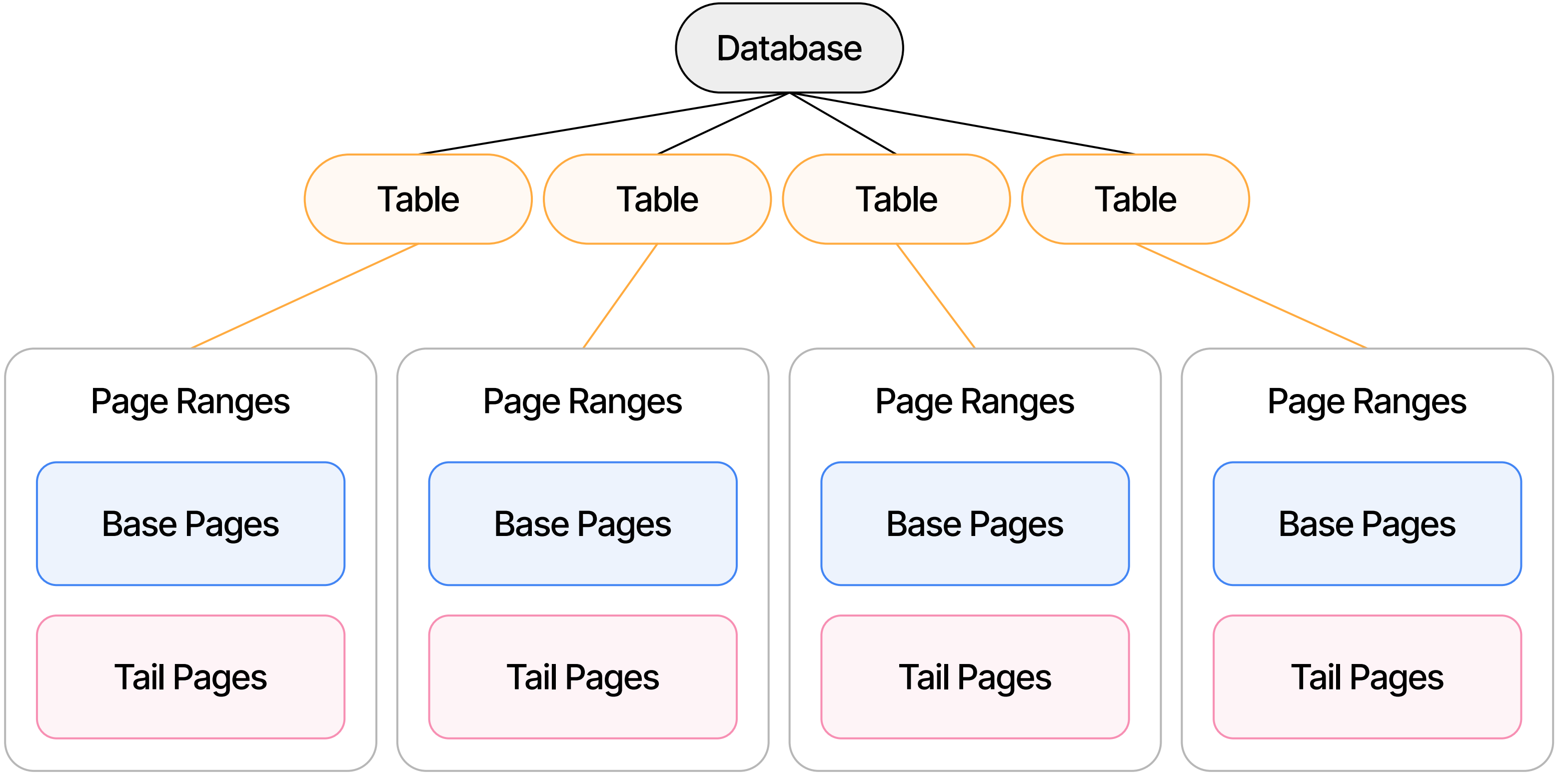
CowabungaDB

George Berdovskiy
Nate Buttke
Kevin Bao
Keyur Parikh
Marcin Wróblewski

Overall Design



Database

Table

Table

Table

Table

Page Ranges

Base Pages

Tail Pages

Page Ranges

Base Pages

Tail Pages

Page Ranges

Base Pages

Tail Pages

Page Ranges

Base Pages

Tail Pages

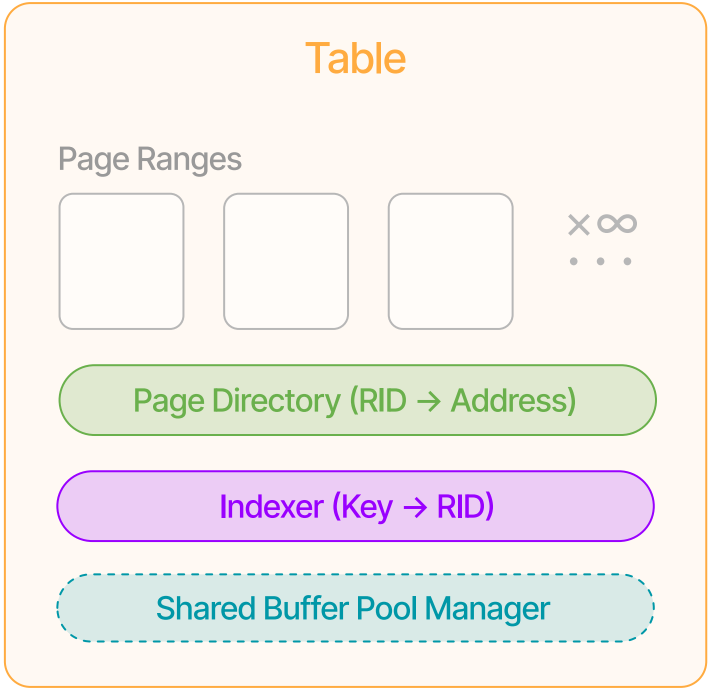
Table

Overview

- Vector of **page ranges** for storing data
- Also holds the **indexer** and **page directory**
    - Indexer is a B-Tree that maps keys to sets of RIDs
    - Page directory maps RIDs to addresses
- Holds reference to **shared buffer pool manager**

Implementation

- Struct `Table` holding structures above
    - Additional fields for number of columns, table name, and so on
- Exposed to Python via `PyO3`
- Buffer pool accessed via `Arc<Mutex<BufferPool>>`
    - *Memory safety!*



Table

Page Ranges

× ∞

Page Directory (RID → Address)

Indexer (Key → RID)

