GSoC 2019 Project Proposal

ARTful Buffer Manager

Objective

The main goal of this project is to implement and evaluate an adaptive radix tree as an alternative to the current underlying data structure of PostgreSQL's buffer manager. ART is a space efficient data structure that supports fast lookup operation, provides data locality and preserves order.

Introduction

Application-level memory management is an essential activity in every user-space program, despite the applied allocation strategy or the tasks to be solved. Relational database management systems(RDBMS) are not an exception, on the contrary, notably attentively try to operate accessible resources. One of the major components of disk-oriented RDBMS is a buffer manager, which main goal is to minimize the number of block transfers between the disk and memory. Traditionally, a data structure such as a hashtable is used for this purpose, mainly, due to its ability to efficiently convert sparse keys into dense array indexes and provide fast lookup. Unfortunately, typical hashtable does not preserve order and has poor data locality - properties, that can be useful during data integrity ensuring process and relation or index removal.

PostgreSQL uses Write-Ahead-Logging(WAL) mechanism to guarantee reliable operation that significantly reduces the number of disk writes because only the log file needs to be flushed to disk to guarantee that a transaction is committed. To keep log file small and decrease system recovery time after crush, all changes to data files must be saved. To do that, a special checkpoint process marks all dirty(changed) buffers, performs sorting and flushes them to disk. Sorting part can be eliminated if an underlying structure can preserve order.

Another problem, that can be solved by a data structure with data locality, is deletion or truncation of an existing relation. Currently, the whole hashtable must be scanned in order to remove buffers that belong to the interested object. Taking into account the memory size of modern server machines, this can be a fairly expensive operation. Moreover, typical hashtable does not optimally utilize CPU cache lines and the Translation Lookaside Buffer that leads to cache misses.

With above being said, i would like to introduce a data structure that provides properties, stated in project objective - Digital/Radix Tree. Early explorations of digital trees apparently discarded them as memory pit not worth pursuing further, but a lot of studies have been made in the area since, namely Judy Array[0], Generalized prefix trees[1], KISS-tree[2], ART[3], HOT[4], etc. The digital tree can be used to index arbitrary data, requires no rebalancing operation, stores keys implicitly and its height depends on the length of the keys.

Project Description

The goal of this project is to implement and evaluate digital tree as a base data structure for buffer management in PostgreSQL. The digital tree consists of two types of nodes: inner(intermediate) nodes and leaf(data) nodes. Span is a *S* bit chunk of the key is used as the index into an array of inner node. The most efficient representation of an inner node is an array of 2^S pointers that point to the next layer of inner nodes or data itself. Such representation determines the next child node without additional comparisons, but, unfortunately, can consume a lot of memory in case of sparse input keys. Span size is critical for the performance, as it determines the height of the tree for a given key. K bit keys imply ceil(K/S) levels of inner nodes.

Currently, BufferTag structure in PostgreSQL that is used for disk page identification occupies 20 bytes, so for a span of 1 byte, the height of the tree will be 20, which, in turn, will have large memory footprint. Due to this reason, adaptive radix tree[3] will be used as a basis for development, since the Linux kernel variant[6] is less space efficient (comparison provided in [3]). ART utilizes path compressions and lazy expansion to reduce the average height of the tree. Four inner node types, named according to their maximum capacity, are used to map partial keys to child pointers. This segregation strategy keeps the representation compact while permitting an efficient search.

Digital tree in contrast to the PostgreSQL hash table implementation has increased requirements to memory management, as its structure dynamically adapts to the keys. For this reason, separate shared memory allocator should be designed and implemented. For simplicity, a pre-allocated memory region can be used in a stack-based manner, with an ability to recycle nodes by keeping separate list for each type of node. On the base of that, first version of ART tree can be developed.

Once base ART tree is implemented it can be extended to support concurrent access(buffer loading and pinning) by multiple backend processes, so that they also can perform tree modifications. Several approaches are presented in ART sync paper[5]. The workflow of buffer manager differs from pure index structure in the way that readers can modify tree during an attempt to read the required buffer, instead of inserting the specified key. If the path already exists, they simply traverse to the leaf and try to lock buffer in shared or exclusive mode. Anyway, despite this difference, mechanisms that are presented in [5] can be adapted and applied.

At the final stage, the digital tree can be benchmarked with different workload profiles, using pgbench and TPC-* datasets as well as checkpoint/drop operations.

Expected Outcomes

A result of this project can produce specific numerical evaluations of the benefits, that can be achieved by substitution of the current underlying buffer manager data structure with ART. Among them are: faster checkpoint and drop/truncate relation operations, possibly higher buffer lookup speed. Most importantly, the digital tree can open new perspective improvement directions in the area such as “neighbour buffer prefetching”, “multiple buffers write combining” and “fine-grained shrink/extend relation operation”.

I am planning to continue work in this area afterward and foresee another extensibility direction of PostgreSQL(among many others) - common interface for different buffer manager realizations.

Timeline

Time Period	From	To
Community Bonding period	07.05	26.05

- introduce the project to the community, get in touch with a mentor;
- setup repository, read the documentation (Buffer manager, WAL, Checkpointer, etc);
- briefly describe related to the project research papers;
- explore source code of existing buffer manager, especially shared memory initialization, buffer allocation/eviction;
- explore open source implementations of the digital tree;
- design memory allocator that will satisfy digital tree needs in different node types, its recycling strategy.

Work period objectives	27.05	27.06

- 75% goal: implement memory allocator and a raw version of the digital tree;
- 100% goal: refine basic digital tree, cover code with sanity checks;
- 125% goal: add validation functionality, perform synthetic lookup/inorder traversal tests and evaluate results.

Phase 1 Evaluation	24.06	28.06
Work period objectives	29.06	25.07

- 75% goal: add concurrency support and perform regression tests;
- 100% goal: evaluate checkpoint/drop operations improvements, play with tree configurations, caching strategies(fast jumps to Relation's tree of blocks, by skipping first 12 bytes of BufferTag), observe subsequent performance changes;
- 125% goal: perform medium-lasting workload tests, stabilize allocator and improve recycling.

Phase 2 Evaluation	22.07	26.07
Work period objectives	27.07	25.08

- fix bugs, run long-lasting workload test, make sure validation functionality works correctly;
- refine and refactor code, apply code style, cover with comments;
- prepare a final report by describing obtained results, present it to the community.

Final Evaluation	19.08	26.08

References

[0] http://judy.sourceforge.net/

[1] https://subs.emis.de/LNI/Proceedings/Proceedings180/227.pdf

[2] http://www.qucosa.de/fileadmin/data/qucosa/documents/8808/damon-kissIndex-camera-ready.pdf

[3] https://db.in.tum.de/~leis/papers/ART.pdf

[4] https://dbis-informatik.uibk.ac.at/sites/default/files/2018-06/hot-height-optimized.pdf

[5] https://db.in.tum.de/~leis/papers/artsync.pdf

[6] https://lwn.net/Articles/175432/