In-Memory Table with Pluggable Storage API - Highgo Software Inc.

Cary Huang

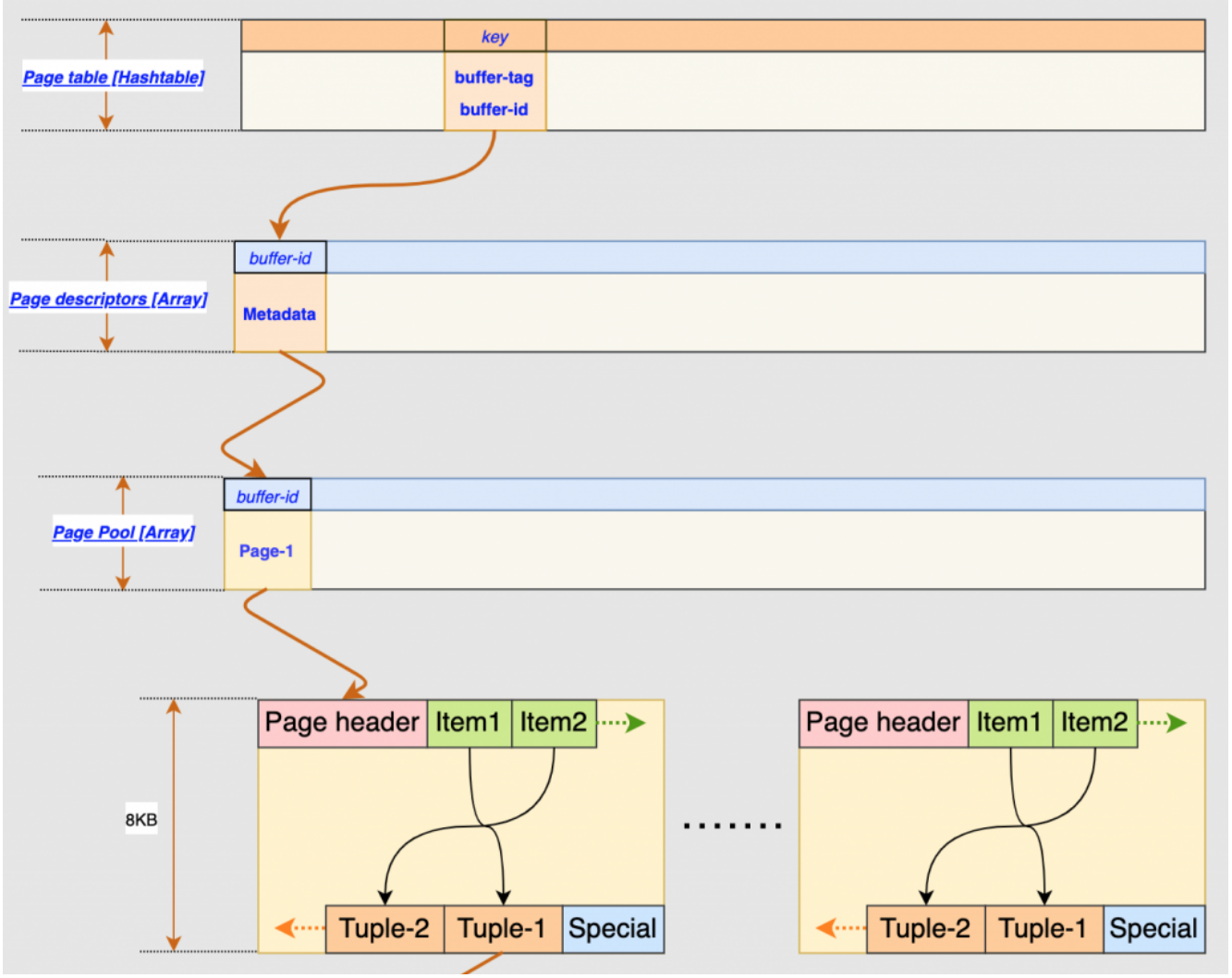
1. Introduction

This blog is to follow up on the post I published back in July, 2020 about achieving an in-memory table storage using PostgreSQL's pluggable storage API. In the past few months, my team and I have made some progress and did a few POC patches to prove some of the unknowns and hypothesis and today I would like to share our progress.