Shared Buffer Pool Manager

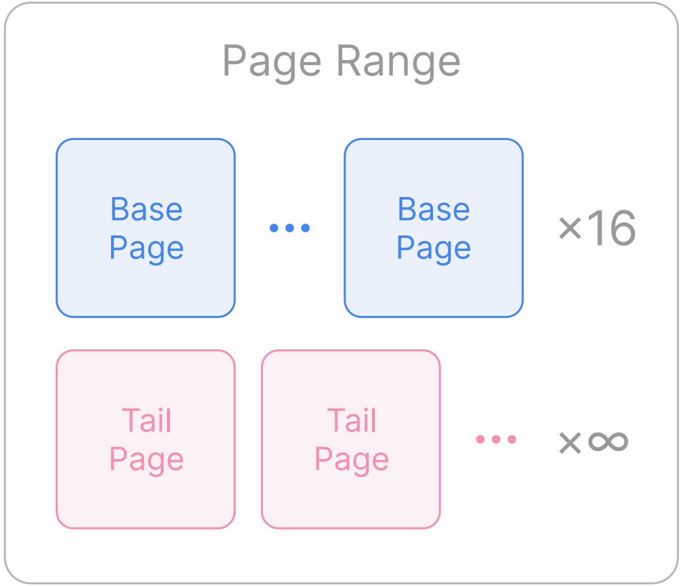
Page Ranges

Overview

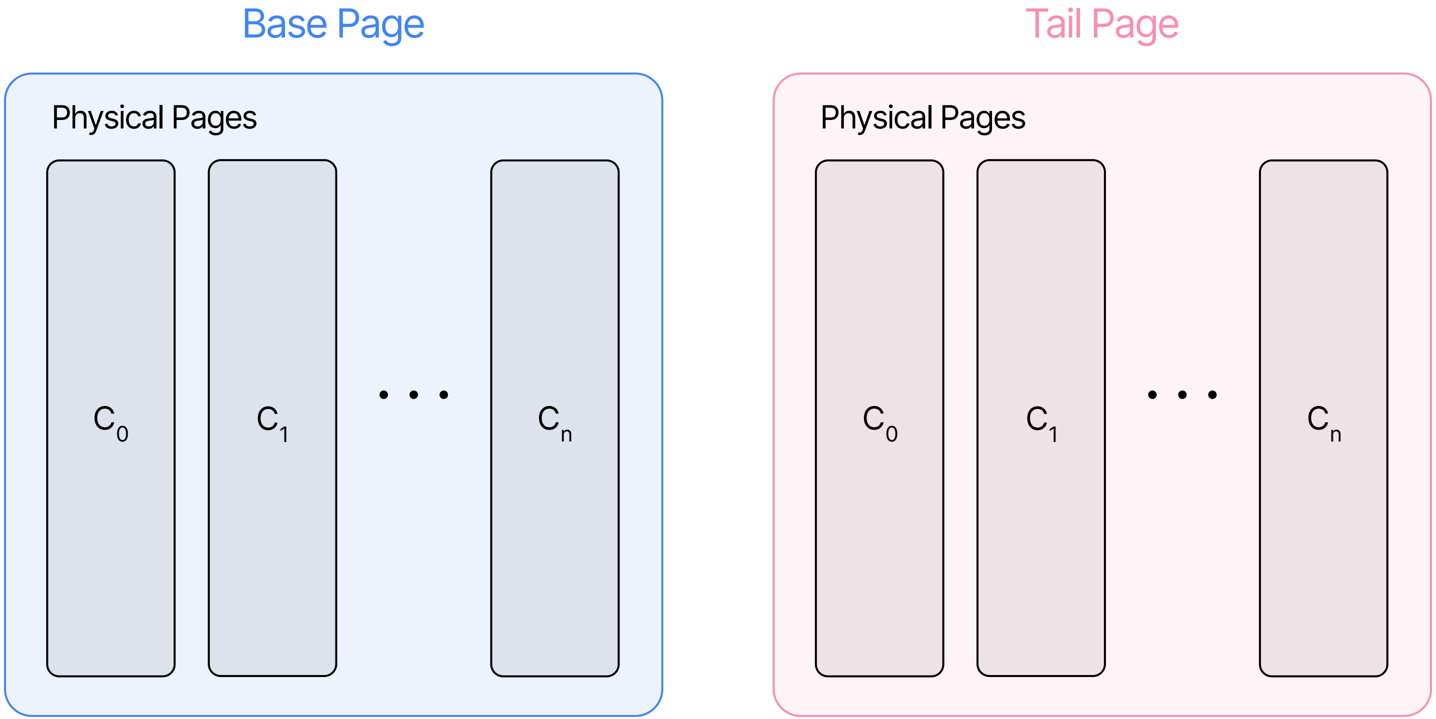
- Dynamic - created as necessary
- Fixed number of **base pages,** unlimited number of **tail pages**
- Serves as component of record address

Implementation

- Struct **`PageRange`** with...
    - Array (fixed size) of **`LogicalPage<Base>`**
    - Vector of **`LogicalPage<Tail>`**
- Generic type arguments → **readability** and **flexible implementation**
    - *We take advantage of the Rust type system to prevent logical bugs as well as memory bugs!*



Logical Pages

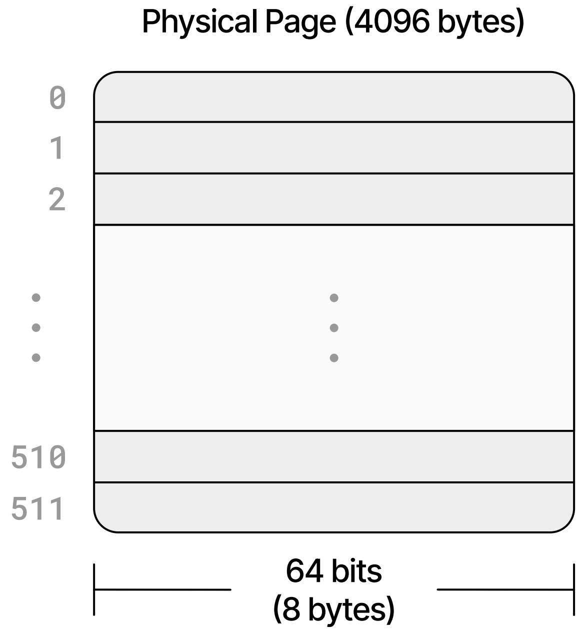


Every physical page represents one column!

Physical Pages

Overview and Implementation

- Physical pages are fixed at **4096** bytes
- "Cells" indexed from 0 to 511
  - Holds 512 values
- Values are either **`None`** or an **`i64`**
- Represented by struct **`Page`**



