Towards Adaptive Storage Views in Virtual Memory

Felix Schuhknecht
Johannes Gutenberg University Mainz
schuhknecht@uni-mainz.de

Justus Henneberg
Johannes Gutenberg University Mainz
henneberg@uni-mainz.de

Traditionally, the storage engine of a DBMS exposes its entire database to the surrounding system components. To limit query processing to parts of interest, additional index structures must be created on top. While these index structures certainly speed up query processing in a large variety of situations, unfortunately, they add a level of physical indirection to query execution. This indirection must be resolved at runtime, which causes unpleasant overhead. Therefore, in this abstract, we present a way to naturally fuse indexing and storage: The storage engine maintains multiple coarse-granular *views* for all tables, each covering a subset of the database with a certain property of interest. By realizing views through the virtual memory management subsystem of the OS, they cause *zero overhead* at access time, and do not require the table to be physically replicated. Our prototype creates views adaptively to best fit the overall query workload, and routes each query automatically to the best fitting views for scanning during query processing.

Adaptive Virtual Views. Figure 1 visualizes the concept for one full view (indexing all data) and two partial views indexing only portions of the data with certain properties (indexing only blue shapes, or only circles, respectively). All views map to (portions of) the very same physical memory area, in which the entire dataset is materialized. If we wanted to find only purple circles, we would scan the partial view indexing circles instead of the full view, as it matches our request best.

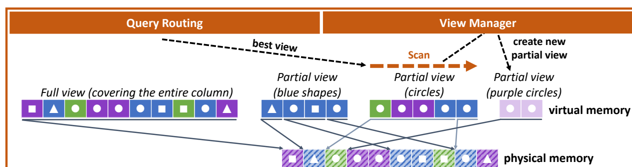

Figure 1: Visualization of the architecture of our storage layer at an example: In addition to the full virtual view, each column provides two partial views indexing only subsets of the data.

As a side-product of query answering, we build a new partial view that now indexes only purple circles. On Linux, views can be created in virtual memory via `mmap` in combination with *main-memory files*: Each memory page containing a purple circle will be referenced by a newly created contiguous virtual memory region. Since `mmap` calls are expensive, we perform all calls from a concurrent mapping thread, and collect adjacent relevant pages into a single `mmap` invocation.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2023. 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

The newly created view can then be utilized for future query processing. Note that newly created partial views are only retained by the view manager if they improve the current indexing situation. For example, if the new partial view v_n covers a *subset* of an existing partial view v_e, but references a similar amount of physical memory pages, then v_e can be used to answer more queries than v_n with comparable scanning effort. Therefore, the view manager will discard v_n to keep the number of indexes to maintain low.

Query Routing. To answer an incoming query using the existing views, we support two modes of operation: In *single-view* mode, we use exactly one view to answer the query, where this view must fully cover the predicates of the query. If there are multiple views available that fulfill this property, we pick the view that indexes the smallest amount of physical pages to minimize the scanning effort. In *multi-view* mode, we potentially use multiple views to answer a single query, provided that these multiple views fully cover the requested range in conjunction. As physical pages might be shared between multiple partial views, we additionally keep track of processed physical pages to avoid scanning a page twice, as this would lead to incorrect results.

Handling Updates. If updates happen through the full views, these updates must be reflected by all existing partial views to ensure correctness. This involves potentially adding and removing pages from each partial view that covers a value range affected by an update using repetitive `mmap`-calls. As this process can become costly when being performed for each update individually, we support updating partial views with respect to an adjustable batch of updates. If too many pages of a partial view would be changed by a batch of updates, the view is rebuild from scratch instead.

Performance. Figure 2 shows how our adaptive partial views improve the individual query response times over fully scanning all data under a sequence of 250 range queries with varying selectivity. We test a clustered data distribution that follows a sine curve. We can see that already early on in the sequence, the adaptively created partial views are used by the query processing to significantly speed up scans.

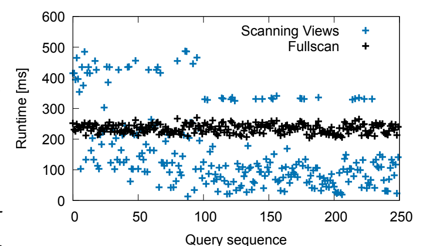

Figure 2: Performance

Extended Version. For a detailed presentation, discussion, and evaluation of virtual storage views, please refer to the extended version of the paper [2]. All code of this project is freely available under [1].

REFERENCES

[1] 2022. https://gitlab.rlp.net/fschuhkn/adaptive-virtual-storage-views
[2] Felix Martin Schuhknecht and Justus Henneberg. 2022. Towards Adaptive Storage Views in Virtual Memory. *CoRR* abs/2209.01635 (2022). arXiv:2209.01635