CowabungaDB

George Berdovskiy

Nate Buttke

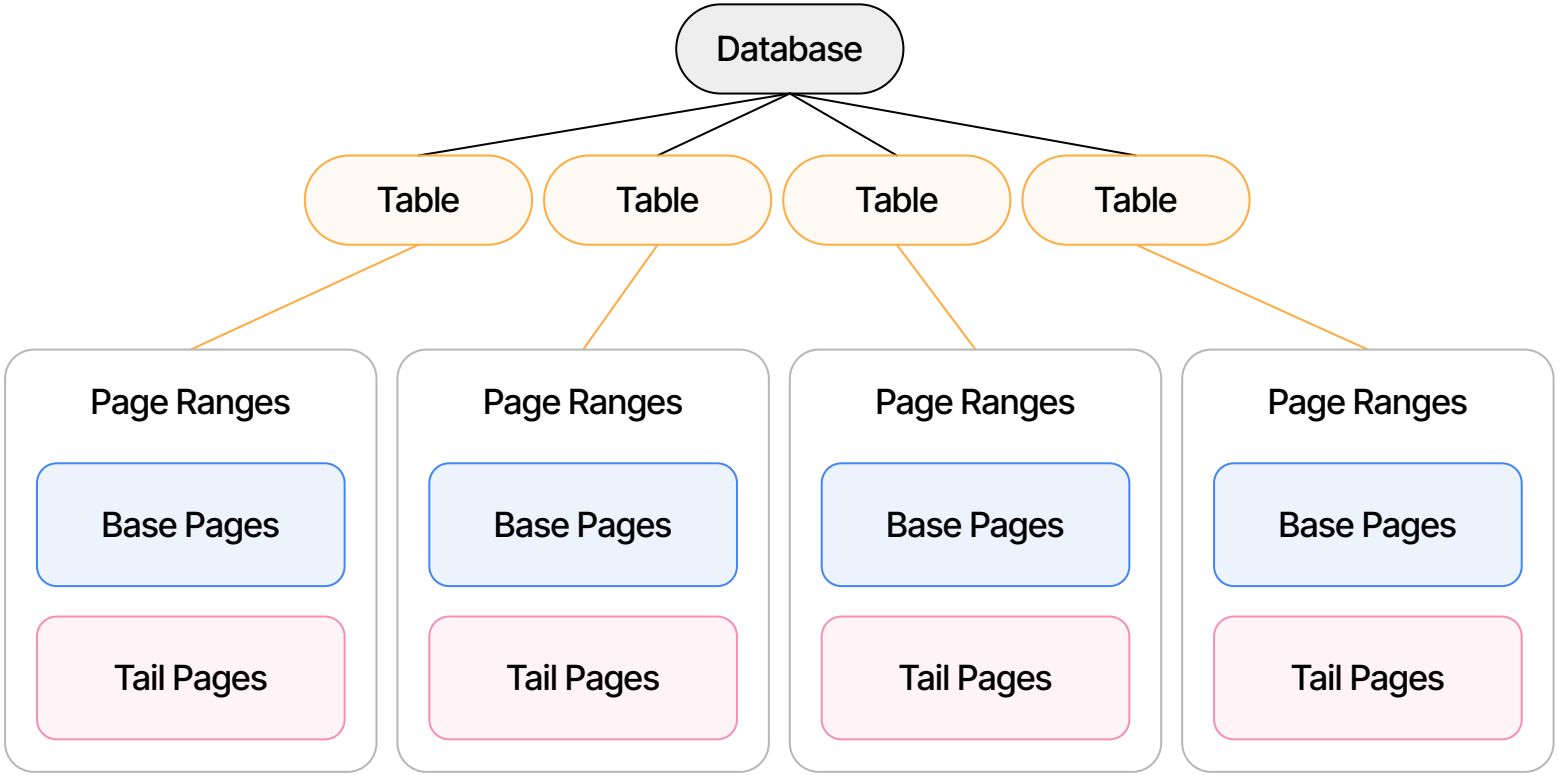
Kevin Bao

Keyur Parikh

Marcin Wróblewski

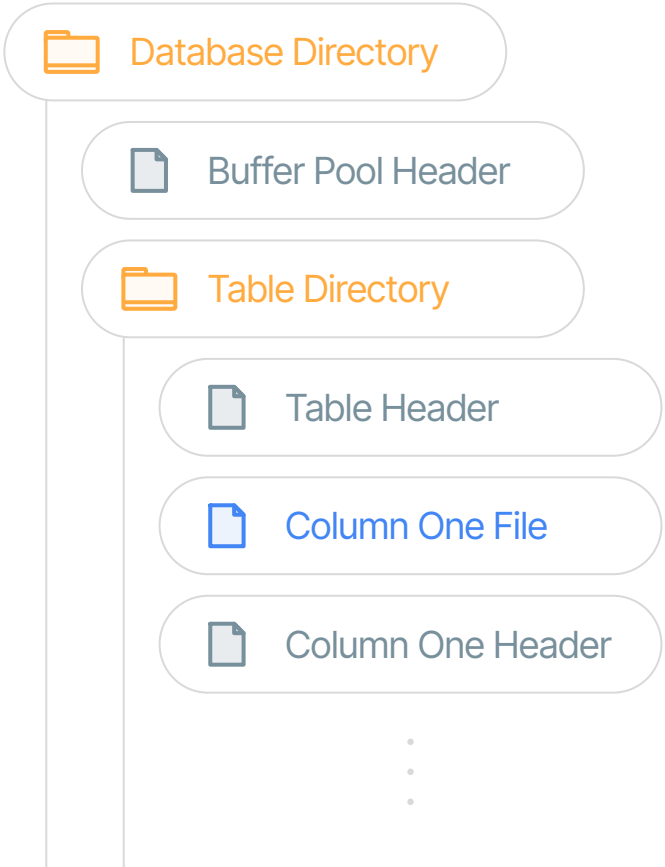
Milestone One Review

Overall Design



Persistence

File Structure



Database Directory

Stores every table in database, determined by `db.open(...)`.

Buffer Pool Header

Stores buffer pool management data (e.g. page identifier map).

Table Directory

Stores table data. Uses numeric identifier instead of name.