2. The PostgreSQL Buffer Manager

In the previous post, I mentioned that we would like to build a new in-memory based storage that is based on the existing buffer manager and its related component and hooked it up with the pluggable storage API. To achieve this, my team and I underwent an in-depth study to understand how the current buffer manager works in PostgreSQL and this chapter at interdb.jp is a good starting point for us to gain a general understanding of the buffer manager design in good details.

The current PostgreSQL buffer manager follows a 3-layer buffer design to manage the data pages as illustrated by this image below:



where it consists of

- Buffer Table (hash table)
- Buffer Descriptors (Array)
- Buffer Pool (Array)

2.1 Page Table

Buffer Table is used like a routing table between PostgreSQL Core and the buffer manager. It is managed using the existing hash table utilities and uses `buffer_tag` to look up the page descriptor and buffer id. Buffer_tag is a structure that contains the table space, database, table name.

2.2 Buffer Descriptor

Buffer Descriptor is used to store the status of a buffer block and also the content lock. Refcount is a part of the buffer state, will be used to indicate the insert and delete operation. it will be increased by one when there is an insertion, and decreased by one when there is a deletion. The Vacuum process will reclaim this page once refcount reaches to 0.

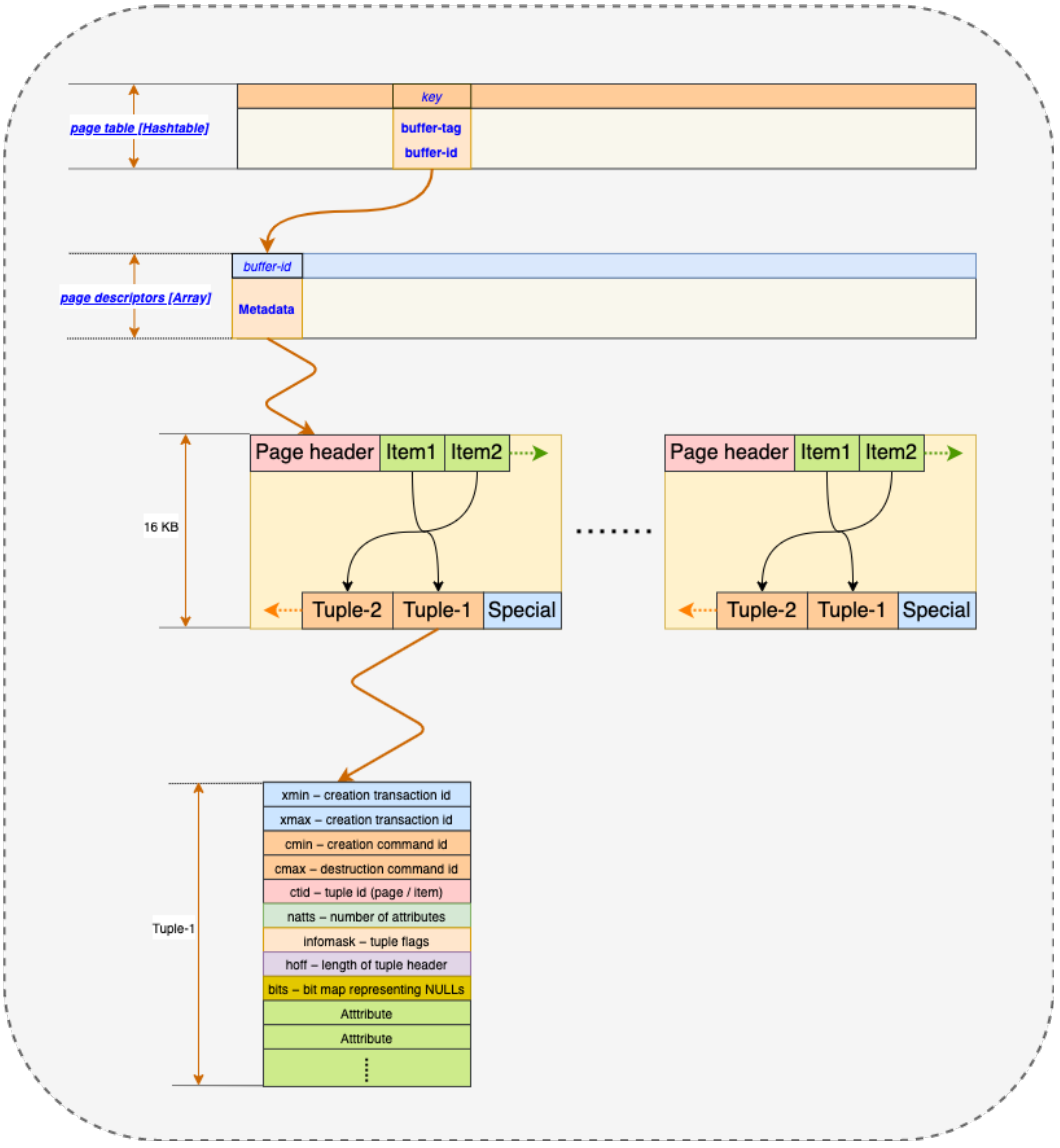
2.3 Buffer Pool

Buffer Pool has a one to one relationship with buffer descriptor. it can be treated a simple pointer pointing to the beginning of the buffer pool, each buffer pool slot is defined as 8KB for now. This is the lowest layer in the buffer manager structure before a page is flushed to disk. The `BM_DIRTY` status flag is used to indicate if a page in the buffer pool is to be flushed to disk

In addition to buffer pool, buffer manager also utilizes a ring buffer for reading and writing a huge table whose size exceeds 1/4 of the buffer pool size. `Clock Sweep` algorithm is used to find a victim page in the ring buffer to eject and flush to disk so new page can enter, thus the name, ring buffer.

3. The In-Memory Only Buffer Manager

Having a general understanding of the existing buffer manager's strucutre, we hypothesize that we could potentially improve its IO performance by eliminating the need to flush any buffer data to disk. This means that the in-memory only version of buffer manager itself is the storage media. For this reason, its strucutre can be simplified as:



