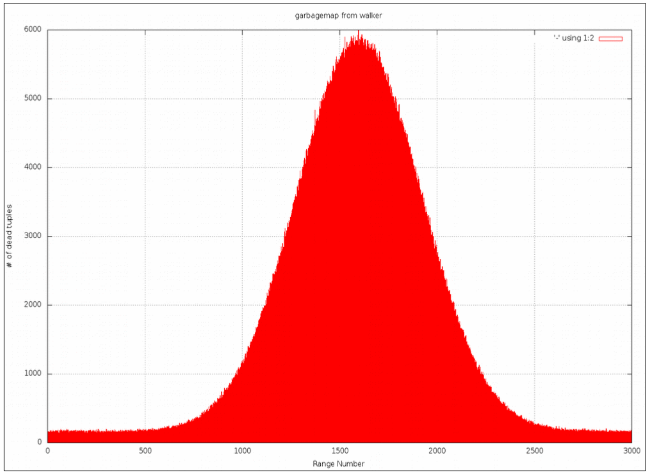
- **If we got free space N byte by vacuum on M byte, efficiency of vacuum k is N/M**
  - k = 1 means we get free space as mush as we vacuumed
  - k ≈ 0 means we don’t get free space even if vacuumed lots of blocks
- **All-visible of VM is cleared if even one tuple is inserted/deleted**



Range Vacuum with Garbage Map

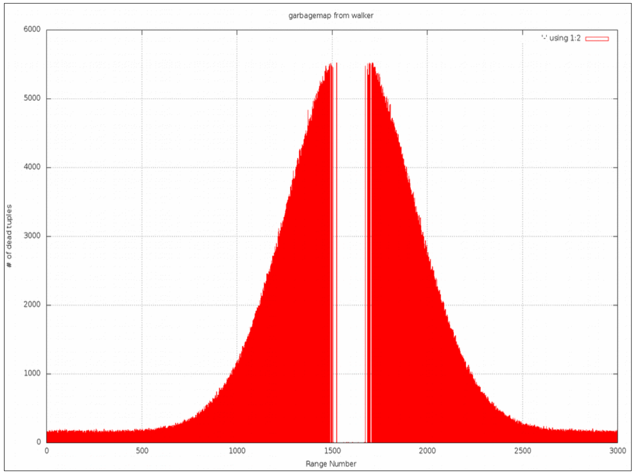
- Garbage map
  - Track garbage status of bunch of blocks
  - Reproduce the garbage status on table
- Range vacuum
  - Preferably vacuum blocks having higher “k”
  - Trigger vacuum more frequently

