

CS 6422 BuzzDB Project Proposal - Advanced Slotted Page Structure - Report

https://github.com/abhijeetsaraha/6422_extra_credit.git

Introduction

The project introduces tuple handling within a page in order to better manage space within a slotted page. The goal is to reduce extra I/O operations (loading page from disk, database extension, insertion time) by reducing internal fragmentation/compaction of memory in a slotted page.

Related Works

Optimizing page storage has been the focus of research for many. Especially with varied storage technologies like SSDs, hardware specific storage optimization techniques like the two level minipage scheme [1] are getting popular. For this particular project, I went with a general approach of compaction of space within a page that is not hardware/architecture specific.

Method


There are many variations of buzzdb available in the class github [2]. For measuring the performance and usage of advanced slotted page structure over the naive approach, the version 16 of buzzdb is chosen. There are minimal other functionalities in this version of buzzdb, hence measuring the correctness/performance of advanced slotted page structure is easier as this avoids other hidden costs.

The base version of buzzdb deletes only the first element using deleteTuple method. In order to check for better generality, I pass a random index to the deleteTuple method. There are two options to generate random indices - uniform distribution and rand(). While both work with the approach, for correctness rand() is a better option as it generates the same pseudo random numbers over multiple runs. This in turn helps in verifying correctness of each implementation by checking the results of buzzdb queries.

Implementation

Compaction on Delete

In this approach, on delete we call a method to rearrange the tuples within the page. Essentially, it involves looping through slot indices starting from the deletion index to the last tuple in the page and shifting them one step to the previous index.



This provides maximum improvement in space utilization within a page. Compacting space removes internal fragmentation. It removes the scenario where the page has the memory space to fit another tuple, but the addTuple operation fails due to the space being fragmented. It is harder to test on the buzzdb implementation as all tuples are of the same size, but variable tuple sizes are more common in databases.

Bookkeeping deleted Indices

While compaction provides maximum space utilization, it is not the best in terms of performance, as it involves looping through the slots on every delete. This takes a lot of processing power, especially in a workload that consists of a lot of deletes.

An alternate approach is to keep track/record of all the deleted indices in a vector and the total number of tuples currently in the page. When performing tuple addition, we check if the vector of free indices has any valid entries. If it does, we check if the tuple fits into the free slot and perform addition on that index. If not, we simply set the incoming tuple to be added on the end of the slot array which is the current number of tuples. In both cases, the index is determined in time complexity of $O(1)$ or amortized $O(1)$. This does not guarantee not internal fragmentation, but if the tuple sizes are similar the performance gain is very high (especially for architectures with large page sizes) and the internal fragmentation is low. The tradeoff in this scenario is good.

References

1. Stephen M. Ash and King-Ip Lin. 2014. Optimizing database index performance for solid state drives. In Proceedings of the 18th International Database Engineering & Applications Symposium (IDEAS '14). Association for Computing Machinery, New York, NY, USA, 237–246. <https://doi.org/10.1145/2628194.2628255>.
2. Joy Arulraj. BuzzDB.
github<https://github.com/jarulraj/buzzdb/blob/main/16-buzzdb.cpp>.