File Structure

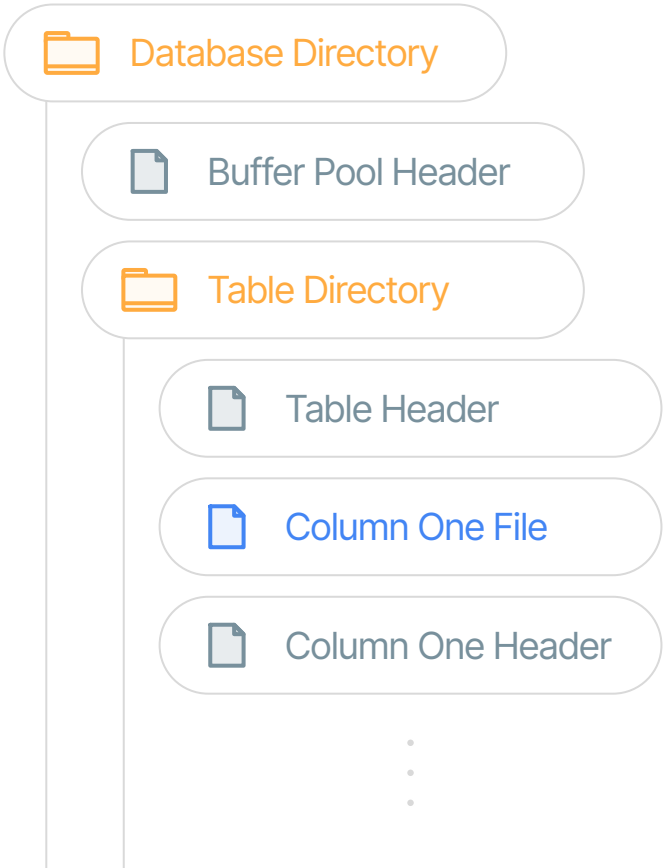


Table Header

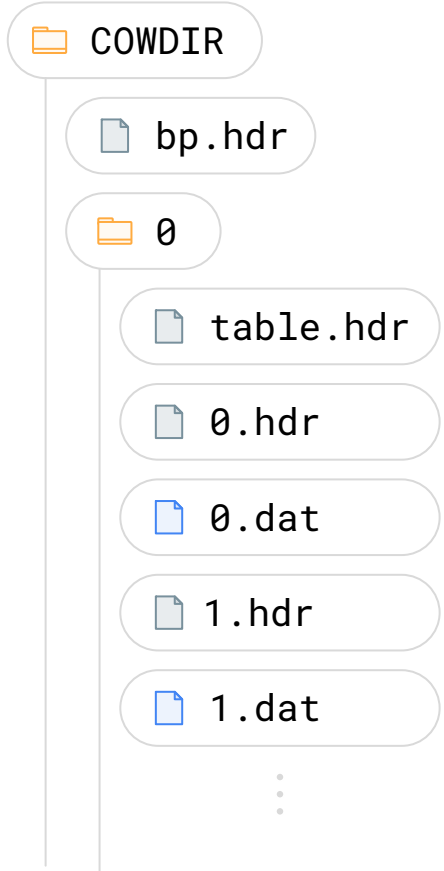
Stores page ranges and metadata like table name, number of columns...

Column File / Header

File stores actual data, header stores next available page slot.

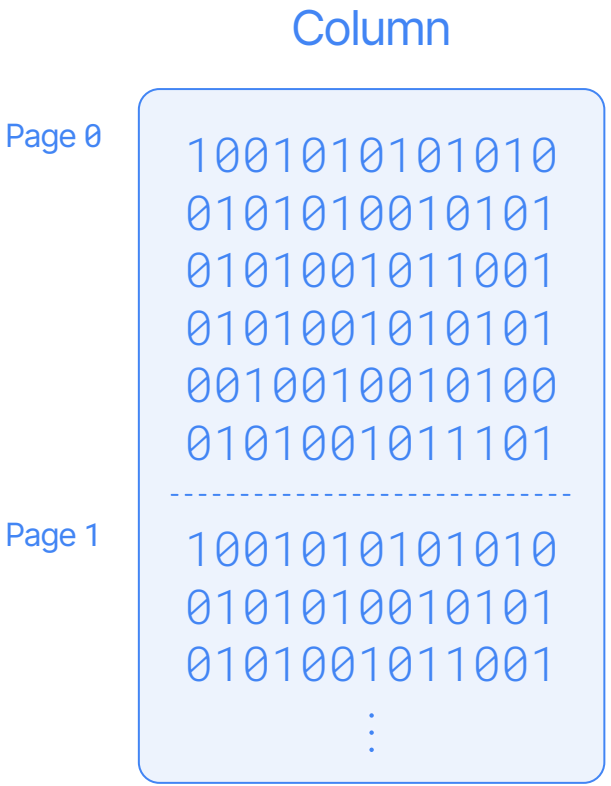
Example File Structure

```
db = Database()
db.open("./COWDIR")
grades_table = db.create_table('Grades', 5, 0)
query = Query(grades_table)
```



Column Files

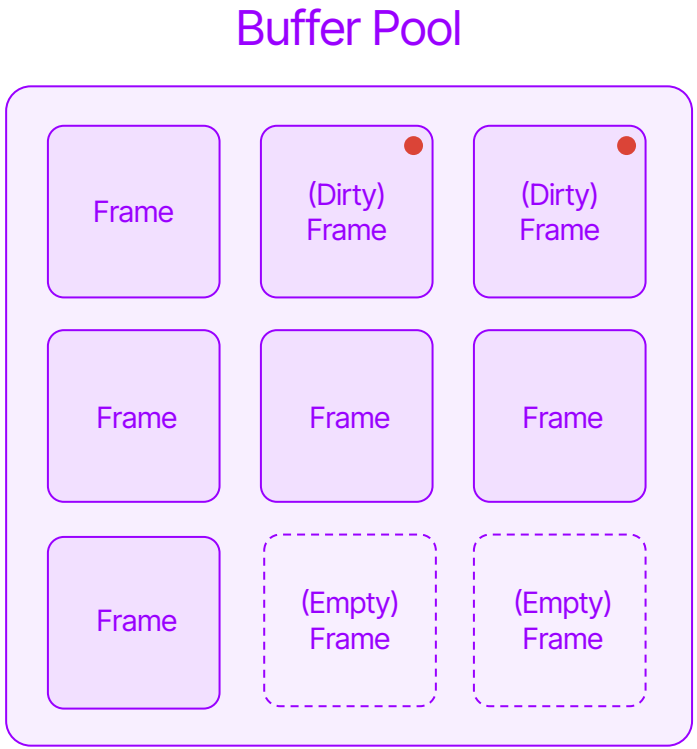
- Written as bytes (no serialization, per specification)
- Array of 512 `i64` values → buffer of 4096 `u8` values
- Seek correct byte offset and write entire buffer
    - Know this from the **physical page ID**
    - Includes table identifier, column identifier, and physical page identifier (zero indexed from beginning of file)