Buffer Pool Interface

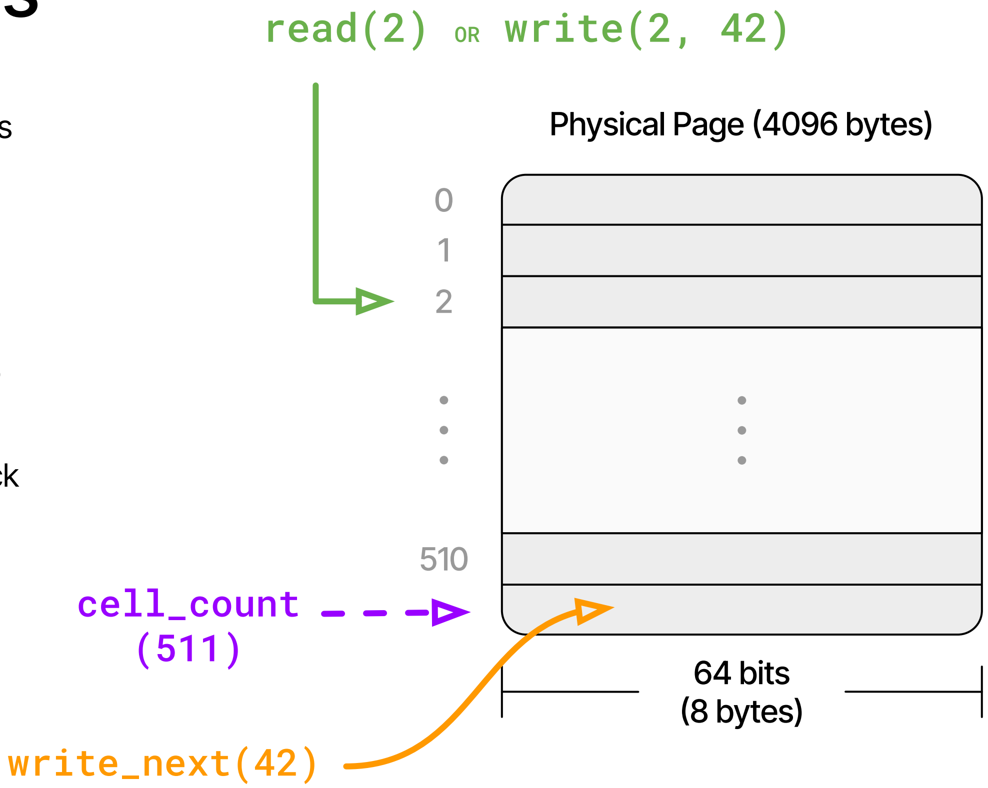
Virtual Buffer Pool

Overview and Implementation

- Although our database is volatile, it's built on top of a **virtual buffer pool**
- Future durability changes should minimally impact more abstract areas of the codebase
- Implemented as BufferPool struct with several methods for reading to / writing from physical pages and cells
- Buffer pool is **shared by all tables!**
- Rust doesn't allow multiple mutable references → wrap with `Arc<Mutex>>`
  - `Arc` → "smart pointer enabling sharing data between threads" 🔗
  - `Mutex` → locks / unlocks value 🔗
  - The result is **memory safety**

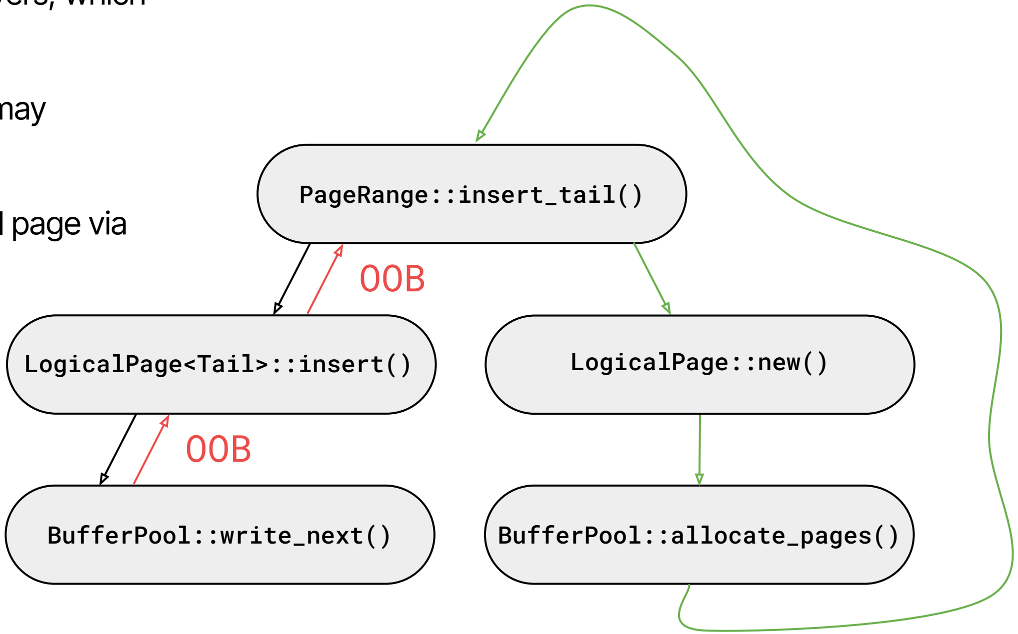
"Buffer Pool" Methods

- In addition to **`Cell`** and **Page** structs, interfaces for reading and writing from pages also provided
    - Namely **`write()`**, **`write_next()`**, and **`read()`**
- Many writes to physical pages (e.g. during **`insert()`** and **`update()`** queries) write to the next available **`Cell`** in a page
- We maintain a **`cell_count`** that keeps track of the next available index / number of occupied cells



"Buffer Pool" Space Handling

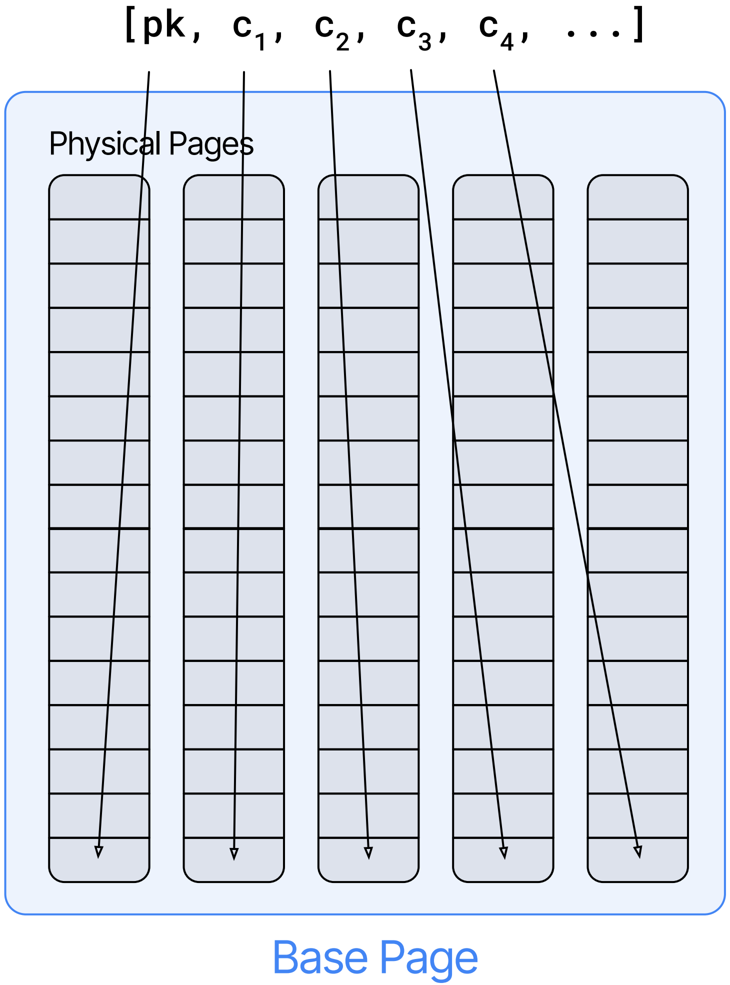
- What happens when a page is filled? Or a write is made to a nonexistent page?
- Errors propagate upward into more abstract layers, which prompts allocation
- Example - `PageRange::insert_tail()` may receive an `Err` originating in `write_next()`
- `insert_tail()` must now allocate a new tail page via `LogicalPage::new()`, which itself calls `BufferPool::allocate_pages()`



