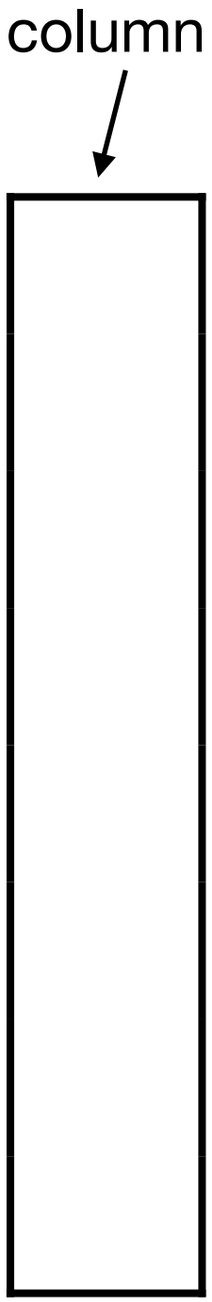
Towards Adaptive Storage Views in Virtual Memory

Felix Schuhknecht and Justus Henneberg

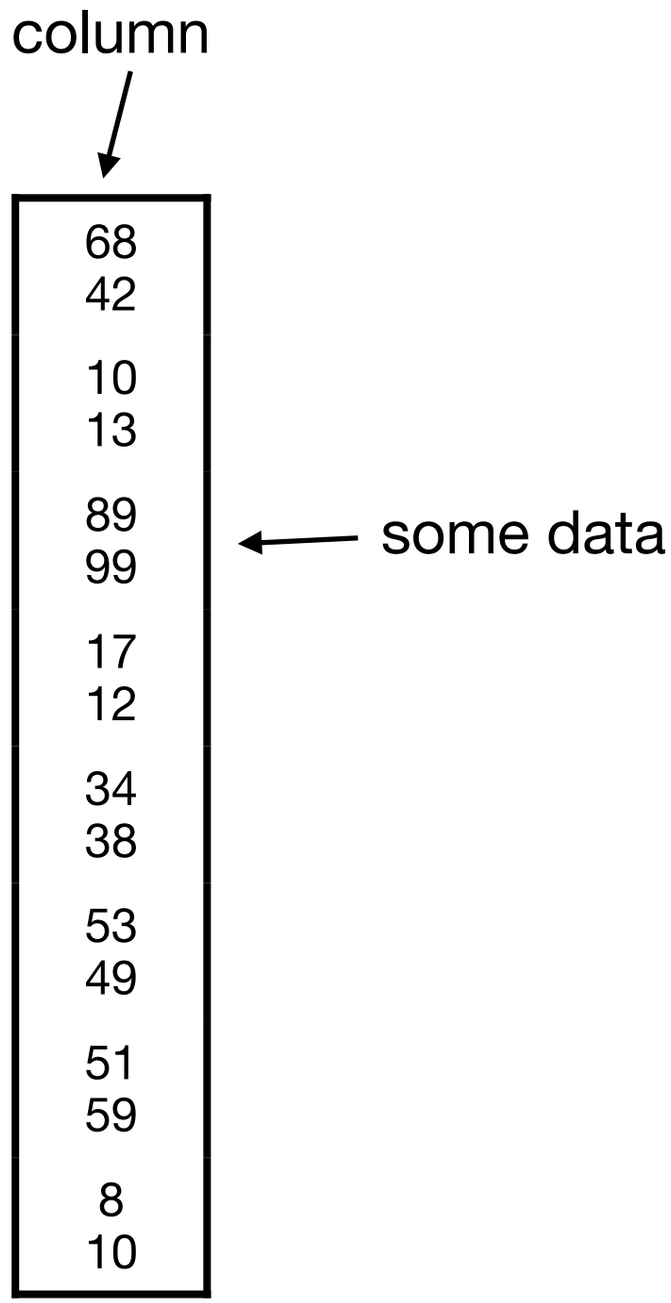
Johannes Gutenberg University Mainz, Germany
infosys.informatik.uni-mainz.de