Buffer Pool

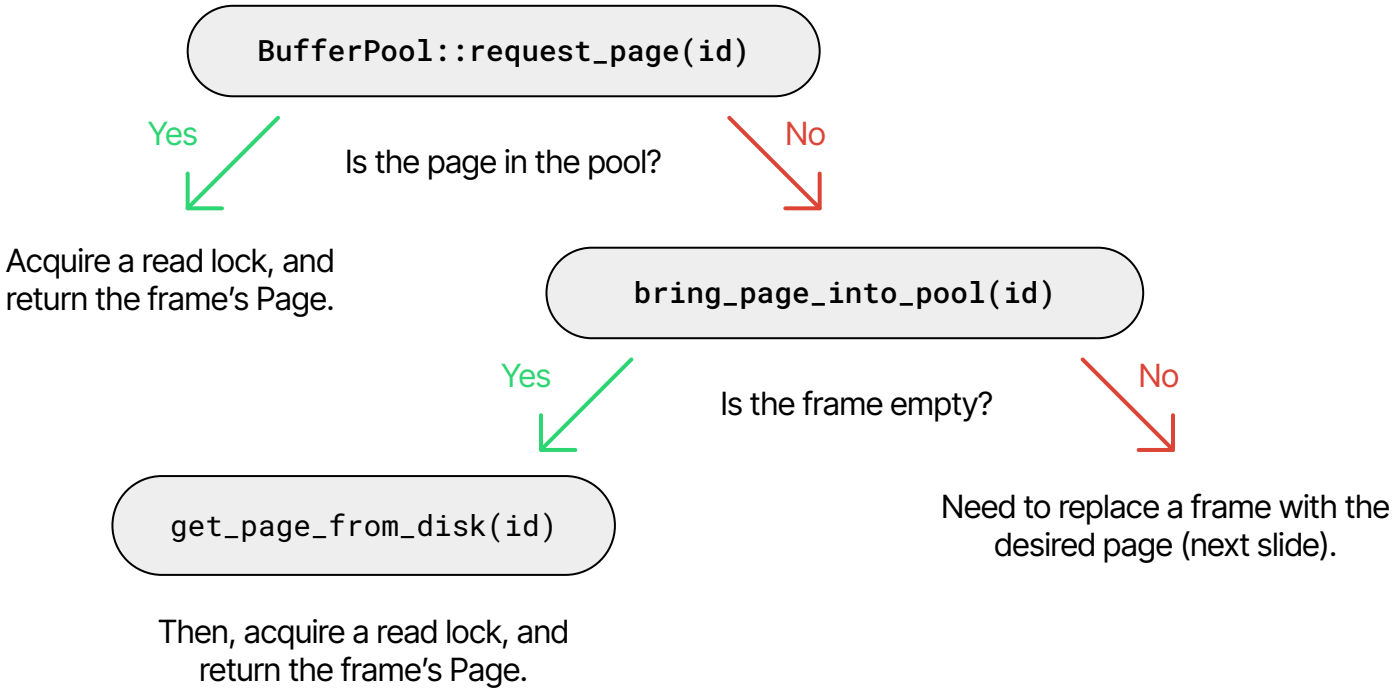
General Design

- At core, list of frames protected by smart pointers wrapping read-write locks (for memory / thread safety)
  - `Arc<RwLock<Frame>>`
- Each `Frame` has...
  - `Page` (in memory) or `None`
  - Boolean fields for `dirty` and `empty` states
  - Its `PhysicalPageID`
- The `Arc` is used to measure **pin count**
  - *Guaranteed to be accurate*



Eviction Process

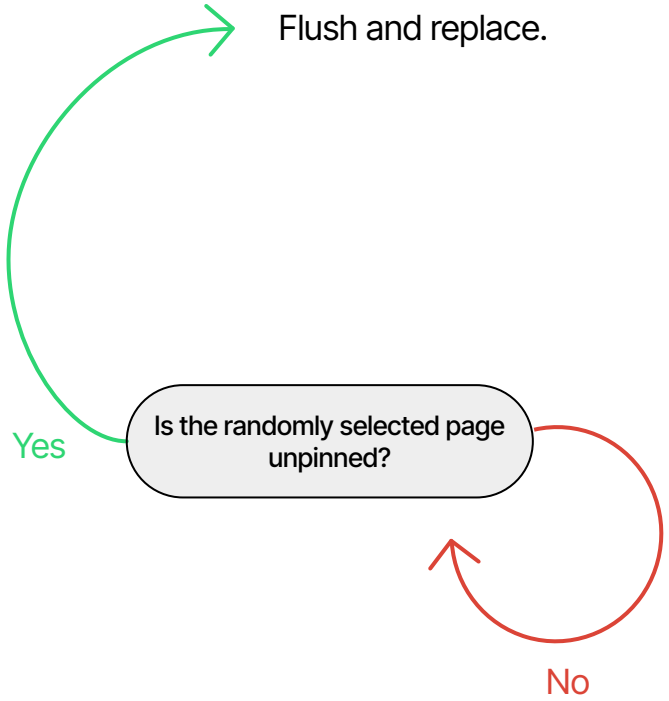
Part One



Eviction Process

Part Two

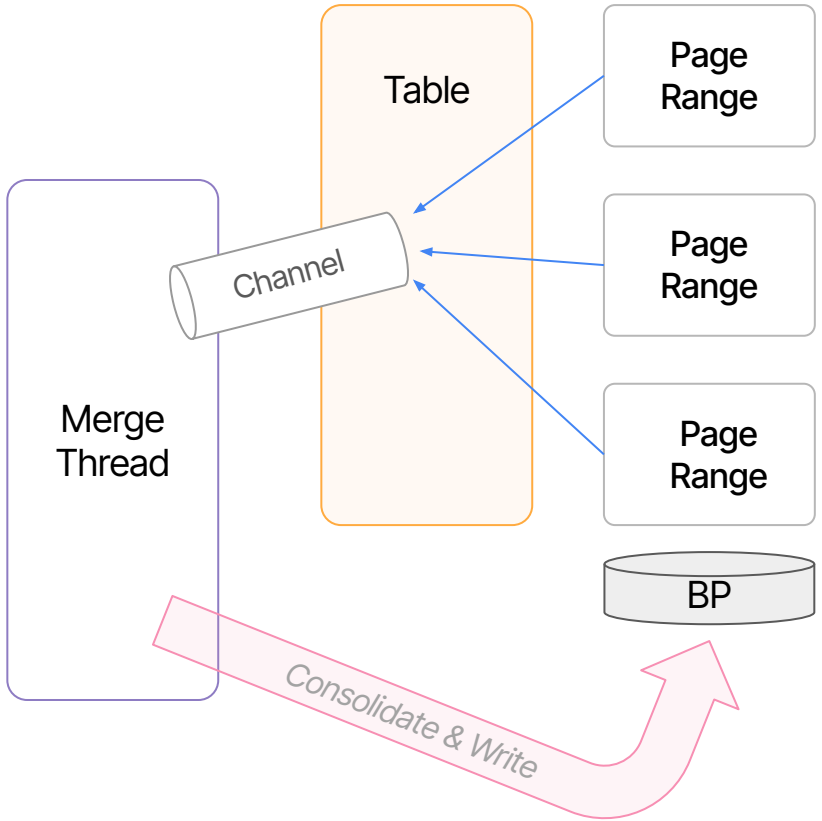
- When there are no empty `Frames`, one must be replaced with the desired page
  - First, prioritize unpinned pages to avoid waiting
  - Second, randomly check `Frame` pin counts until we find one without pins
  - This is effectively *busy waiting*
- The selected `Frame` is flushed if needed, then replaced.
- The busy waiting strategy represents a *tradeoff*
  - It makes it unnecessary to do accounting of unpinned pages at every possible modification