Queries

Insert

- Arguments…
    - An array of column values
- The primary key may be **any column**
- Insert adds this record to the next available base page along with relevant metadata columns



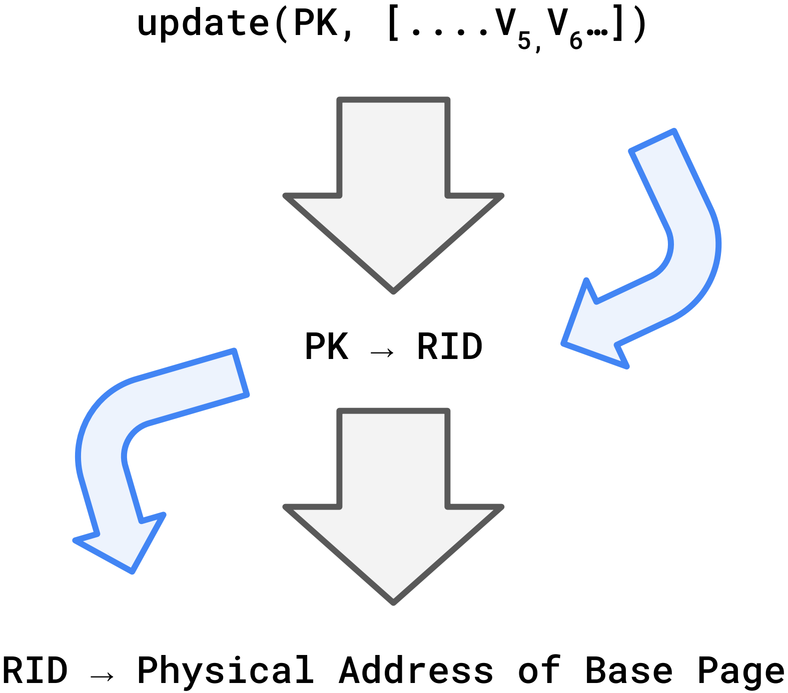
$[pk, c_1, c_2, c_3, c_4, ...]$

Physical Pages

Base Page

Update

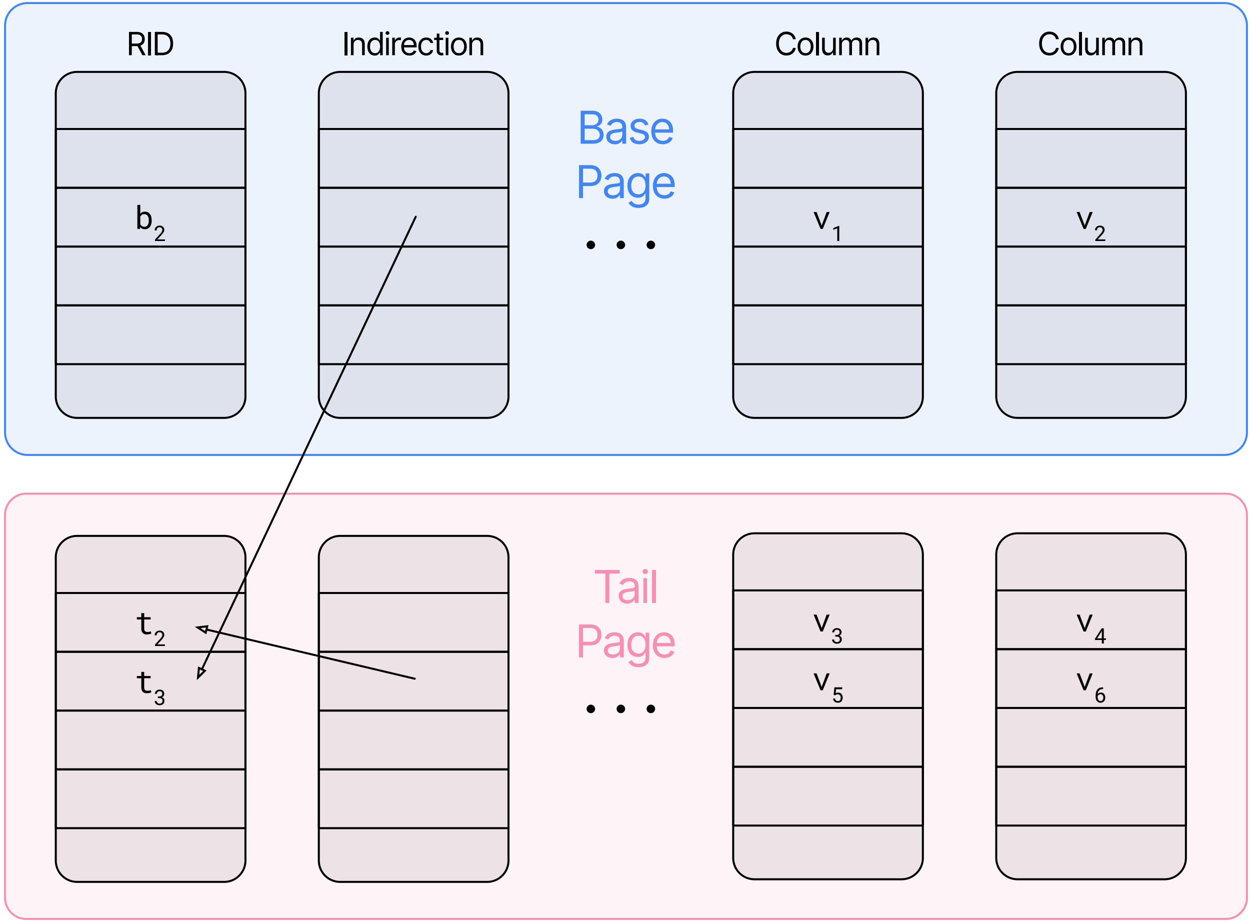
- Takes the primary key and the columns that need to be updated as arguments
- Creates tail record and points indirection column of base page to the RID of the tail page



update(PK, [....V_5,V_6…])

PK → RID

RID → Physical Address of Base Page

RID
Indirection
Column
Column
Base
Page
• • •
b2
v1
v2
Tail
Page
• • •
t2
t3
v3
v4
v5
v6

Select

- Arguments…
    - Search key, search key index, and the columns the user wants (the **projection**)
- Obtain the RIDs of the records that match the search key from the appropriate index tree
- **`select_by_rid`** - for a given RID, peek at a base (and possibly tail) record and return the appropriate record only containing the projected columns (indicated via 0's or 1's as appropriate)