CIDR 2023

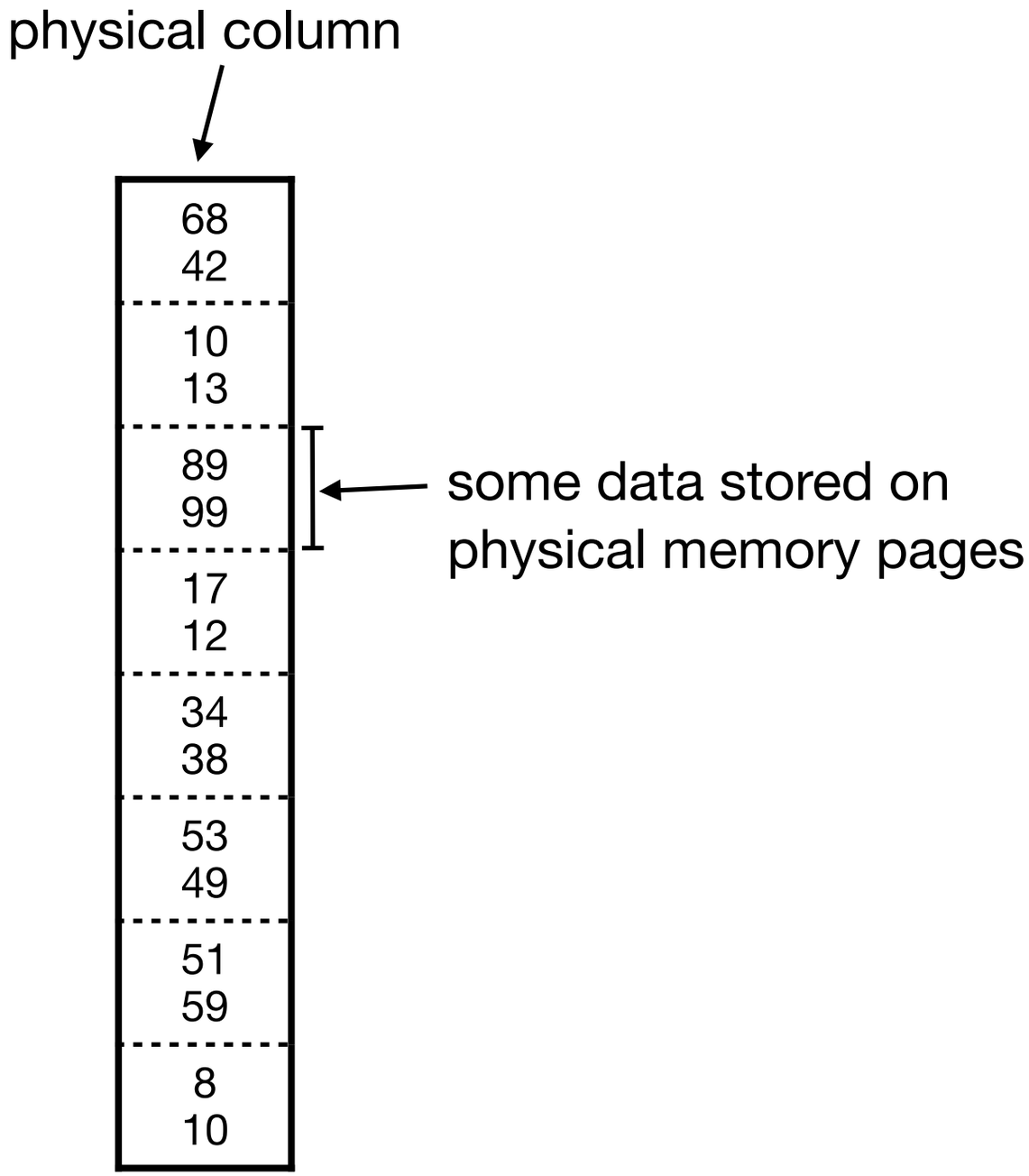
A Traditional Storage Engine



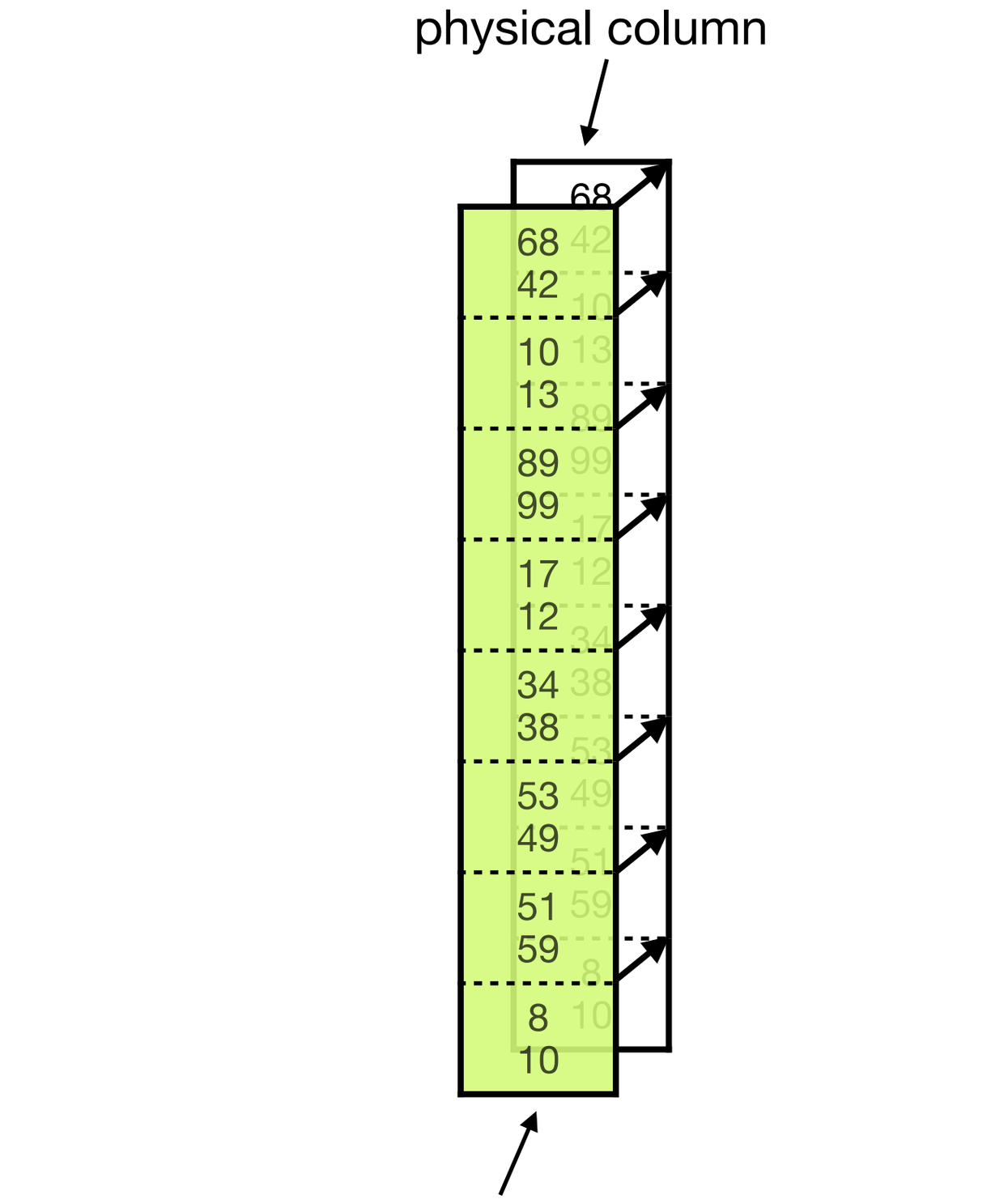
A Traditional Storage Engine



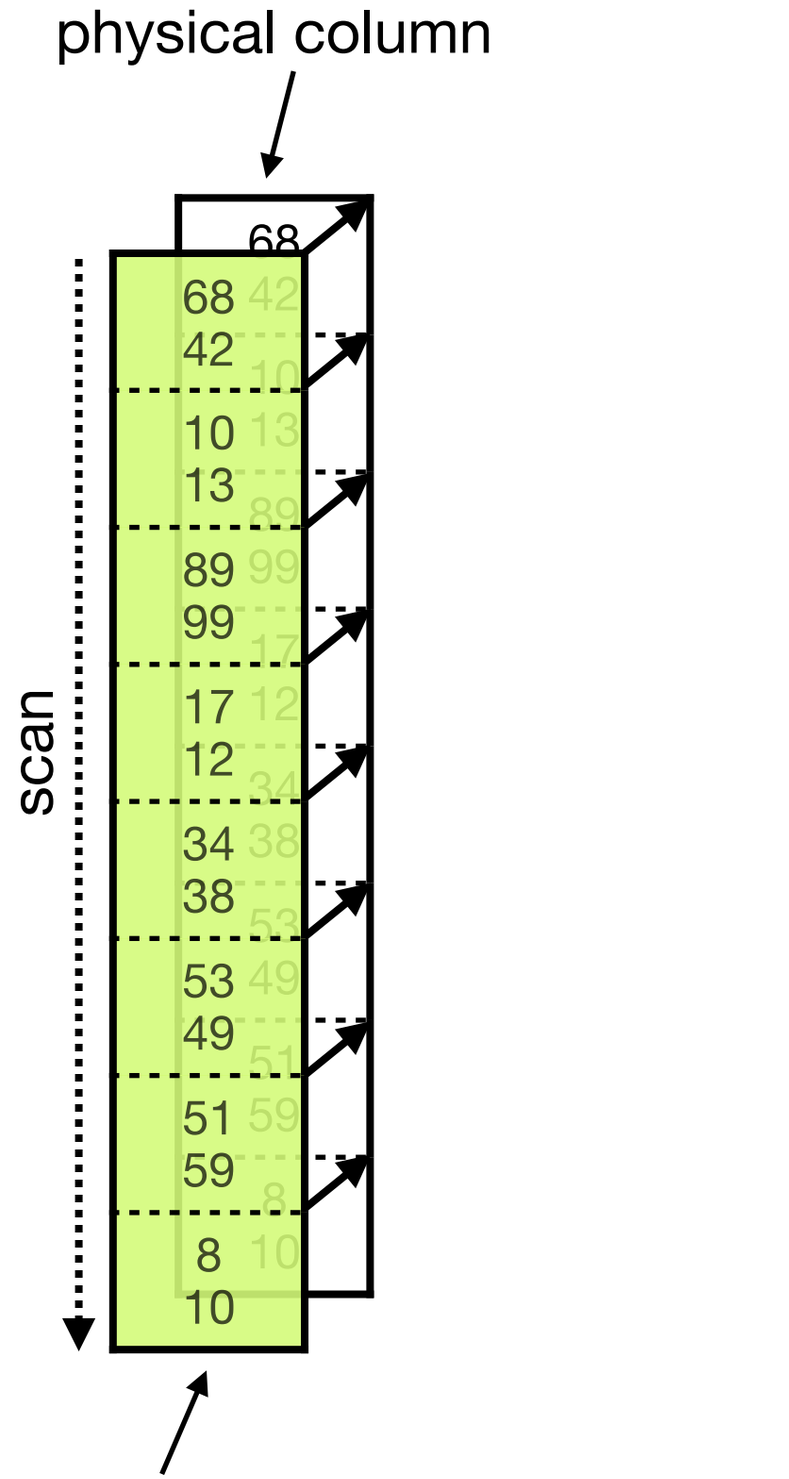
A Traditional Storage Engine (in Main-Memory)



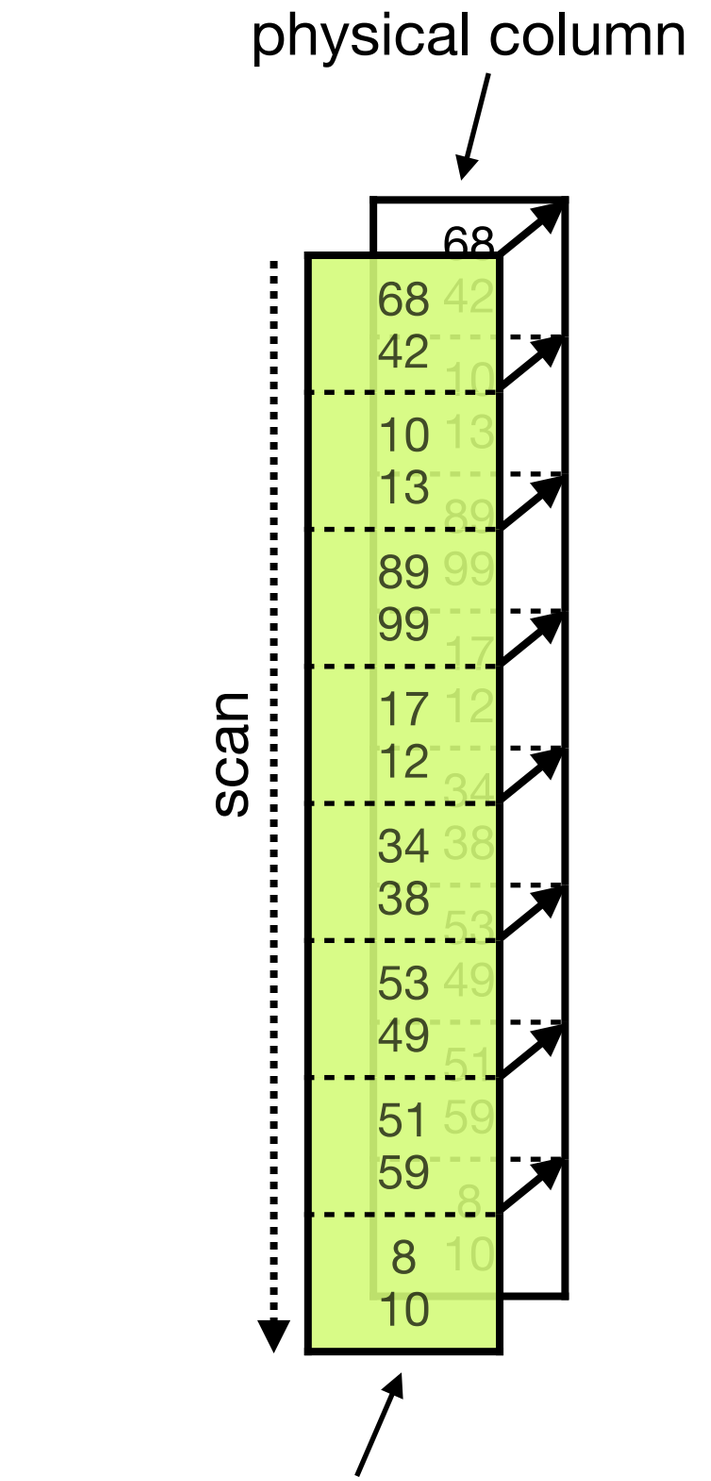
A Traditional Storage Engine (in Main-Memory)