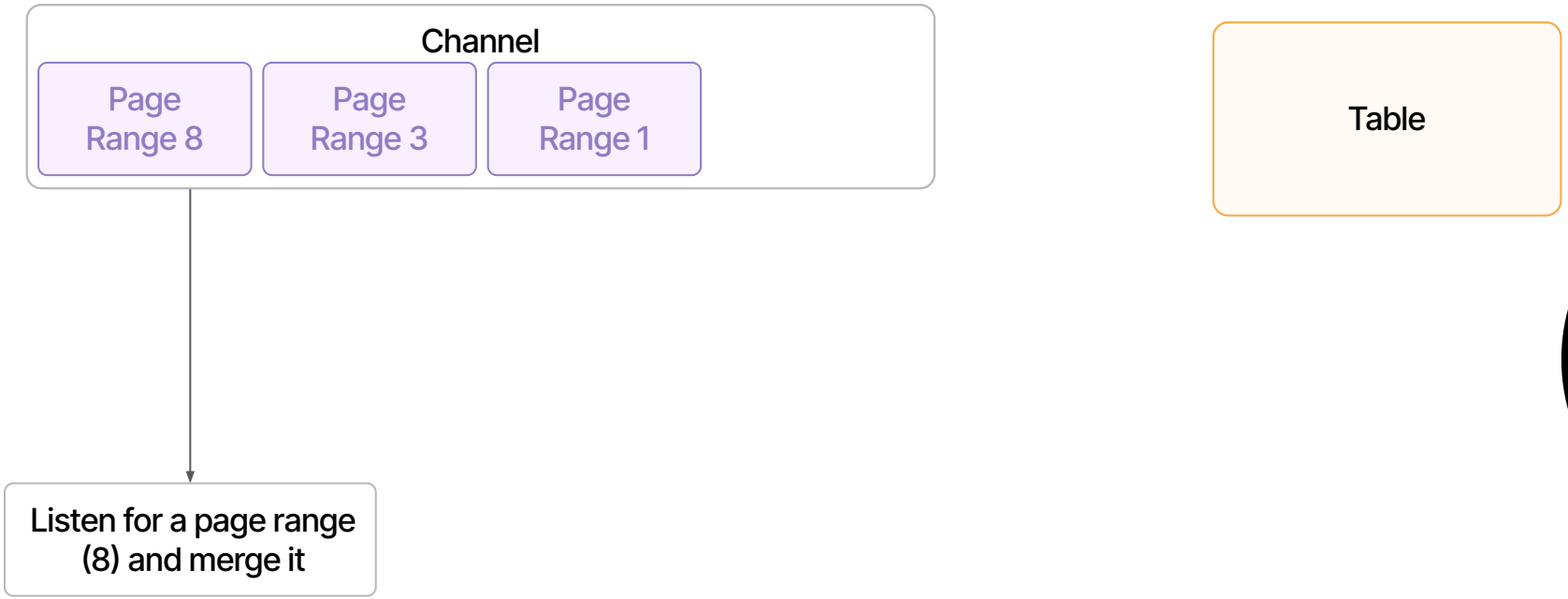
Merge

Writes for Merge

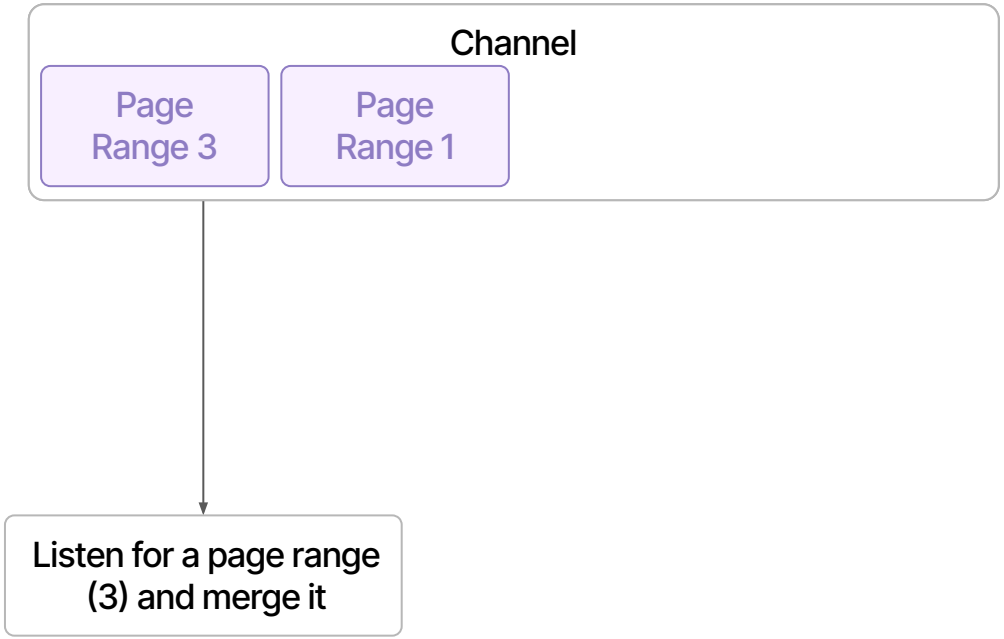
- Array of 512 `i64` values → buffer of 4096 `u8` values
- Seek correct byte offset and write entire buffer
    - Know this from the **physical page ID**
    - Includes table identifier, column identifier, and physical page identifier
- Merge is called after 500 updates (by default)
    - Each `PageRange` to be merged is placed into a channel (queue that serializes access)
        - Table sends, `merge_thread` receives requests