`select(SK, SK index, projected columns)`

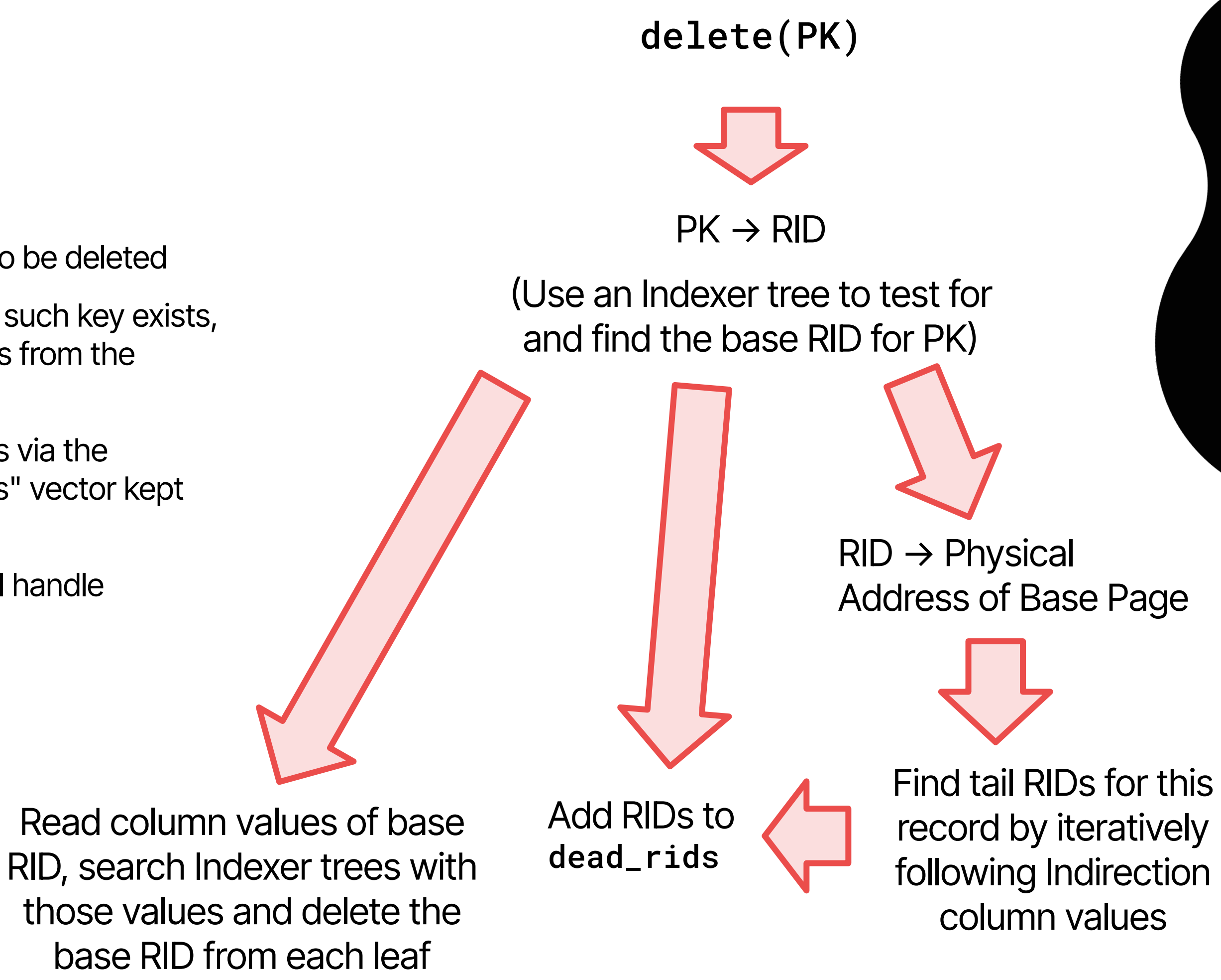
`SK → RID`
(Using the B-Tree corresponding to the search key index)

`RID → base page address`
(Check the indirection column to see if there are any updates)

Return updated values in the projected columns

Delete

- Arguments
  - Primary key of the record to be deleted
- First make sure that a record with such key exists, and if it does we remove its entries from the indexer
- Add the base RID and any tail RIDs via the indirection column to a "dead RIDs" vector kept by the **`Table`**
- This a *logical delete,* which we will handle properly in Milestone 2



`delete(PK)`

PK → RID

(Use an Indexer tree to test for and find the base RID for PK)

RID → Physical Address of Base Page

Find tail RIDs for this record by iteratively following Indirection column values

Add RIDs to **`dead_rids`**

Read column values of base RID, search Indexer trees with those values and delete the base RID from each leaf

Sum

- Arguments
    - Start range, end range, and the column in which we want a sum
- Obtain all RIDs in the range from the appropriate indexer tree
- Call **`select_by_rid`** on each RID, projecting on the column of interest and adding the result to a running total
    - Return this total!

$$\text{RID range} := \{\text{RID}_i \mid y \in [\text{start}, \text{end}] \text{ and } \text{RID}_i \in \text{Indexer}_{\text{col\_index}}(y)\}$$

$$\sum_{x \in \text{RID range}} \text{select\_by\_rid}(x, \text{proj\_on\_col\_index})[0]$$

Indexing

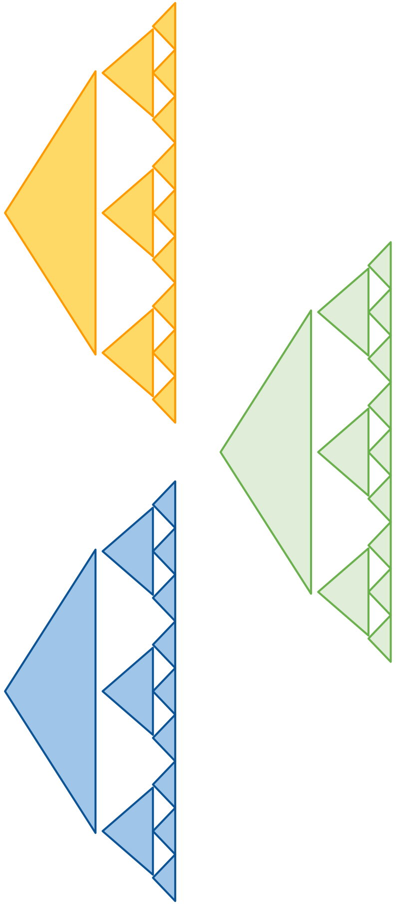
Indexing

Indexer Class - B-Tree for every column, mapping each key to a collection of RIDs...

- Built-in **range** capabilities
- Can trivially acquire all RIDs sharing a key value
- Possesses an "enabled" flag–external schema control

Changes reflected **with every** insertion, update and delete (exposed methods), with tree restructuring or simply just atomic *find-and-alter* actions.

