Towards Adaptive Storage Views in Virtual Memory

Felix Schuhknecht and Justus Henneberg

Johannes Gutenberg University Mainz, Germany

infosys.informatik.uni-mainz.de

CIDR 2023
A Traditional Storage Engine

column
A Traditional Storage Engine

column

68
42
10
13
89
99
17
12
34
38
53
49
51
59
8
10

some data
A Traditional Storage Engine (in Main-Memory)

physical column

some data stored on physical memory pages
A Traditional Storage Engine (in Main-Memory)

physical column

virtual memory mapping on the whole column
A Traditional Storage Engine (in Main-Memory)

physical column

scan

virtual memory mapping on the whole column
A Traditional Storage Engine (in Main-Memory)

physical column

full view covering \([-\infty, \infty]\)
A Traditional Storage Engine (in Main-Memory)

Physical column

Scan the whole column

Full view covering $[-\infty, \infty]$
A Traditional Storage Engine (in Main-Memory)

full view covering \([-\infty, \infty]\)
Why not use the index that is already in place?
The Index of the OS

OS page table

full view covering [-∞, ∞]
Partial Virtual Views

physical column

full view covering $[-\infty, \infty]$
Partial Virtual Views

- Partial view covering [8, 17]
- Full view covering [-∞, ∞]
- Physical column
- `mmap()` calls

Diagram shows a partial view covering [8, 17] and a full view covering [-∞, ∞].
Partial Virtual Views

![Diagram showing a partial virtual view and a full view.](image)

- **Physical column:**
  - First column: 68, 42, 10, 13, 89, 99, 17, 12, 34, 38, 53, 49, 51, 59, 8, 10

- **Partial view covering [8, 17]:**
  - Sub-column: 10, 13, 17, 12, 8, 10

- **Full view covering [-∞, ∞]:**
  - Sub-column: 49, 51, 59, 10, 8, 10

- **mmap()** functions are used to map different views onto the physical column.
Partial Virtual Views

A partial view covering [8, 17]

Full view covering [-∞, ∞]
How to Create Partial Views? Adaptively!

Q1: [40, 70]

1. Find best existing view(s) to answer the query
How to Create Partial Views? Adaptively!

Q1: [40,70]

2. scan and filter for [40,70]

[−∞, ∞]
How to Create Partial Views? Adaptively!

Q1: [40,70]

3. Create new partial view covering [40,70] as side-product
Q1: [40,70]

4. Does the new view improve our situation? Yes → keep it!
How to Create Partial Views? Adaptively!

Q2: [50,55]

Next query can utilize the new partial view

[40,70]

[{-∞,∞}]
Virtual Views vs Traditional Counterparts

<table>
<thead>
<tr>
<th>Runtime [ms]</th>
<th>Index-Selectivity [percentage of qualifying pages]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.65</td>
<td>0.65</td>
</tr>
<tr>
<td>1.27</td>
<td>0.65</td>
</tr>
<tr>
<td>2.53</td>
<td>0.65</td>
</tr>
<tr>
<td>4.97</td>
<td>0.65</td>
</tr>
<tr>
<td>9.71</td>
<td>0.65</td>
</tr>
<tr>
<td>18.48</td>
<td>0.65</td>
</tr>
<tr>
<td>33.55</td>
<td>0.65</td>
</tr>
</tbody>
</table>

- Explicit (Zone Map)
- Explicit (Bitmap)
- Explicit (Vector of Page-IDs)
- Virtual View
Figure 1 visualizes the concept for one region. Since the very same physical memory area, in which the entire dataset is available that fulfills this property, we pick the view that indexes only purple circles, we would scan the partial view indexing circles instead of the full view, as it matches our request best.

As a side-product of query answering, we build a new partial view that now indexes only purple circles. On Linux, views can be referenced by a newly created contiguous virtual memory region. Since memory can be created in virtual memory via `mmap` mode, we potentially use multiple views to adaptively create partial views that cover a value range a fragment.

If too many pages of a partial view would be changed by a batch of updates, the view is rebuild from scratch instead. If too many pages of a partial view would be changed by a batch of updates, the view is rebuild from scratch instead. Instead, we parse the mapping page page-wise in a binary search. For each partial view, we then have a mapping page which can be used to quickly obtain the current mapping between virtual pages and physical pages. As parsing this file is costly if a suitable file already contains all qualifying pages. This resembles an artificial limit to the number of qualifying pages that we can represent.

In order to update our partial views as described in Section 2.4, we need to obtain the current mapping between virtual pages and physical pages, before the actual values are materialized. During the update process. After the batch of updates has been applied, we materialize the parsed mappings page-wise in a binary search. Variations of the page, before the actual values are materialized. During the update process.
Towards Adaptive Storage Views in Virtual Memory

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

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

ABSTRACT

Traditionally, DBMSs separate their storage layer from their in-memory layer, as illustrated by the three-layer stack in Figure 1. While this clearly separates concerns, it also adds a level of indirection to maintaining the database. This includes providing low-level access to records with the property can be found. Consequently, it provides a high-level interface of the form getRecordIterator(), which maps to subsets of the database having certain properties of interest. This creation happens fully adaptively as a side-product of the indexing layer sitting on top of the storage layer. It maps properties to virtual views, which are segmented into pages, spans the entire physical main memory, allowing us to create partially overlapping views. On the one hand, such a separation of concerns yields a clean design. On the other hand, it prevents efficient handling of conditions such as a specification of the semantics of the records, i.e., it cannot be asked to provide two partial views indexing only the subsets of the data. Towards Adaptive Storage Views in Virtual Memory

1.1 Virtual Views

Virtual views is the name given to an in-memory database layer, which can be used to simplify the data structures used by the database. The In-Memory Database (IMDB) API is an example of such an interface. However, this leads to a lot of undetected and undetected indirections, such as the need to maintain a virtual view to access the data. Towards Adaptive Storage Views in Virtual Memory

Optimal Scan Performance:

• Technical details and optimizations
• Single-view vs multi-view query answering
• Efficiently handling updates
• Evaluation under various data and query distributions

Code: https://gitlab.rlp.net/fschuhkn/adaptive-virtual-storage-views