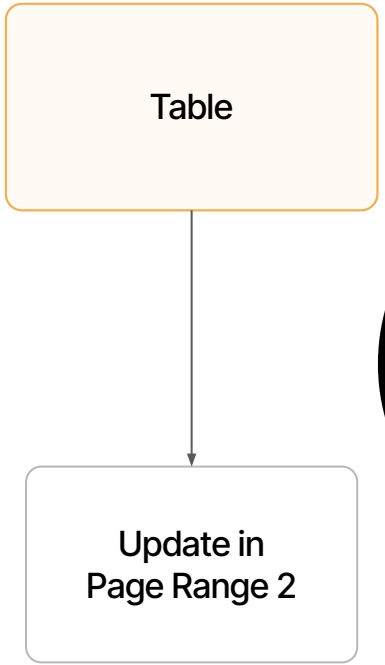


Merge: Initialization

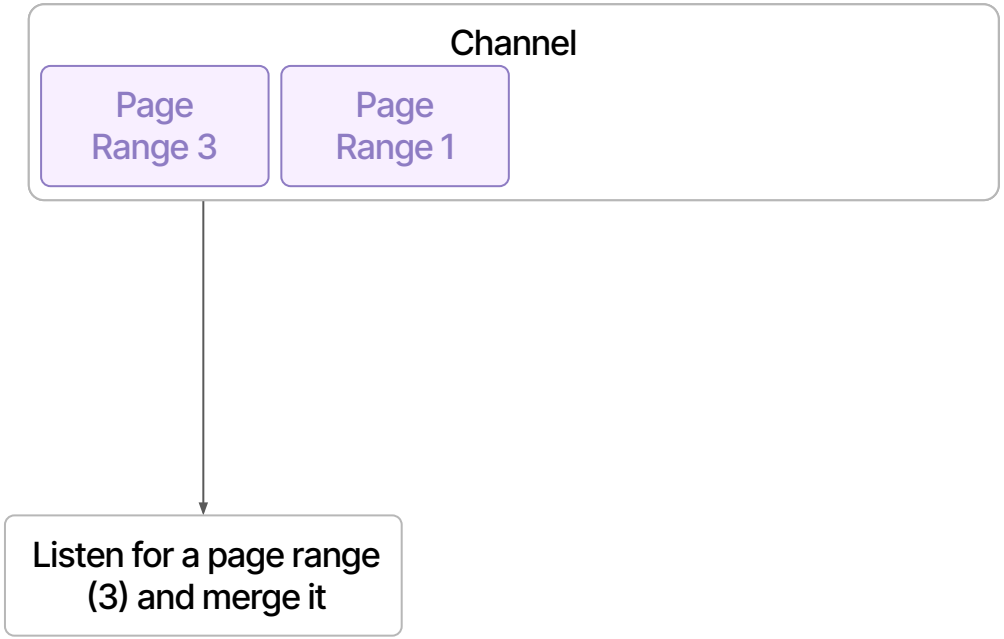


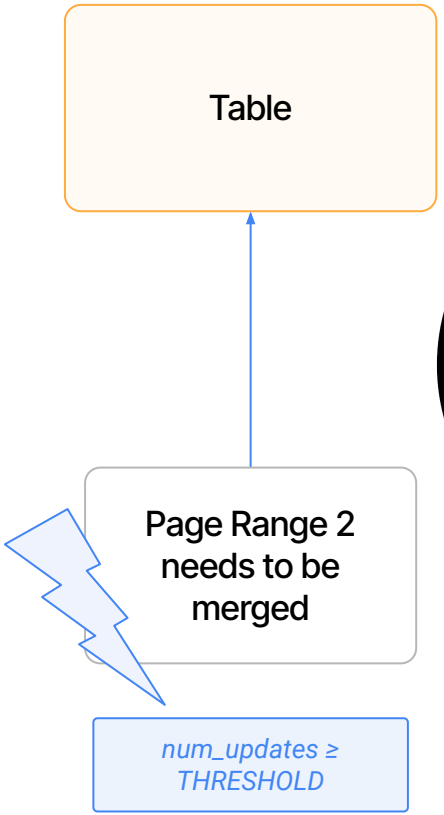
Merge: Initialization



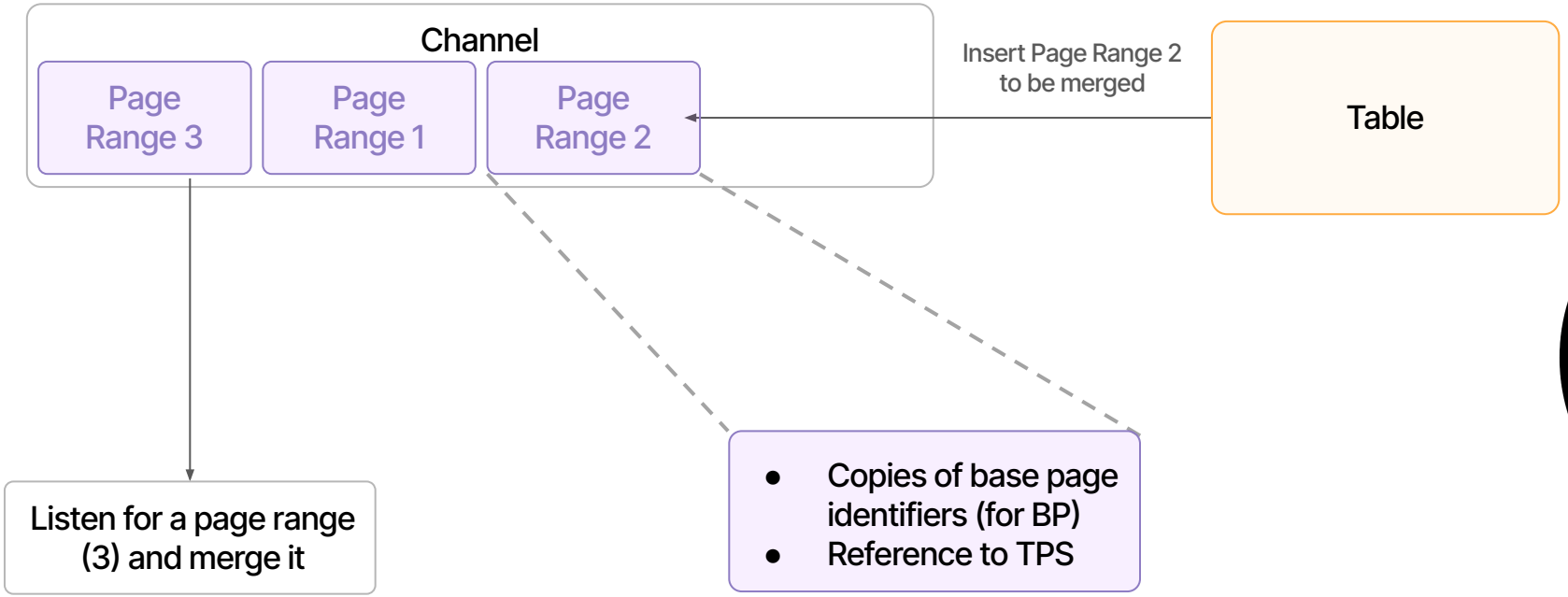