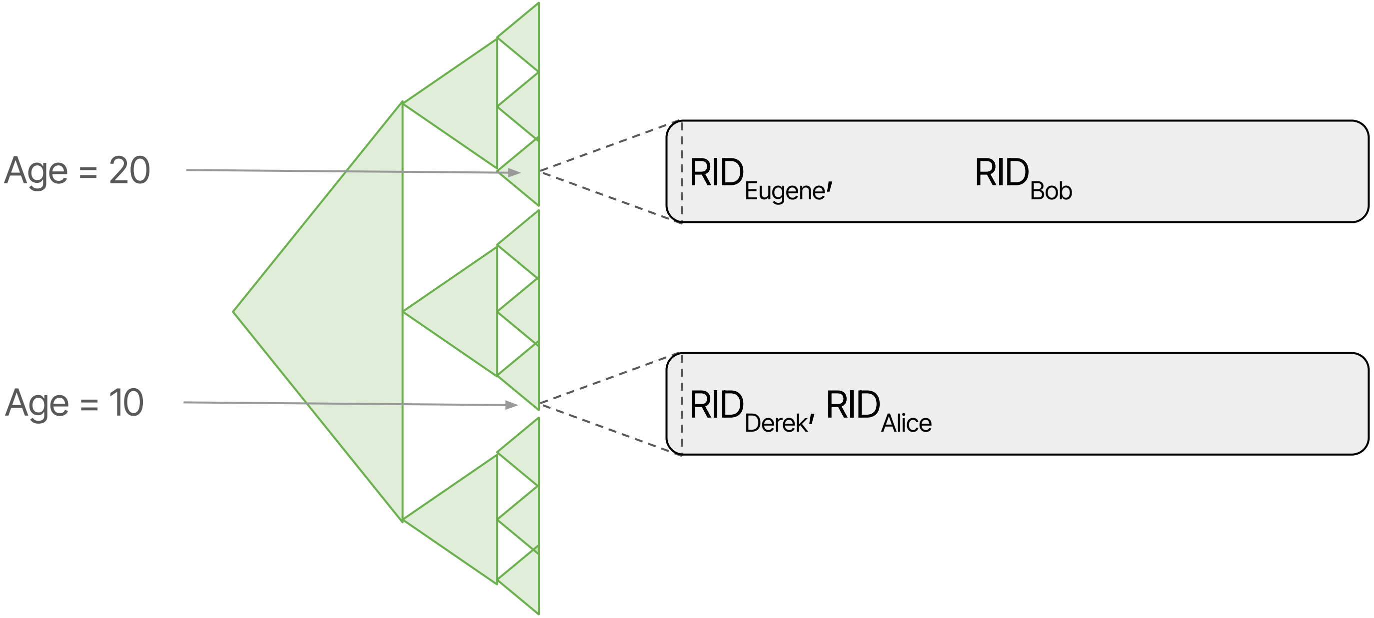
Additional uses - test for presence of a record with some primary key



Rust’s standard BTreeMap is cache efficient (the more a node uses from one I/O, the better), and like *B+* trees allows efficient range-based queries. Amortized, it's fast in general!

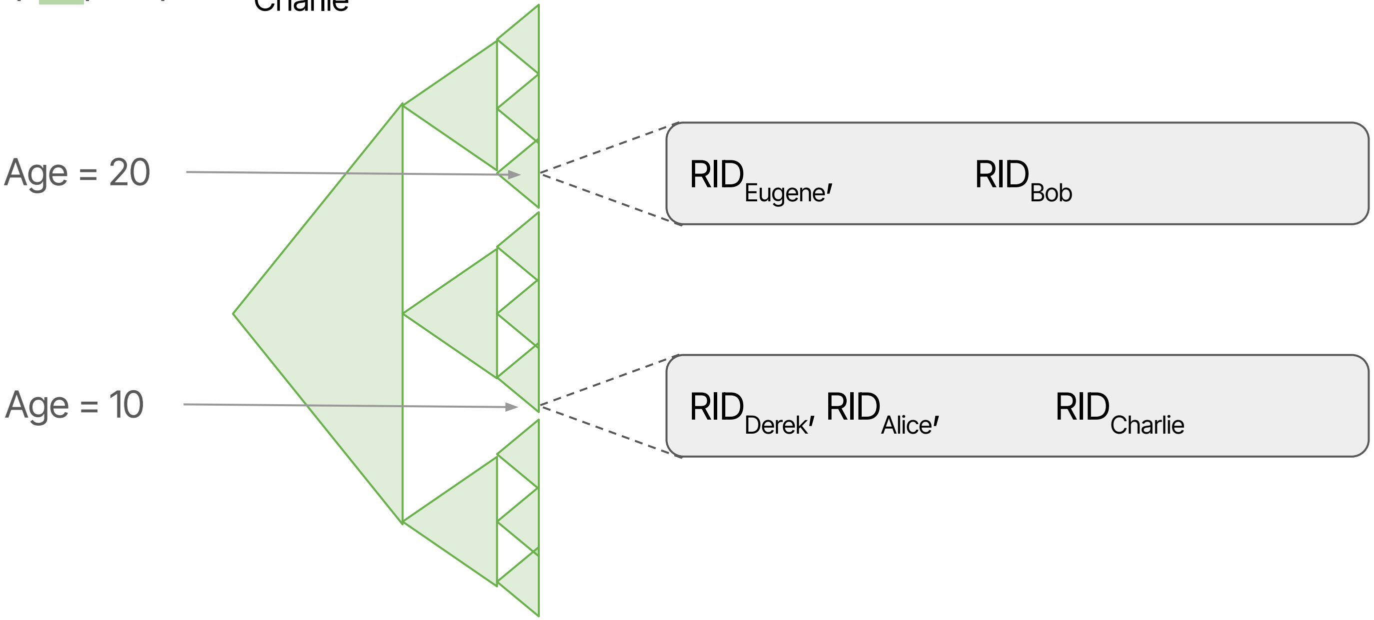
Duplicate keys are a non-issue.

Indexing - Find & Alter



Indexing - Find & Alter

Insert([..., 10, ...], $\text{RID}_{\text{Charlie}}$)