Before



Vacuum higher 10% ranges

After

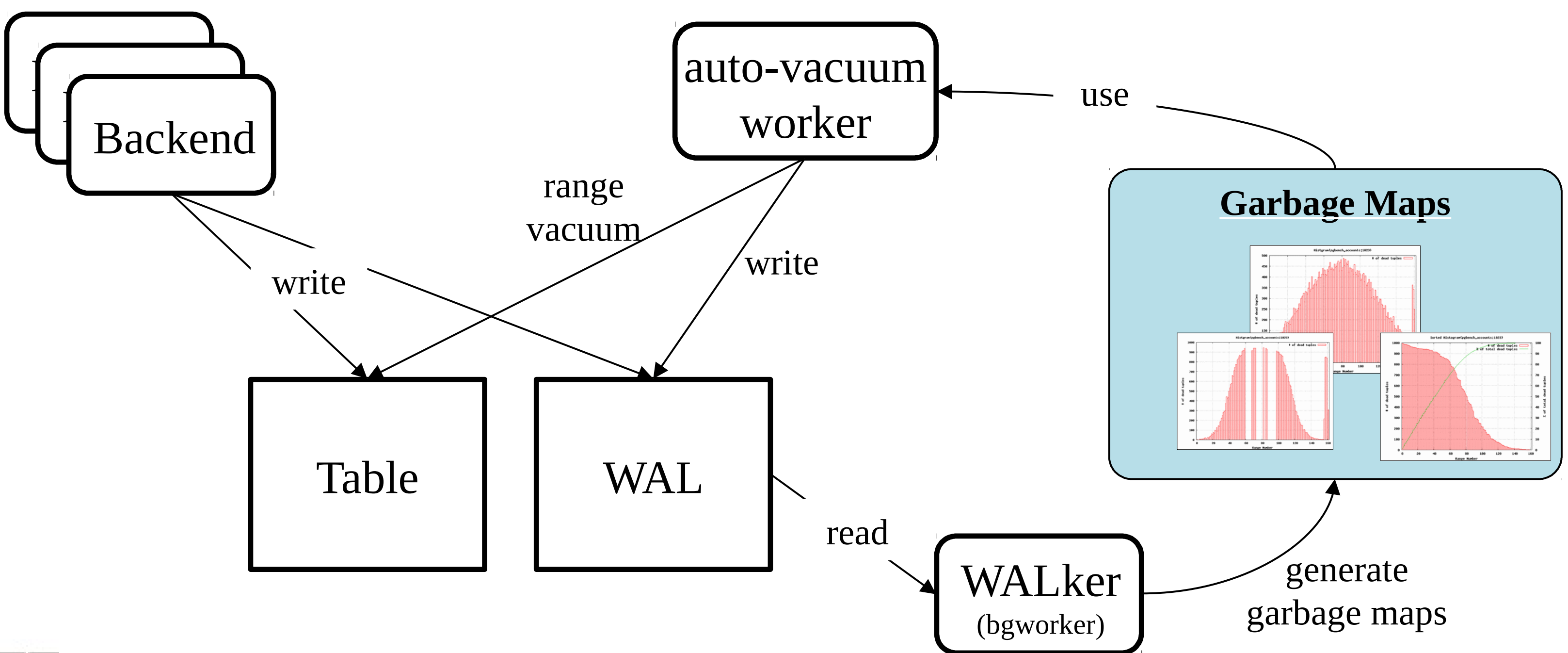


Building Garbage Maps

- **WAL-based**
  - WAL knows the all block change information
  - Don’t increase transaction latency as mush as possible
- **Logical decoding didn’t match (so far)**
  - Need to track block-level changes
  - Need to track aborted transactions
- **“WALker” module**
  - A background worker that continues to read WAL
  - Invoke corresponding plugin callbacks
    - “garbagemap” plugin builds garbage maps
  - Repository at https://github.com/MasahikoSawada/walker

Garbage Map Details

- **Divide a table by 4096 blocks (32MB) logically into ranges**
  - Track of # of garbage tuples per range by integer. 4MB for 2^32 blocks.
- **Reorder transaction information and make garbage maps**
  - In a commit transaction, deleted tuples become garbage tuples
  - In a abort transaction, inserted tuples become garbage tuples
- **Vacuum only ranges having higher efficiency**
  - Also added the lower bound of using range vacuum