Merge: Initialization

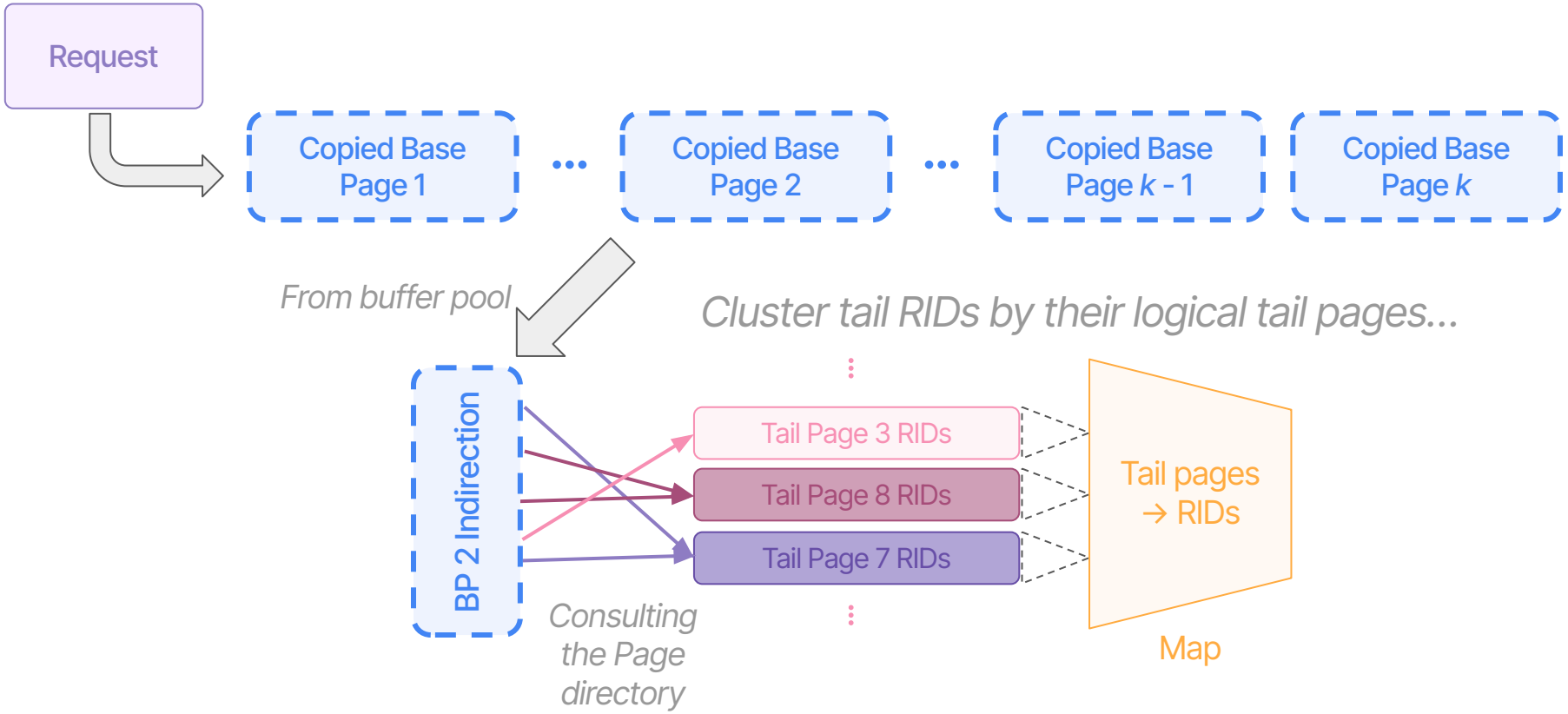




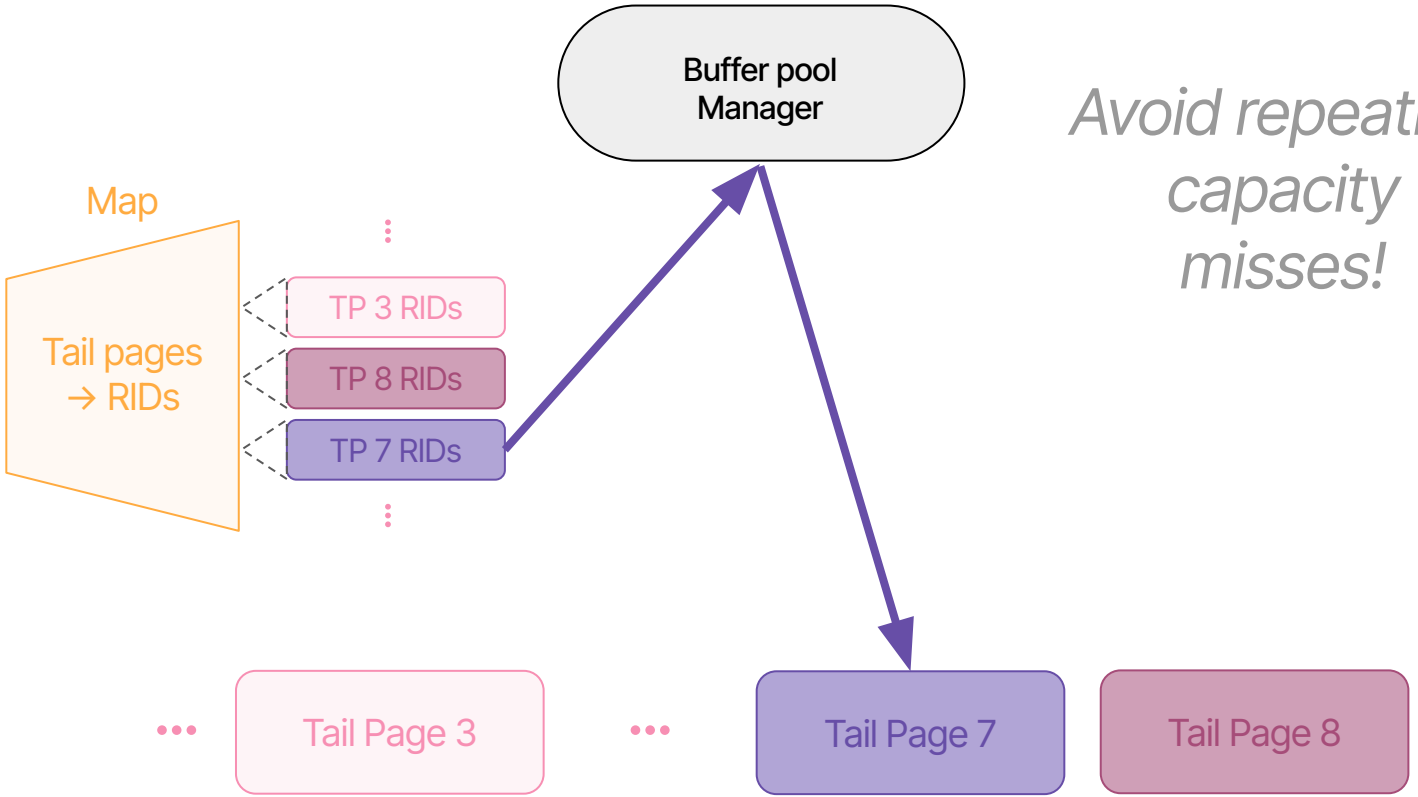
Merge: Initialization



Merge: Tail Records

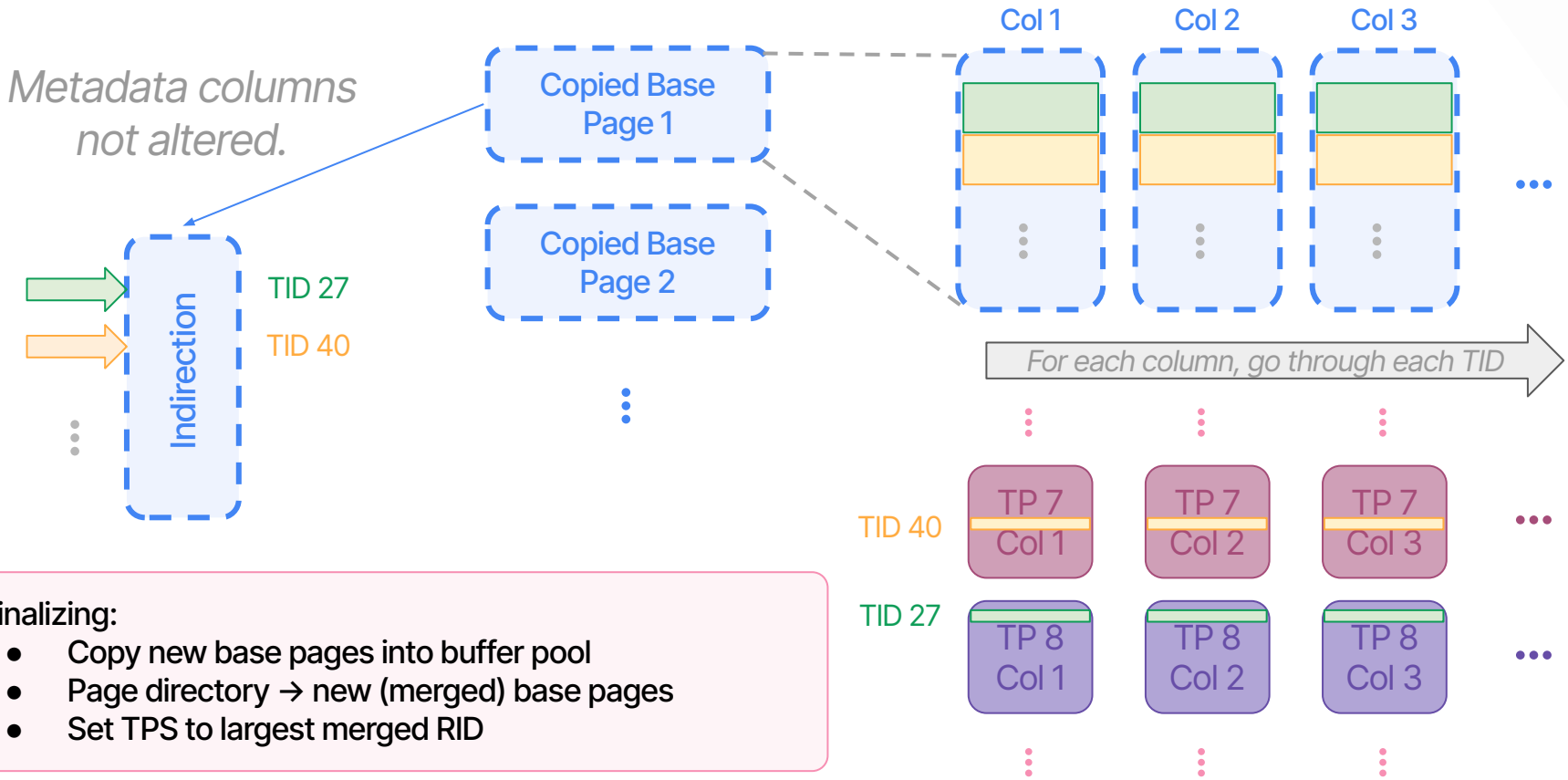


Merge: Tail Records



Avoid repeating capacity misses!

Merge: Consolidation



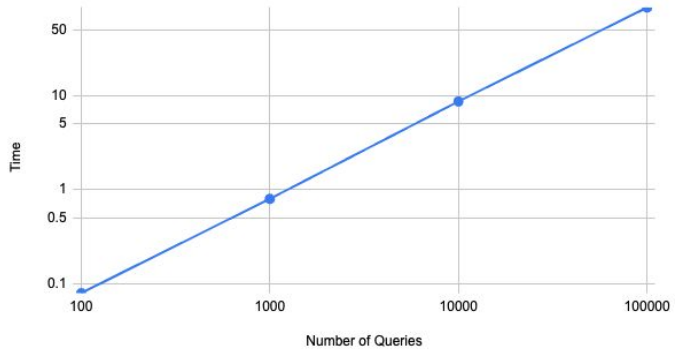
Finalizing:

- Copy new base pages into buffer pool
- Page directory → new (merged) base pages
- Set TPS to largest merged RID