where the buffer descriptor points to a dedicated memory storage that contains the actual page and tuple. This memory space can be allocated to a certain size at initlaization and there will not be a need to flush a page to disk. All data page and tuple will reside in this memory space. In the case where there is a huge reading and writing load, the ring buffer will not be allocated as the logic to find a victim page to evict and flush to disk will be removed since everything will reside in a dedicated memory space. For this reason, if the memory space is not sufficiently allocated, the user will get "no unpin buffer" is available, which basically means "your disk is full" and you need to do delete and vacuum.

Using this approach, when the server shuts down or restarts, the data in this memory space is of course lost. So, data persistence to disk would be a topic of interest next, but right now, we already see some useful business case with this in-memory based table where data processing speed is more important than data persistence

4. Initial Results

Using the same tuple structure and logic as the current heap plus the memory based buffer manager with 1GB of memory allocated, we observe

some interesting increase in performance comparing to PostgreSQL with default settings. For 20 million row, we observe about 50% increase for insert, 70% increase for update, 60% increase for delete and 30% increase in vacuum. This result is not too bad considering we are still at the early stages and I am sure there are many other ways to make it even faster

5. Next Step

Having some solid results from the initial test, it would make sense for us to also be looking into having the index tuples as in-memory storage only. In addition, free space map and visibility map files that are stored in the PG cluster directory could potentially also be made as in-memory to possibly further increase the DML performance.

Cary is a Senior Software Developer in HighGo Software Canada with 8 years of industrial experience developing innovative software solutions in C/C++ in the field of smart grid & metering prior to joining HighGo. He holds a bachelor degree in Electrical Engineering from University of British Columnbia (UBC) in Vancouver in 2012 and has extensive hands-on experience in technologies such as: Advanced Networking, Network & Data security, Smart Metering Innovations, deployment management with Docker, Software Engineering Lifecycle, scalability, authentication, cryptography, PostgreSQL & non-relational database, web services, firewalls, embedded systems, RTOS, ARM, PKI, Cisco equipment, functional and Architecture Design.