Age = 20

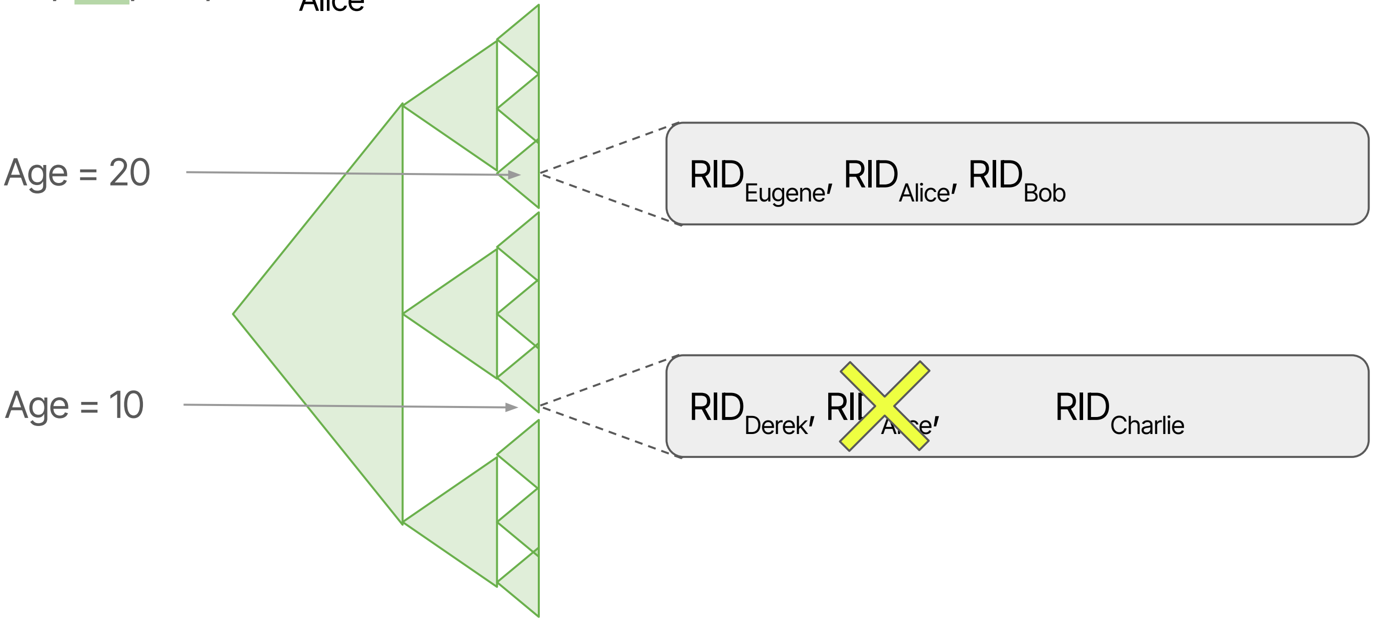
$\text{RID}_{\text{Eugene}}$, RID_{Bob}

Age = 10

$\text{RID}_{\text{Derek}}$, $\text{RID}_{\text{Alice}}$, $\text{RID}_{\text{Charlie}}$

Indexing - Find & Alter

Update([..., 20, ...], $\text{RID}_{\text{Alice}}$)



Age = 20

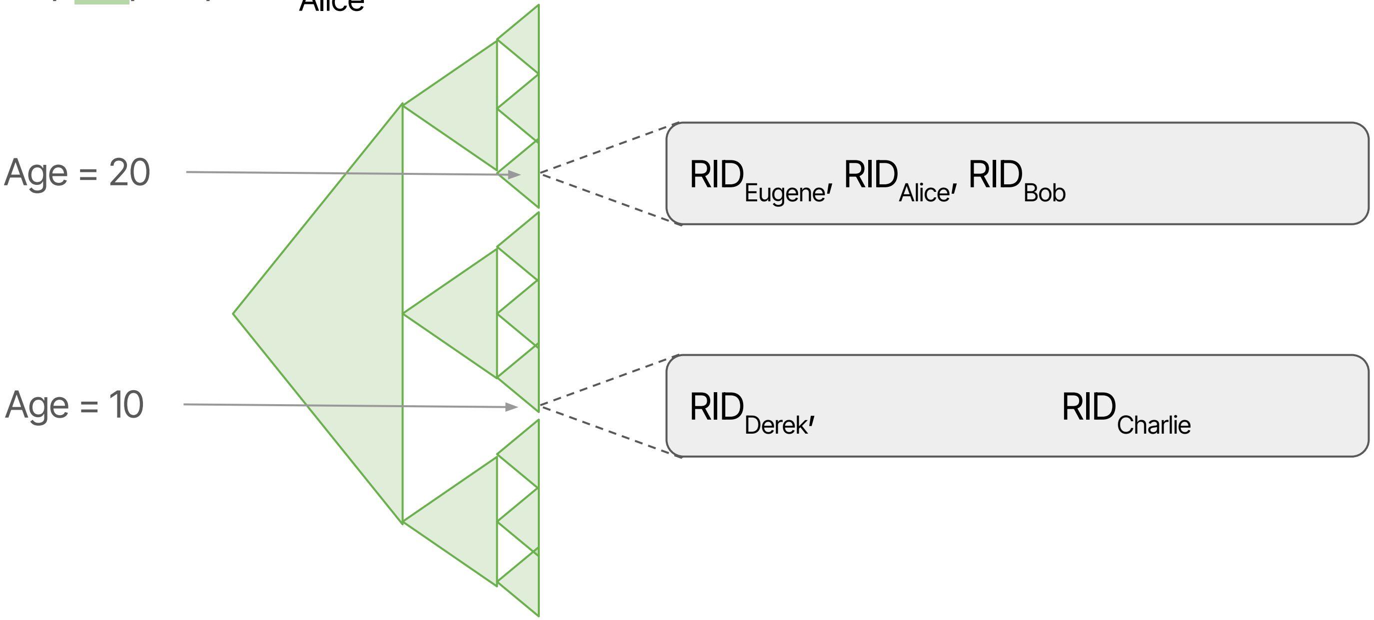
Age = 10

$\text{RID}_{\text{Eugene}}$, $\text{RID}_{\text{Alice}}$, RID_{Bob}

$\text{RID}_{\text{Derek}}$, ~~$\text{RID}_{\text{Alice}}$~~, $\text{RID}_{\text{Charlie}}$

Indexing - Find & Alter

Update([..., 20, ...], $\text{RID}_{\text{Alice}}$)



Age = 20

Age = 10

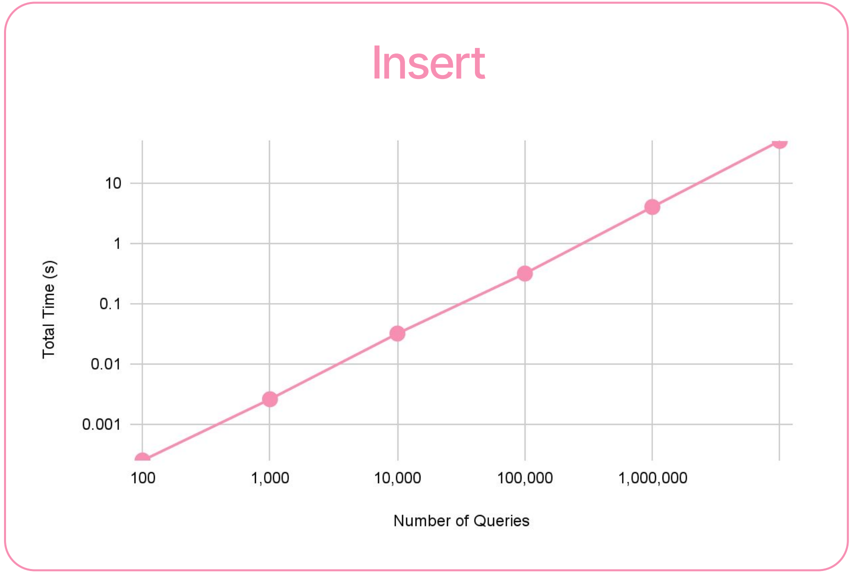
$\text{RID}_{\text{Eugene}}$, $\text{RID}_{\text{Alice}}$, RID_{Bob}