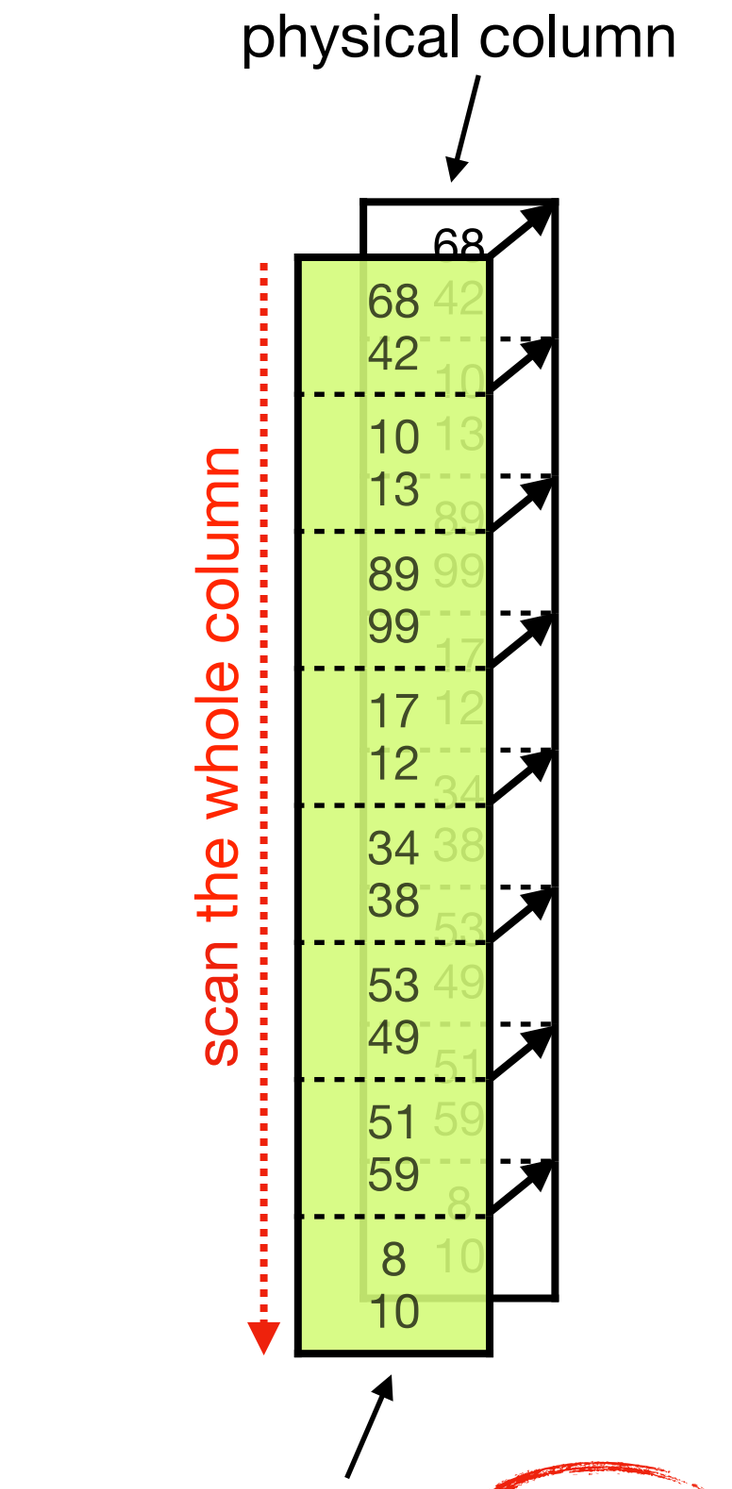
A Traditional Storage Engine (in Main-Memory)



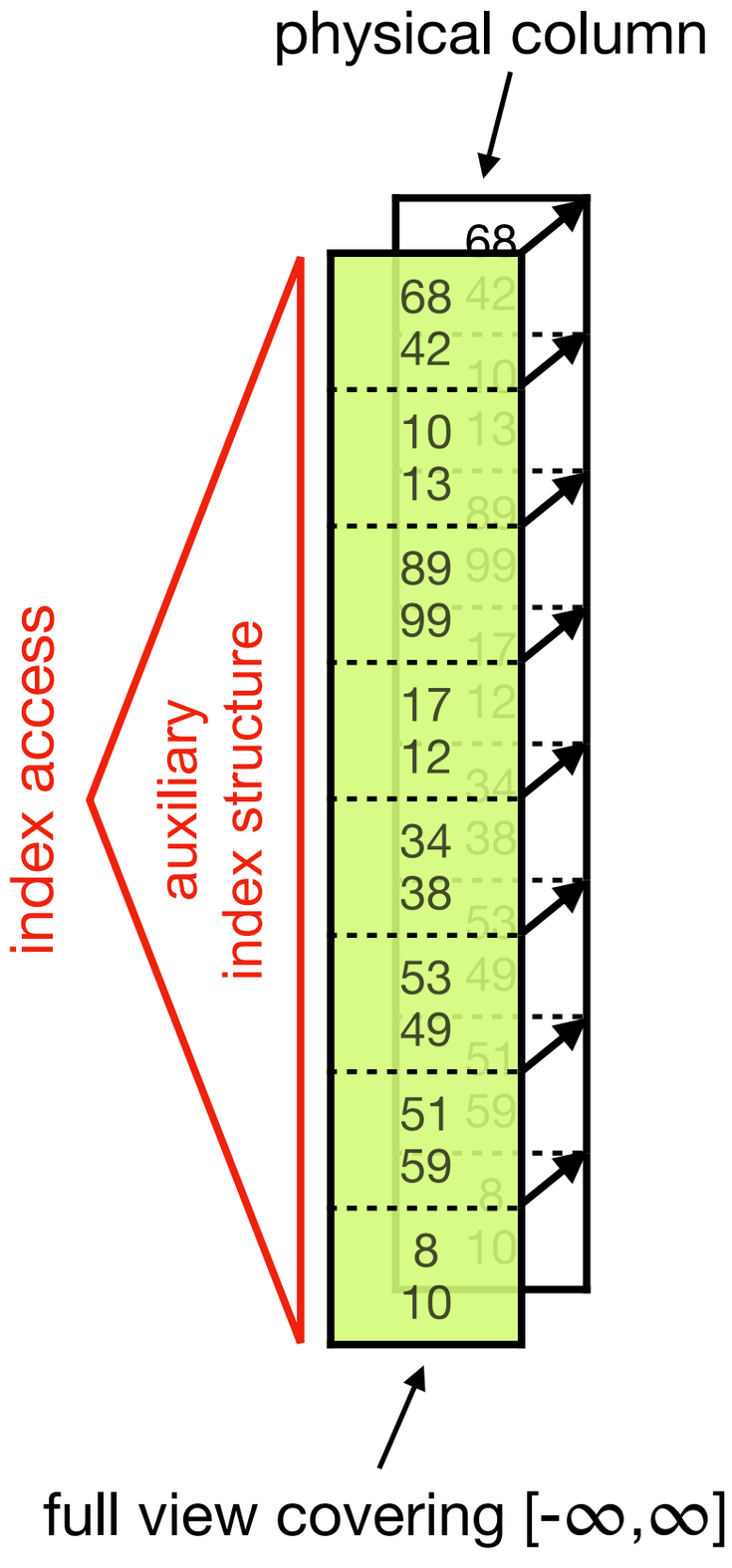
A Traditional Storage Engine (in Main-Memory)



A Traditional Storage Engine (in Main-Memory)

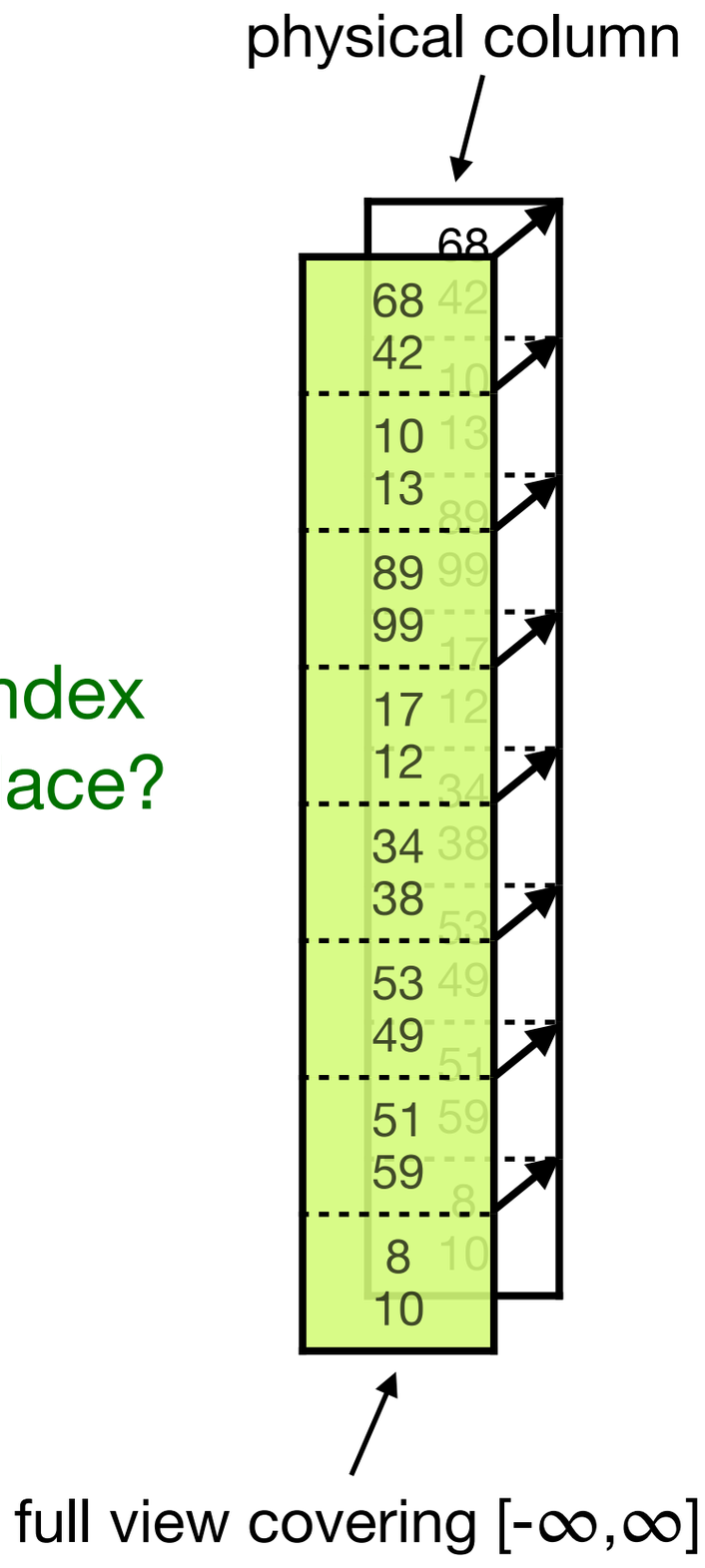


A Traditional Storage Engine (in Main-Memory)