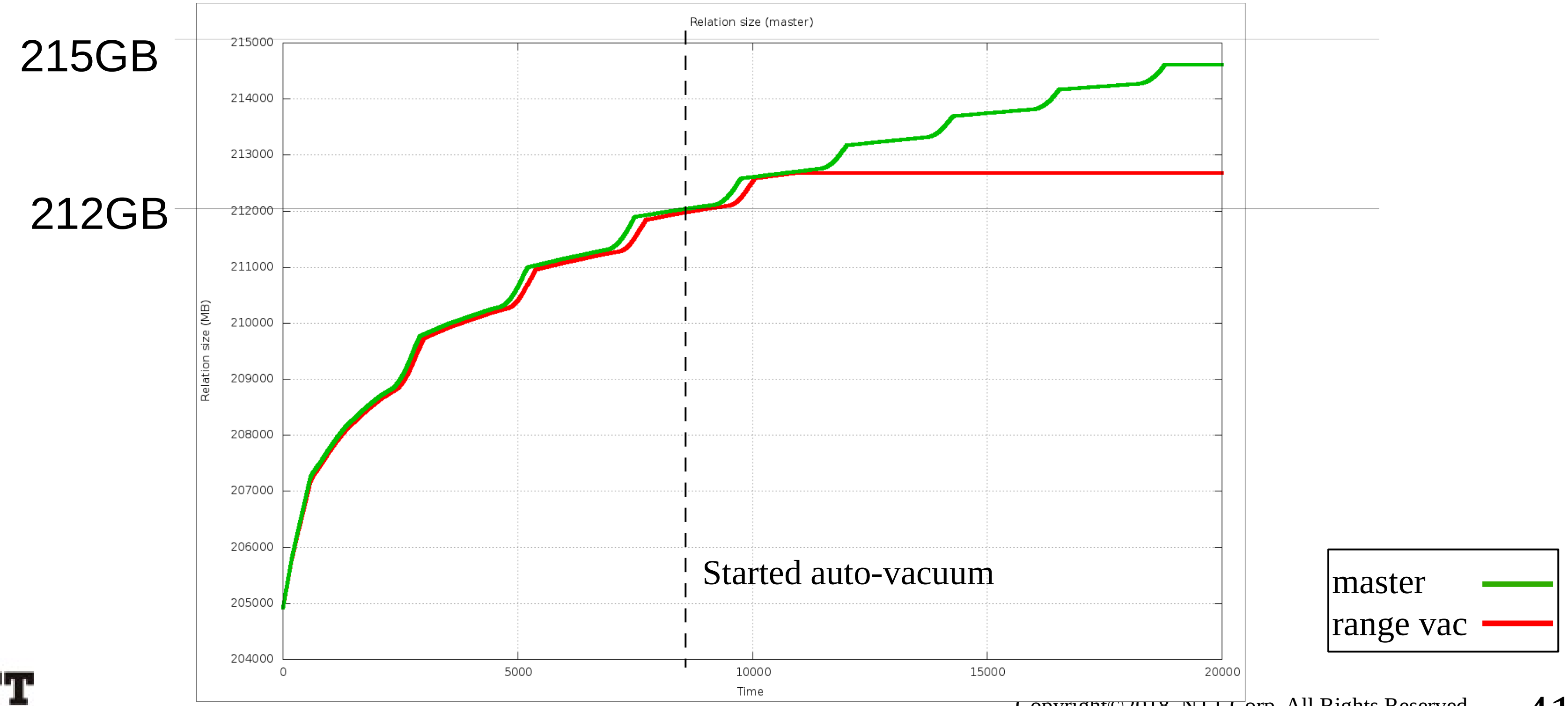


Evaluation

- **Machine**
  - 144cores, 126GB RAM, 1.5TB SSD
- **Target**
  - master branch (ff49430 snapshot) and with range vacuum feature
- **Configurations**
  - autovacuum_vacuum_scale_factor = 0.04
  - **autovacuum_vacuum_cost_limit = 1000 (default is 200)**
  - autovaucum_vacuum_cost_delay = 20ms (by default)
- **Workload**
  - pgbench (TPC-B) at scale factor 16000 (about 200GB)
  - Using custom script (gaussian : uniformly = 9 : 1)
  - 5 hours
  - **Run open-transaction for 10 min with 30 min intervals (to generate garbage faster)**
- **Observation**
  - Transaction TPS
  - Transaction latency
  - Relation size

Results: Relation size

- **auto-vacuum started about 2 hours after**
- **Master branch**
  - Didn’t complete auto-vacuum within 5 hours
  - Took over 9 hours (not recorded)
- **Range vacuum**
  - Run 6 times
  - Processed 800 ranges (27GB, 10% of table) within 50min at an average