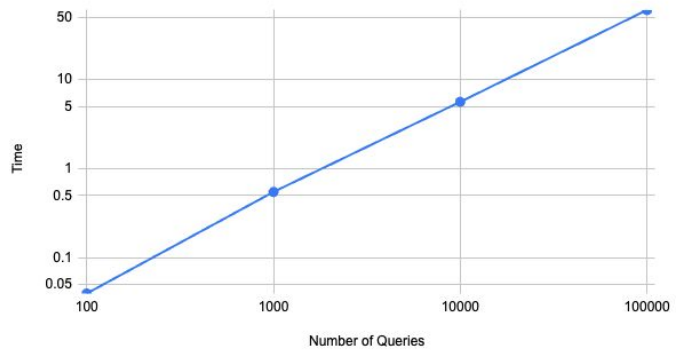
Evaluation

Query Performance

Insert

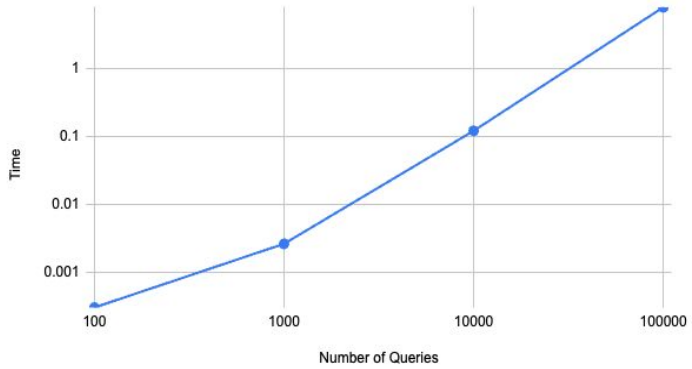


Update

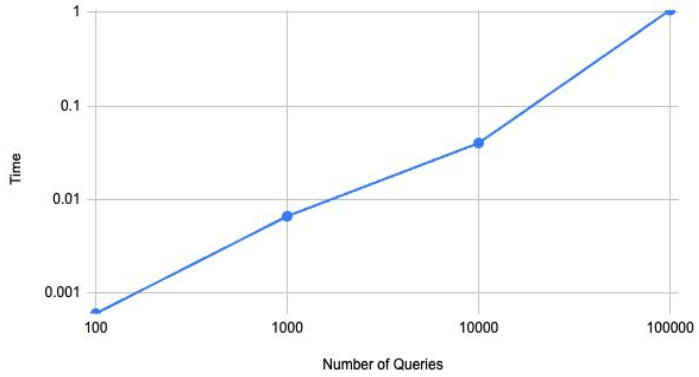


Query Performance

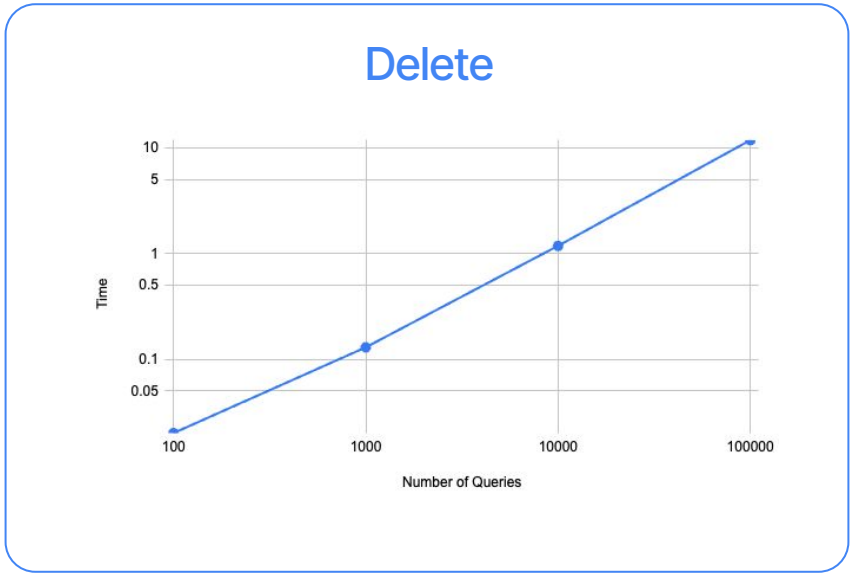
Sum



Select

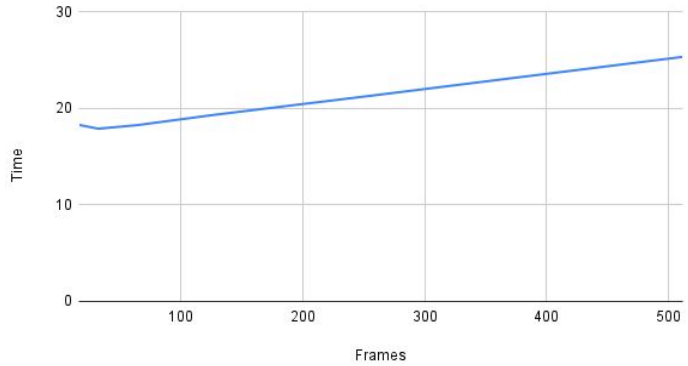


Query Performance

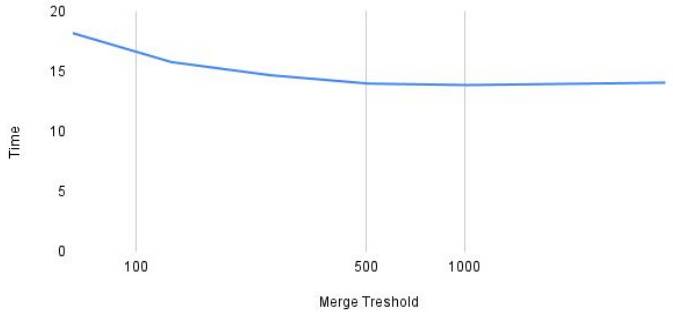


Buffer Pool and Merge

Time vs. Buffer Pool Frames

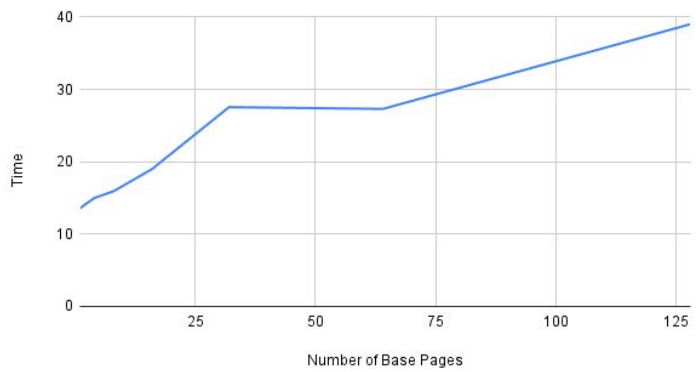


Time vs. Merge Threshold

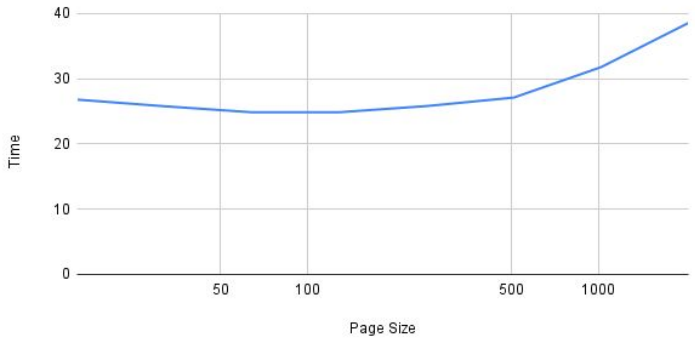


Page Range Size and Page Size

Time vs. Number of Base Pages / Range



Time vs. Page Size



Demonstration