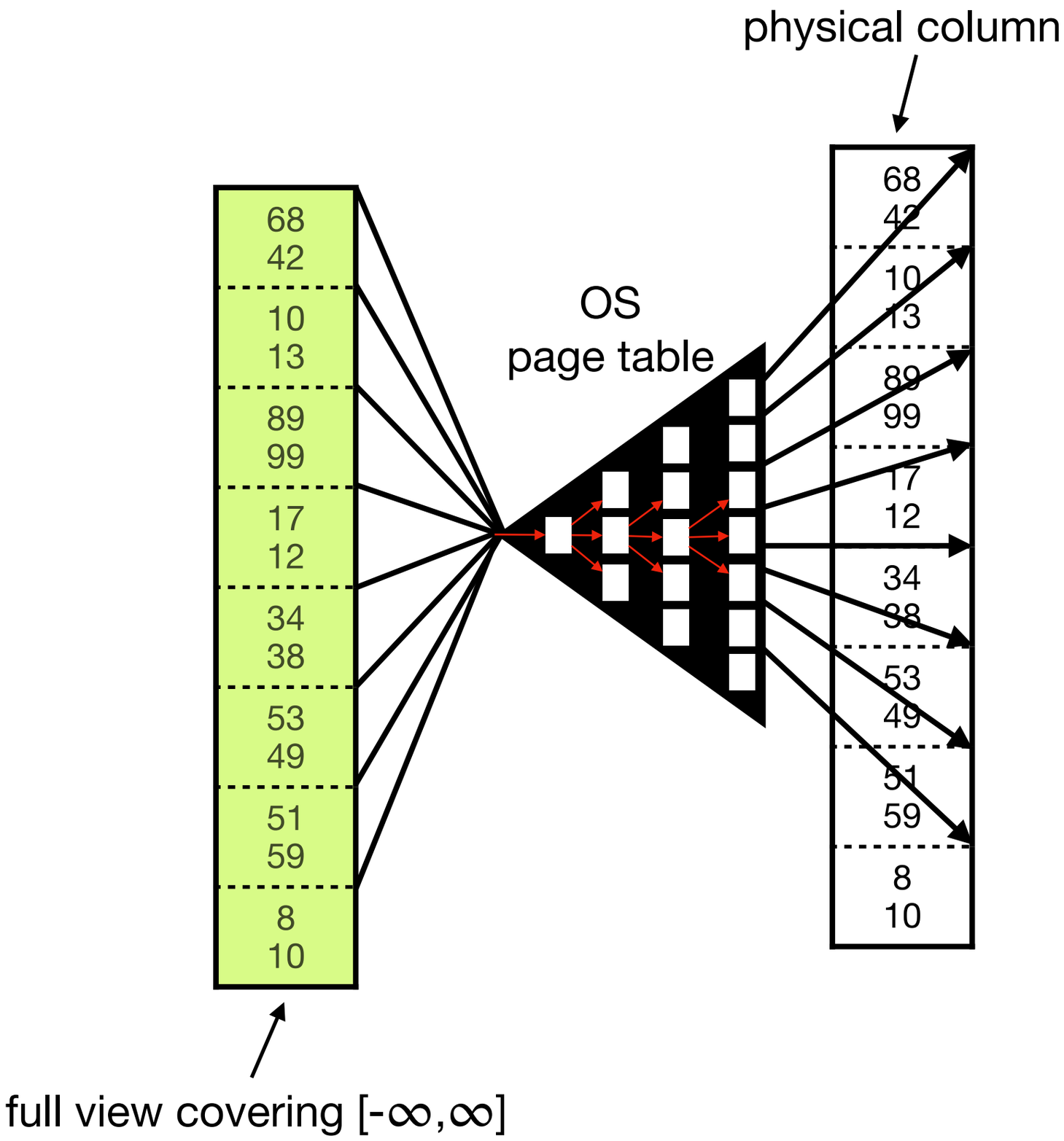


Alternatives

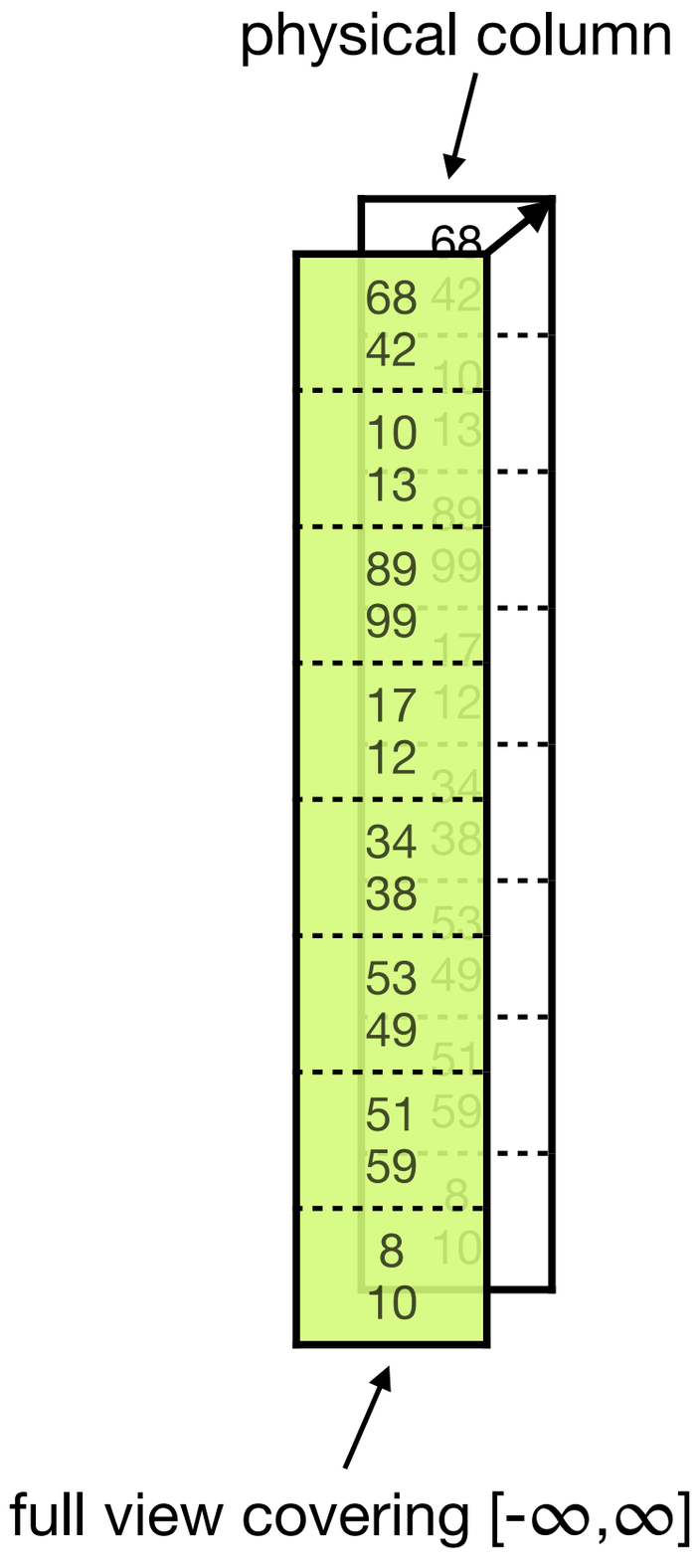
Why not use the index that is already in place?