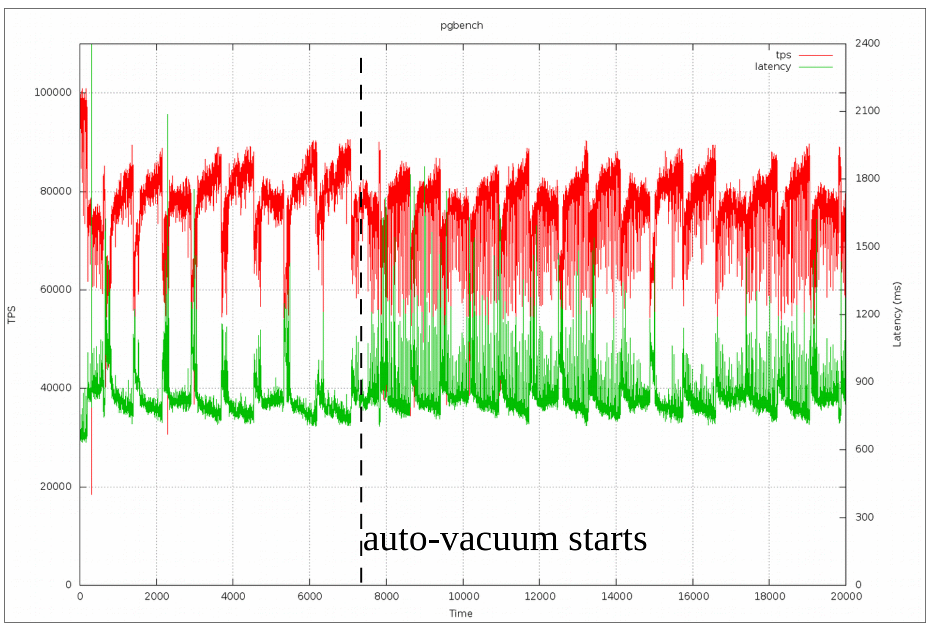


Results: TPS and latency

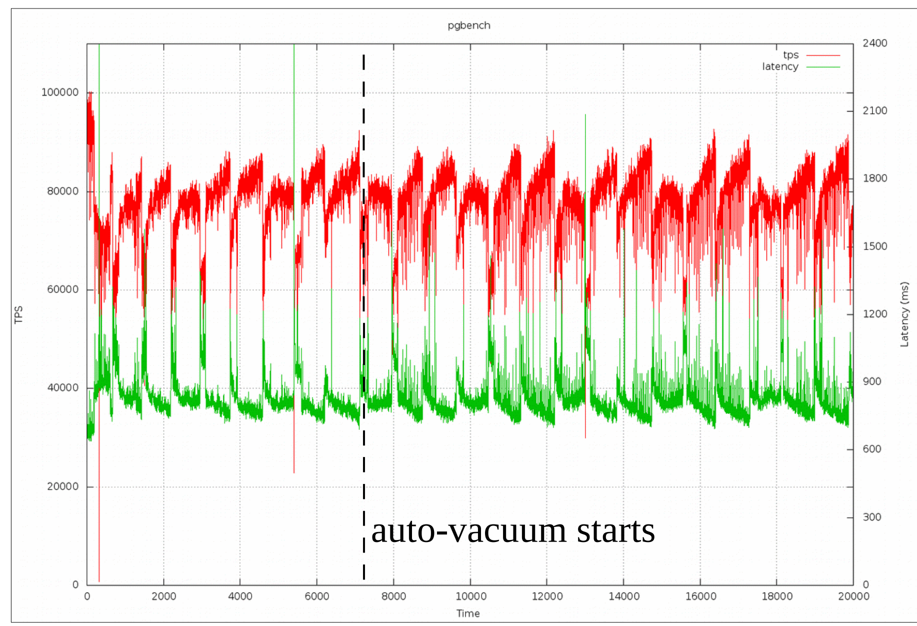
- In master branch, latency became sometimes large after auto-vacuum started
  - Frequently updated blocks likely to be loaded to shared buffer
- TPS and latency of range vacuum branch was more stable

TPS
Latency

Master



Range Vacuum



Summary

- **Range vacuum reclaims garbage space with minimum side-affects in a short time**
- **Invoking range vacuum more frequently also means calling index vacuum more frequently as well**
  - Combining with deferring index vacuum would be good idea
- **Each range has the number of garbage tuples**
  - Could be the size of garbage instead
- **Need to vacuum whole table if garbage placed uniformly on the table**

Conclusion

- **Improvement ideas**
  - Parallel vacuum
  - Deferring index vacuum
  - Range vacuum and garbage map

- **More improvement points**
  - Auto vacuum scheduling
    - Patch is proposed
  - Resource managements
    - Using cgroups
  - etc

Thank you!