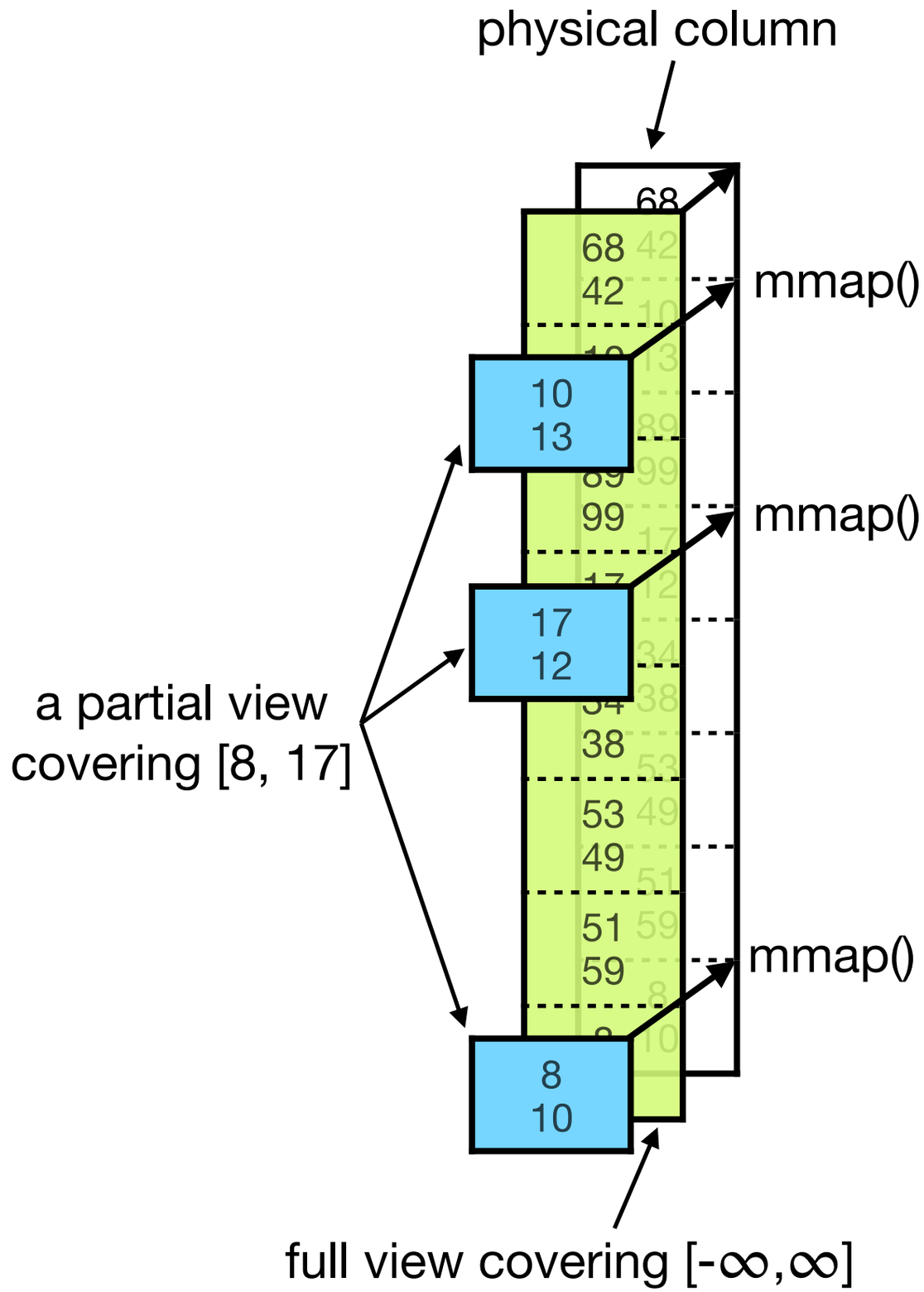
The Index of the OS



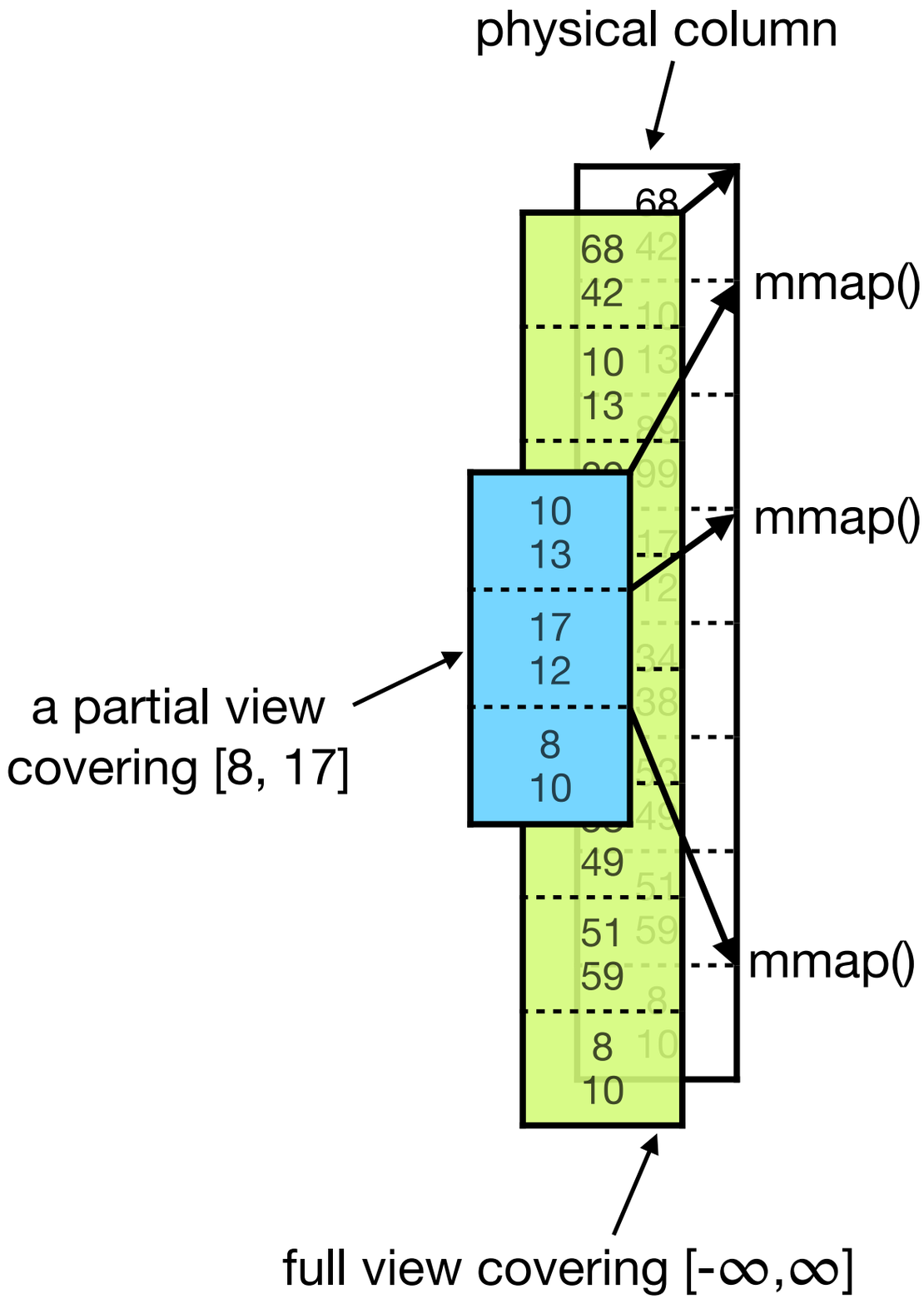
Partial Virtual Views



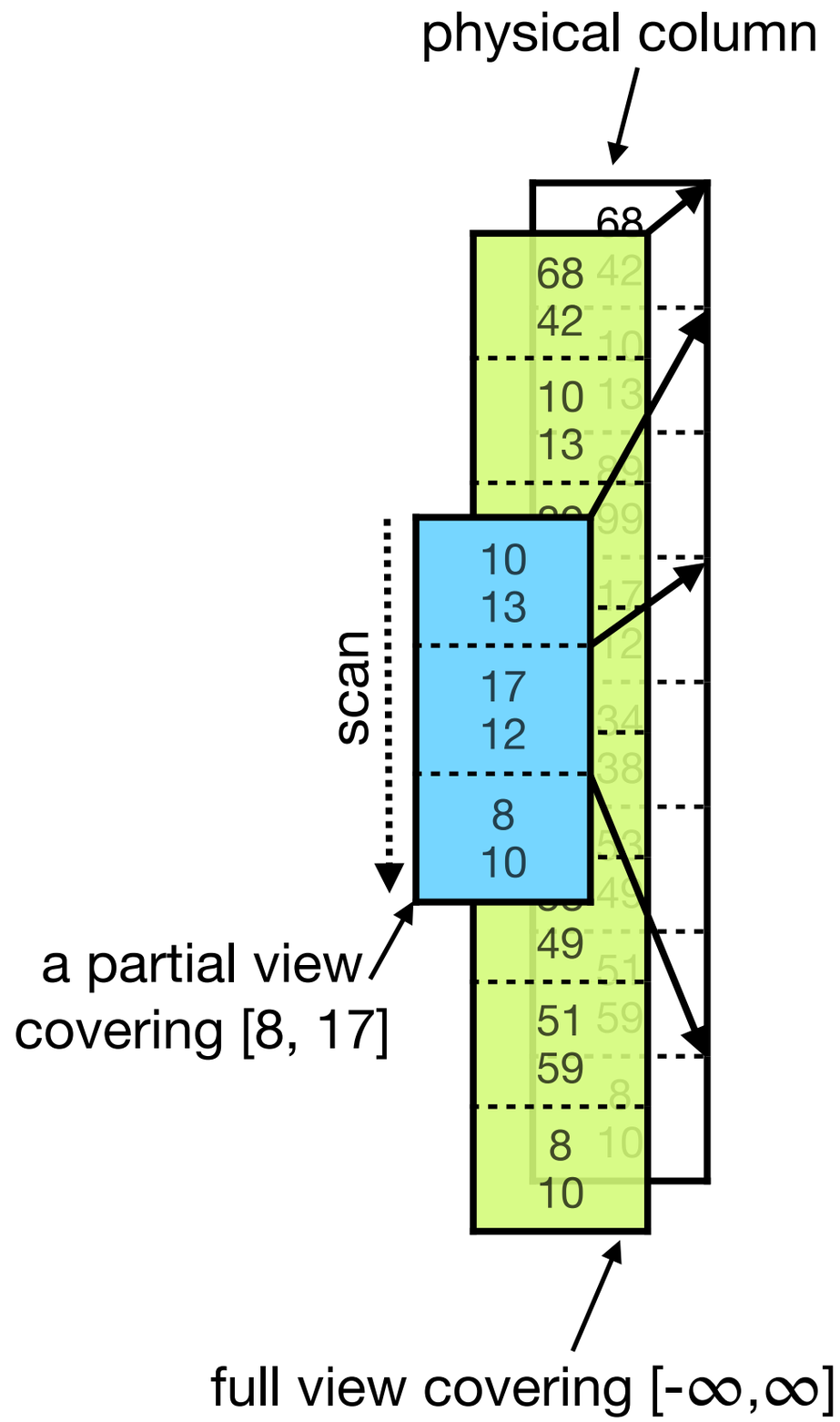
Partial Virtual Views



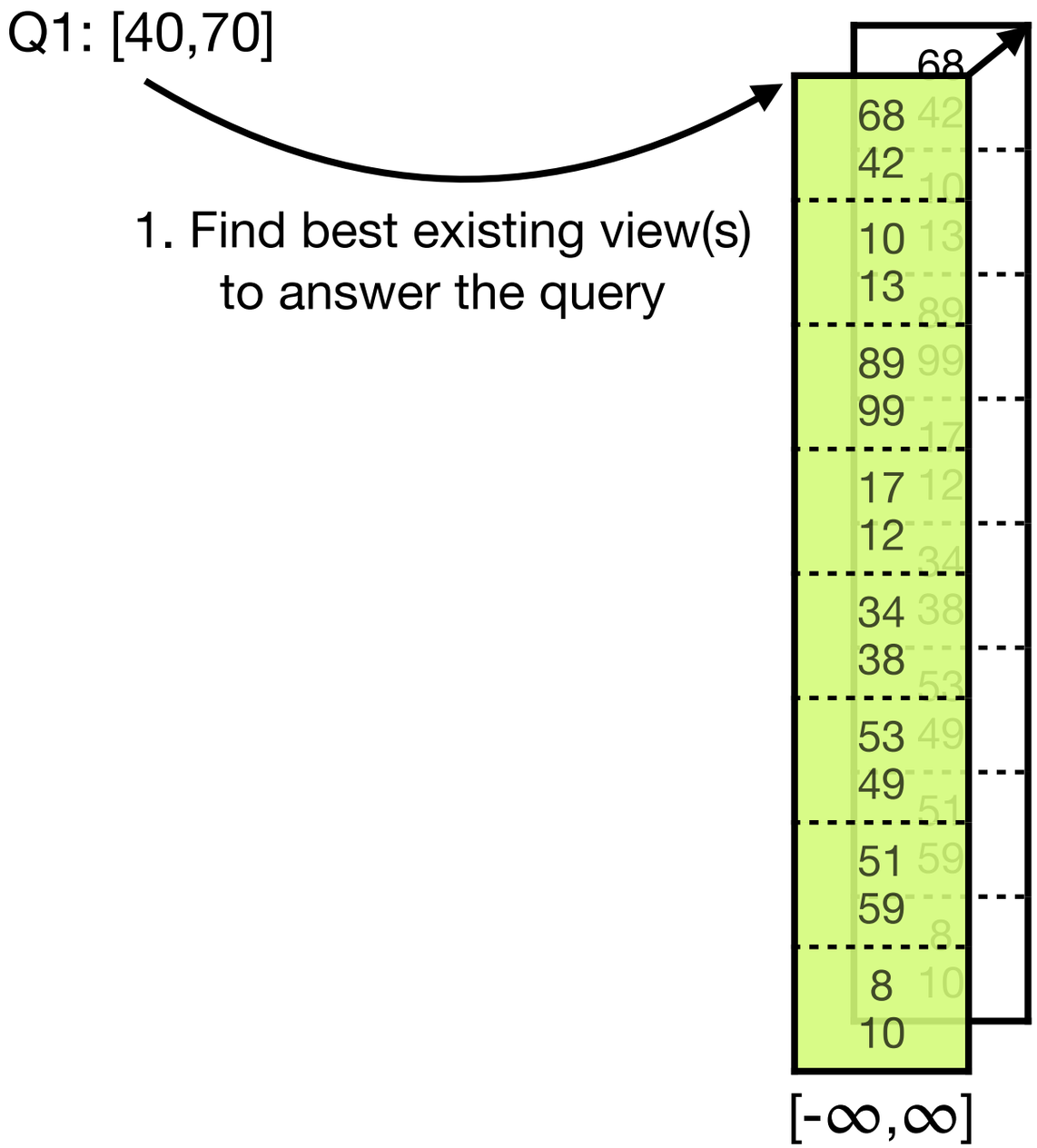
Partial Virtual Views



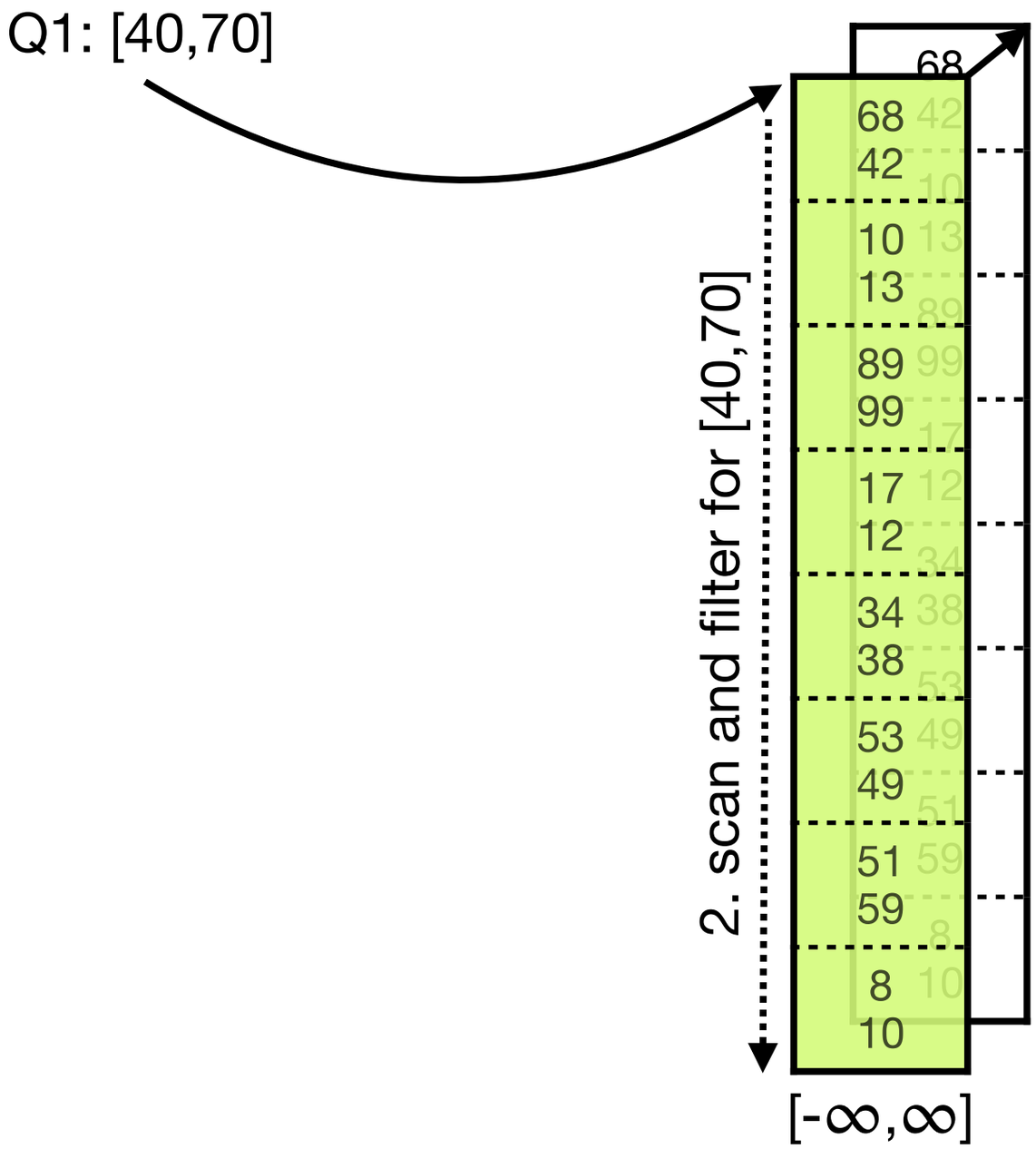
Partial Virtual Views



How to Create Partial Views? Adaptively!



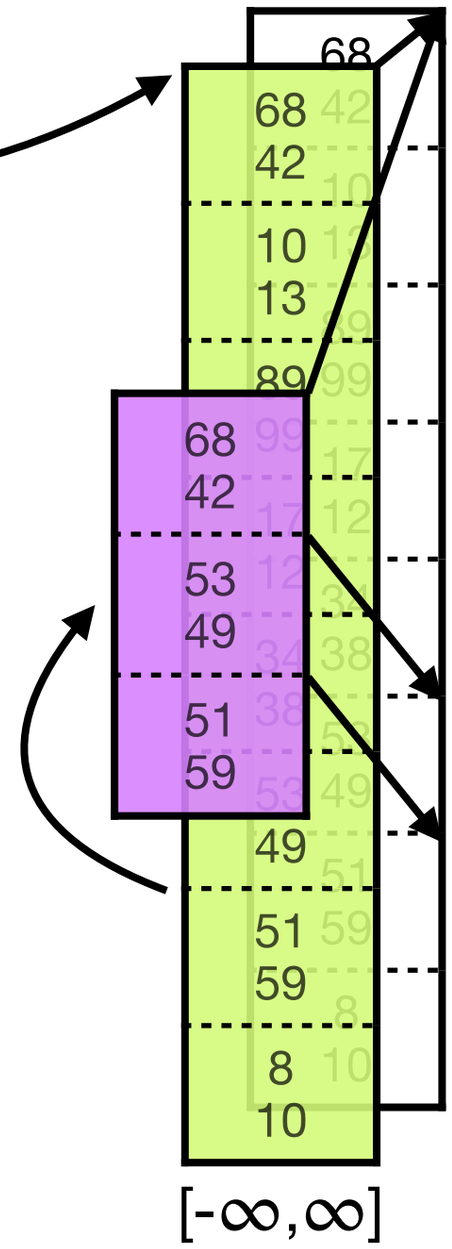
How to Create Partial Views? Adaptively!



How to Create Partial Views? Adaptively!

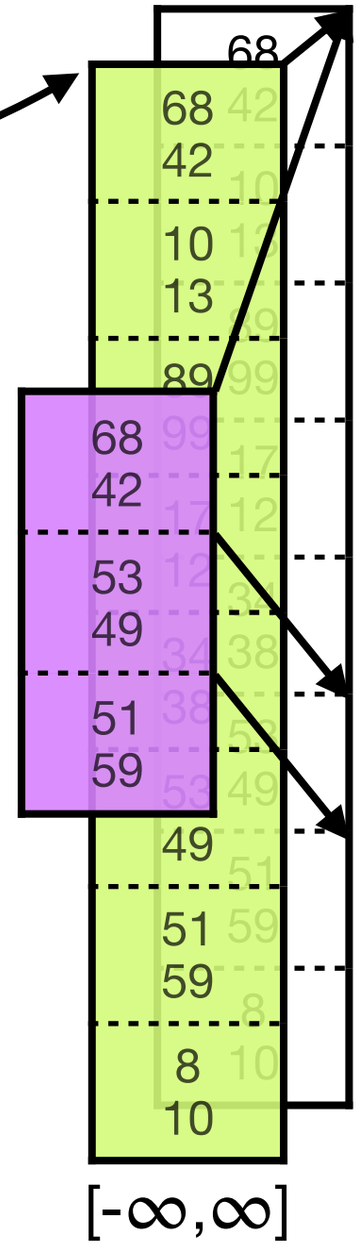
Q1: [40,70]

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



How to Create Partial Views? Adaptively!