Masahiko Sawada
Mail: sawada.mshk@gmail.com
Twitter: @sawada_masahiko

Spooling Dead Tuples TIDs

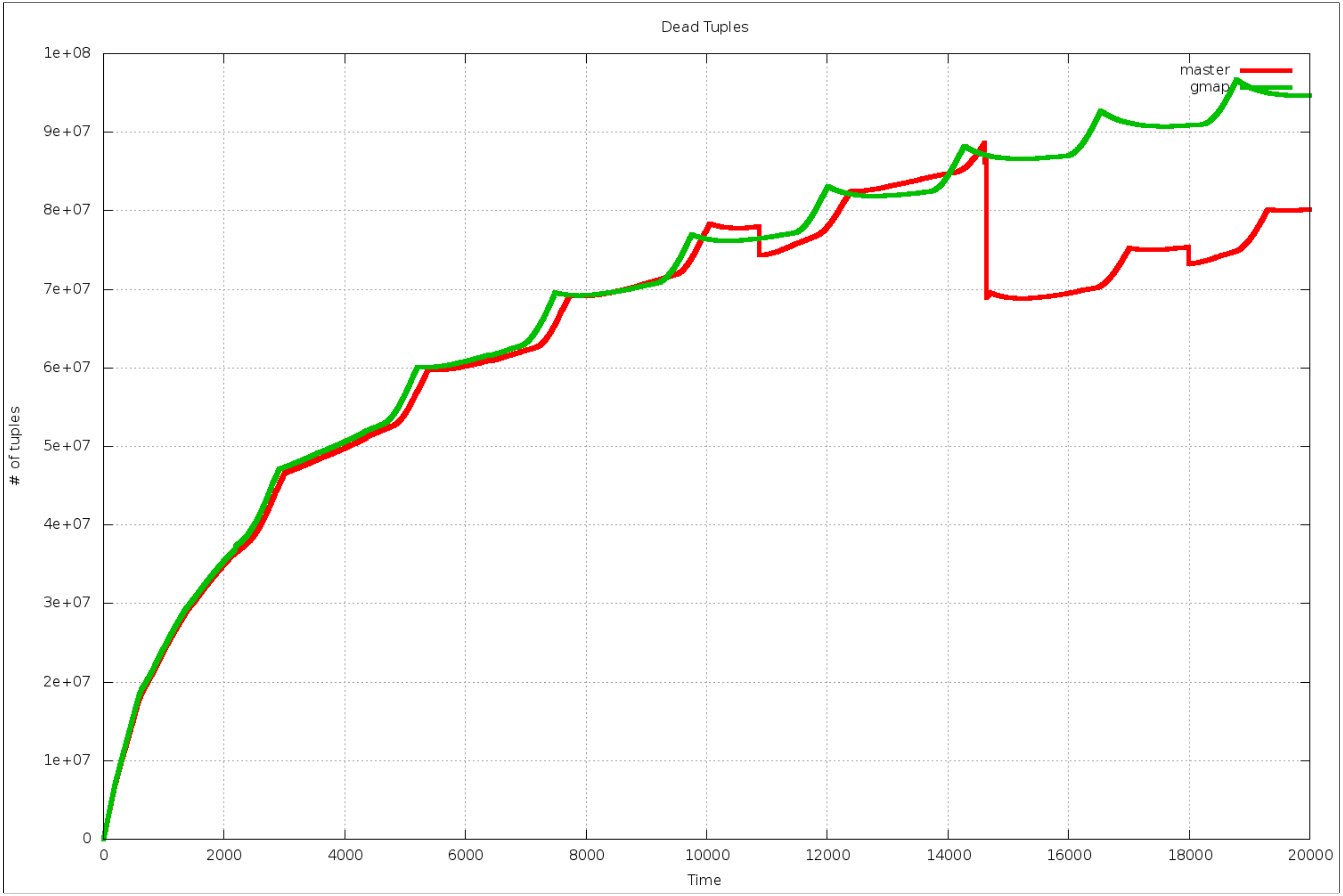
1. **HOT-pruning and table vacuum mark all item pointers that are being pointed by index tuple as VACUUM_DEAD**
2. **Spool dead tuple TIDs as bitmap per block**
3. **In an index vacuum, scan each index pages and check if index tuples are pointing to spooled dead tuple TIDs**
4. **Reclaim matched index tuples and clear corresponding bits**
   - If all bits are cleared, record LSN where index vacuum invoked along with bitmap
5. **At HOT-pruning or vacuum, mark VACUUM_DEAD item pointers as UNUSED if current LSN > stored LSN**

- **Data representation of dead tuple TIDs**
  - dead tuple TIDs are stored into a new fork <relfilenode>_dt
  - 300 bits (25 byte) for bitmap and 8 byte for LSN per 8kB block
    - 1 dt page has 234 blocks information
    - 1GB table -> 4MB dt fork, 1TB -> 4GB dt fork
  - To existing check faster, before starting index vacuum create bloom filter for blocks of which has any bits.

Configurations

- autovacuum_vacuum_scale_factor = 0.04
- autovacuum_naptime = 10
- autovacuum_vacuum_cost_limit = 1000
- autovacuum_vacuum_cost_delay = 20ms
- checkpoint_completion_targt = 0.3
- garbagemap.min_range_vacuum_size = 10GB
- garbagemap.range_vacuum_percent = 30
- shared_buffers = 50GB
- max_wal_size = 100GB
- min_wal_size = 50GB

Dead Tuples



Dead Tuples
1e+08
9e+07
8e+07
7e+07
6e+07
5e+07
4e+07
3e+07
2e+07
1e+07
0
# of tuples
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
Time
master
gmap