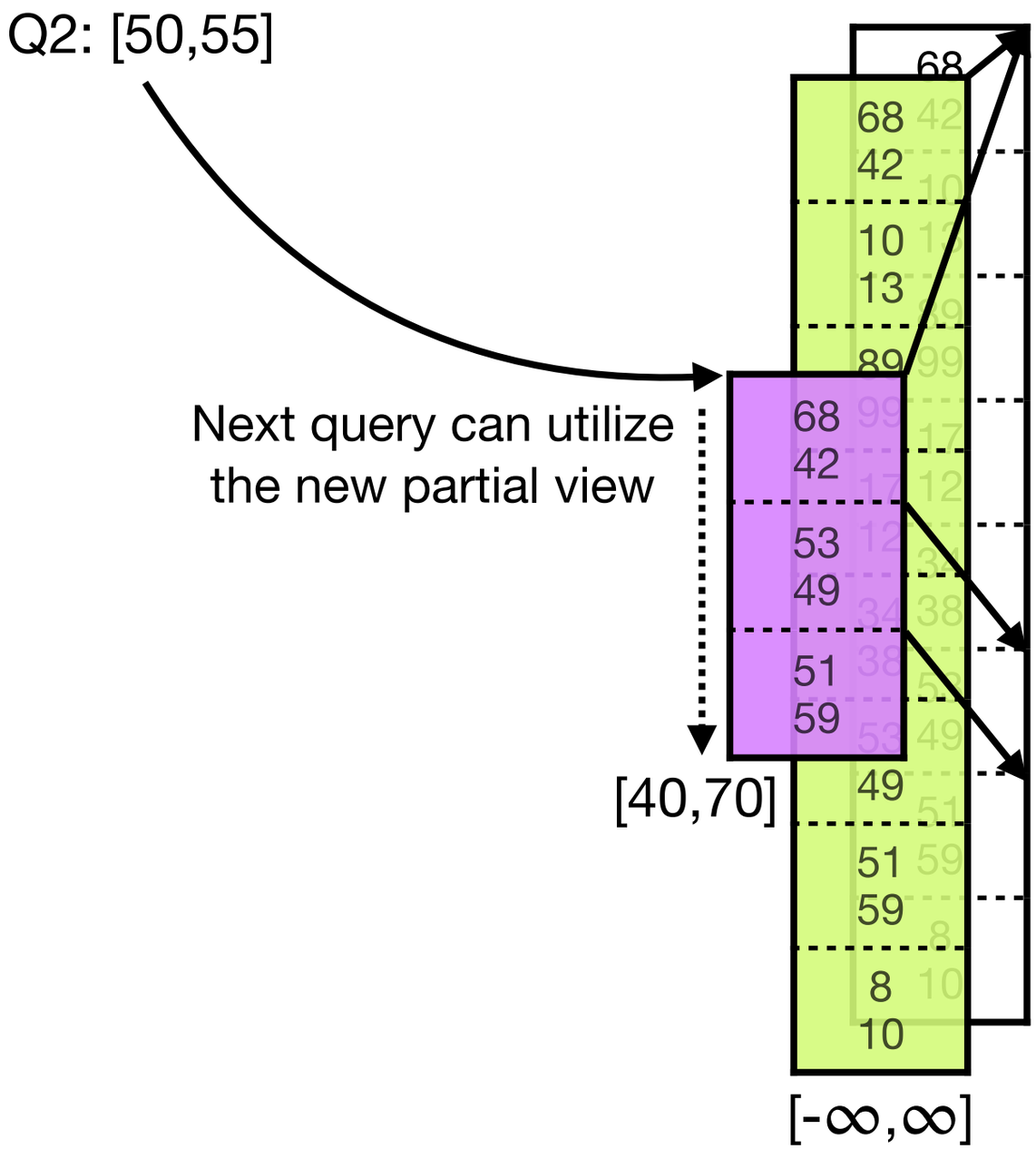
Q1: [40,70]



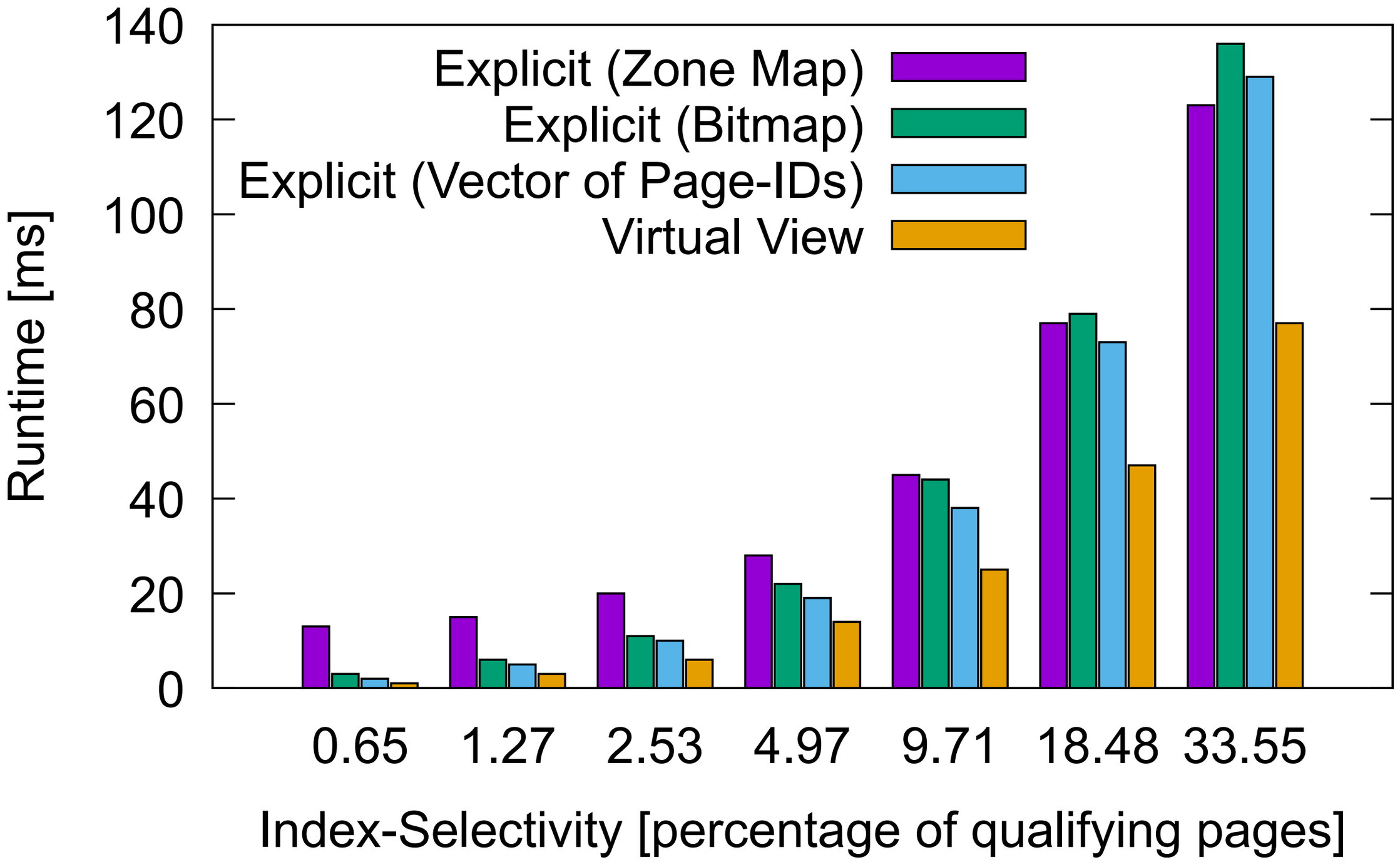
4. Does the new view improve our situation?

Yes → keep it!

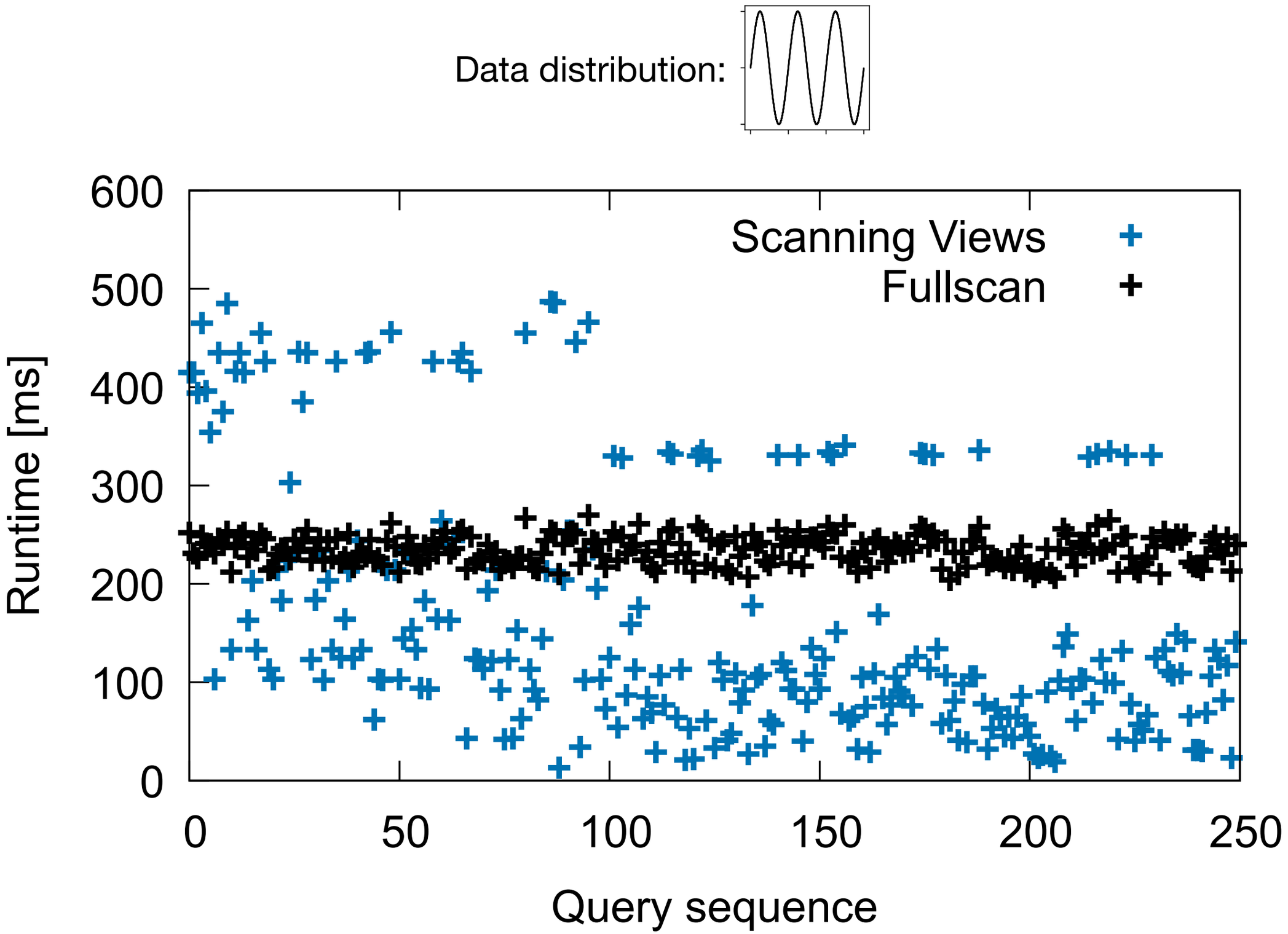
How to Create Partial Views? Adaptively!



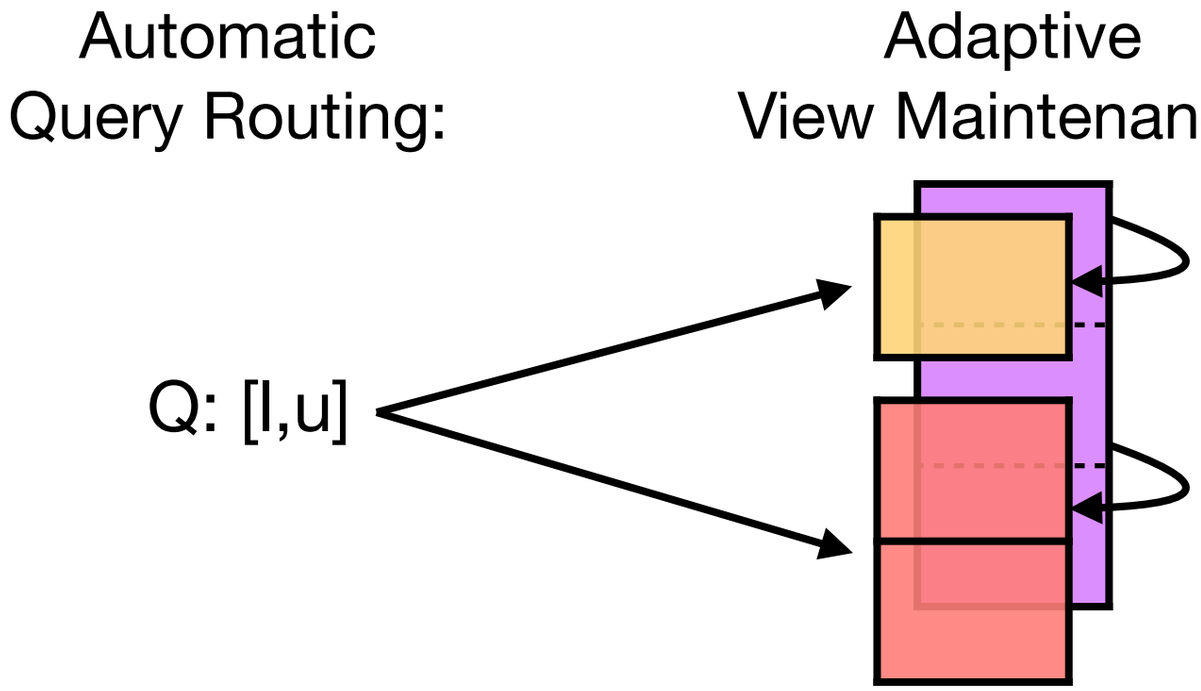
Virtual Views vs Traditional Counterparts



Query Response Time using Adaptive Views



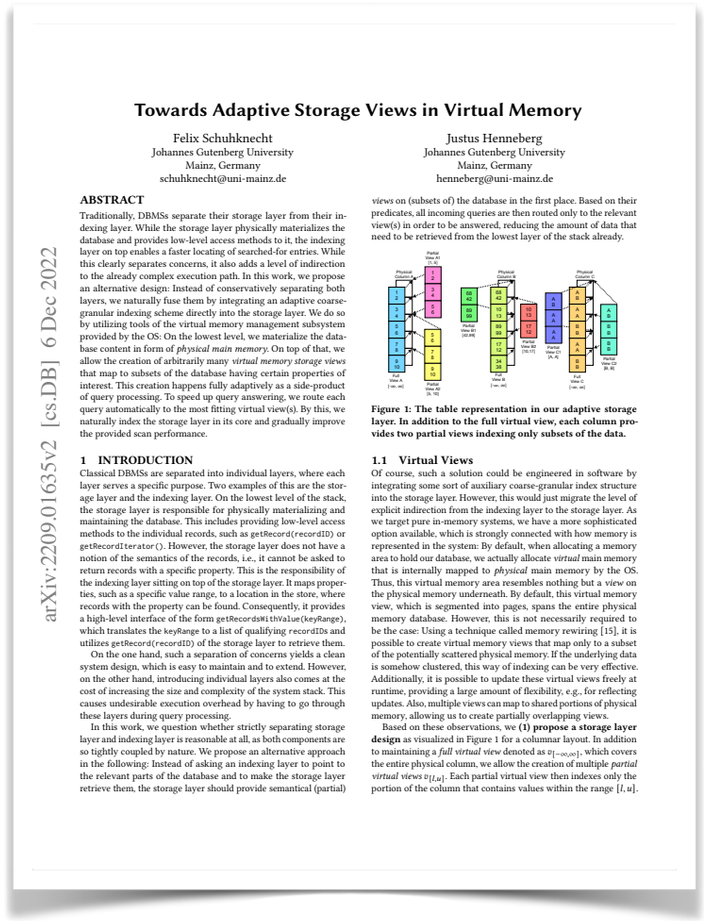
Summary



Long Version: https://arxiv.org/abs/2209.01635

- 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



Towards Adaptive Storage Views in Virtual Memory

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

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

ABSTRACT

Traditionally, DBMSs separate their storage layer from their indexing layer. While the storage layer physically materializes the database and provides low-level access methods to it, the indexing layer on top enables a faster locating of searched-for entries. While this clearly separates concerns, it also adds a level of indirection to the already complex execution path. In this work, we propose an alternative design: Instead of conservatively separating both layers, we naturally fuse them by integrating an adaptive coarse-granular indexing scheme directly into the storage layer. We do so by utilizing tools of the virtual memory management subsystem provided by the OS: On the lowest level, we materialize the database content in form of *physical main memory*. On top of that, we allow the creation of arbitrarily many *virtual memory storage views* that map to subsets of the database having certain properties of interest. This creation happens fully adaptively as a side-product of query processing. To speed up query answering, we route each query automatically to the most fitting virtual view(s). By this, we naturally index the storage layer in its core and gradually improve the provided scan performance.

1 INTRODUCTION

Classical DBMSs are separated into individual layers, where each layer serves a specific purpose. Two examples of this are the storage layer and the indexing layer. On the lowest level of the stack, the storage layer is responsible for physically materializing and maintaining the database. This includes providing low-level access methods to the individual records, such as `getRecord(recordID)` or `getRecordIterator()`. However, the storage layer does not have a notion of the semantics of the records, i.e., it cannot be asked to return records with a specific property. This is the responsibility of the indexing layer sitting on top of the storage layer. It maps properties, such as a specific value range, to a location in the store, where records with the property can be found. Consequently, it provides a high-level interface of the form `getRecordsWithValue(keyRange)`, which translates the `keyRange` to a list of qualifying `recordIDs` and utilizes `getRecord(recordID)` of the storage layer to retrieve them.

On the one hand, such a separation of concerns yields a clean system design, which is easy to maintain and to extend. However, on the other hand, introducing individual layers also comes at the cost of increasing the size and complexity of the system stack. This causes undesirable execution overhead by having to go through these layers during query processing.

In this work, we question whether strictly separating storage layer and indexing layer is reasonable at all, as both components are so tightly coupled by nature. We propose an alternative approach in the following: Instead of asking an indexing layer to point to the relevant parts of the database and to make the storage layer retrieve them, the storage layer should provide semantical (partial) *views* on (subsets of) the database in the first place. Based on their predicates, all incoming queries are then routed only to the relevant view(s) in order to be answered, reducing the amount of data that need to be retrieved from the lowest layer of the stack already.

Figure 1: The table representation in our adaptive storage layer. In addition to the full virtual view, each column provides two partial views indexing only subsets of the data.

1.1 Virtual Views

Of course, such a solution could be engineered in software by integrating some sort of auxiliary coarse-granular index structure into the storage layer. However, this would just migrate the level of explicit indirection from the indexing layer to the storage layer. As we target pure in-memory systems, we have a more sophisticated option available, which is strongly connected with how memory is represented in the system: By default, when allocating a memory area to hold our database, we actually allocate *virtual* main memory that is internally mapped to *physical* main memory by the OS. Thus, this virtual memory area resembles nothing but a view on the physical memory underneath. By default, this virtual memory view, which is segmented into pages, spans the entire physical memory database. However, this is not necessarily required to be the case: Using a technique called memory rewiring [15], it is possible to create virtual memory views that map only to a subset of the potentially scattered physical memory. If the underlying data is somehow clustered, this way of indexing can be very effective. Additionally, it is possible to update these virtual views freely at runtime, providing a large amount of flexibility, e.g., for reflecting updates. Also, multiple views can map to shared portions of physical memory, allowing us to create partially overlapping views.

Based on these observations, we **(1) propose a storage layer design** as visualized in Figure 1 for a columnar layout. In addition to maintaining a *full virtual view* denoted as $v_{[-\infty,\infty]}$, which covers the entire physical column, we allow the creation of multiple *partial virtual views* $v_{[l,u]}$. Each partial virtual view then indexes only the portion of the column that contains values within the range $[l, u]$.