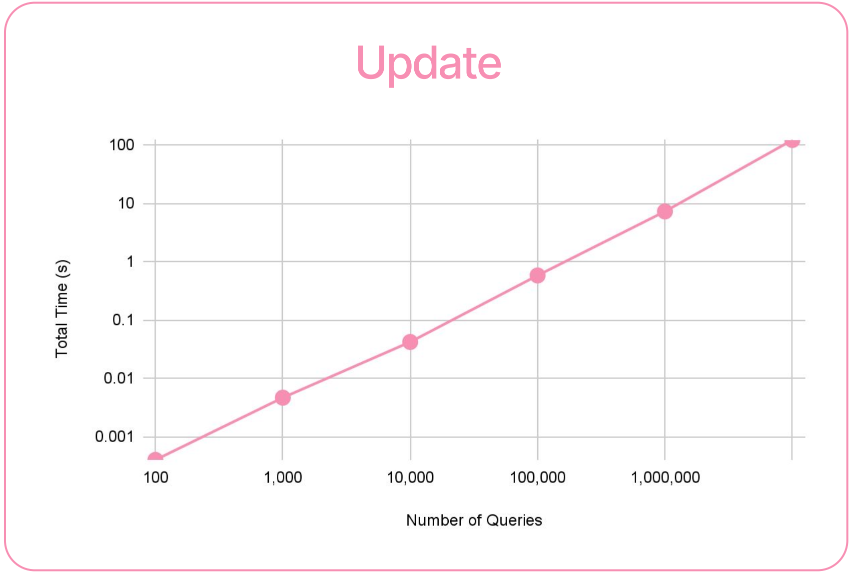
$\text{RID}_{\text{Derek}}$, $\text{RID}_{\text{Charlie}}$

This could be parallelizable. What about other operations?

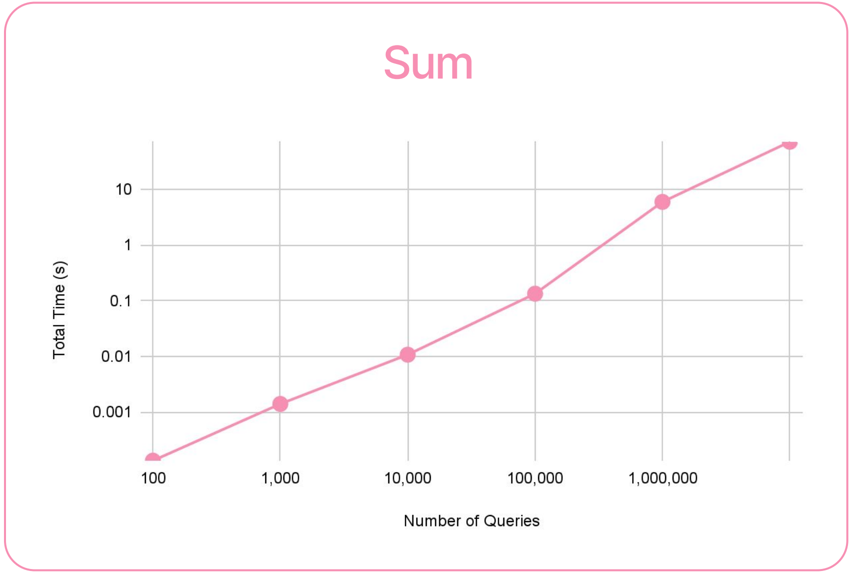
Evaluation

Query Performance

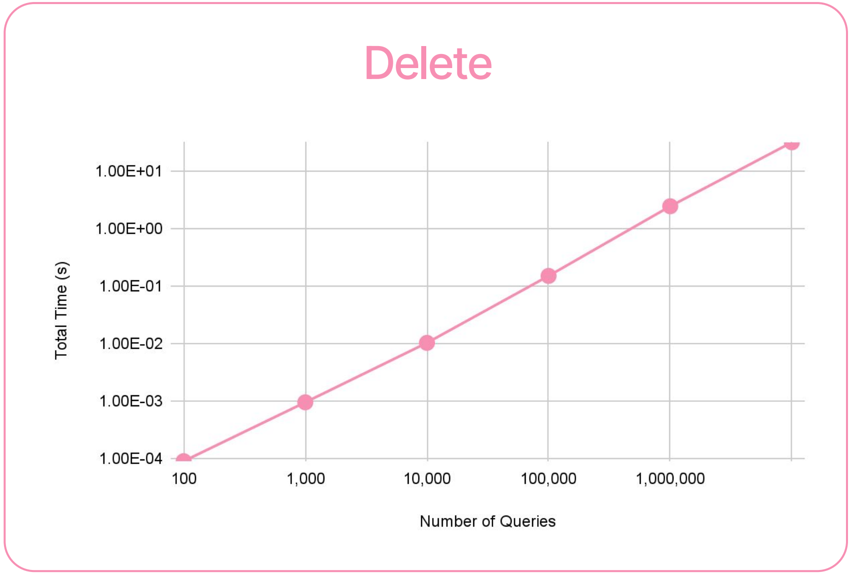

Insert


Update

Query Performance

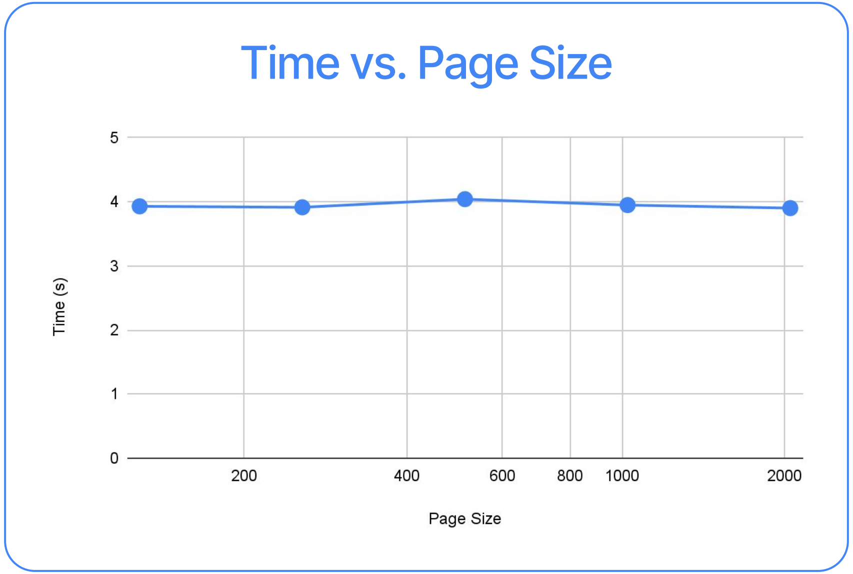


Query Performance



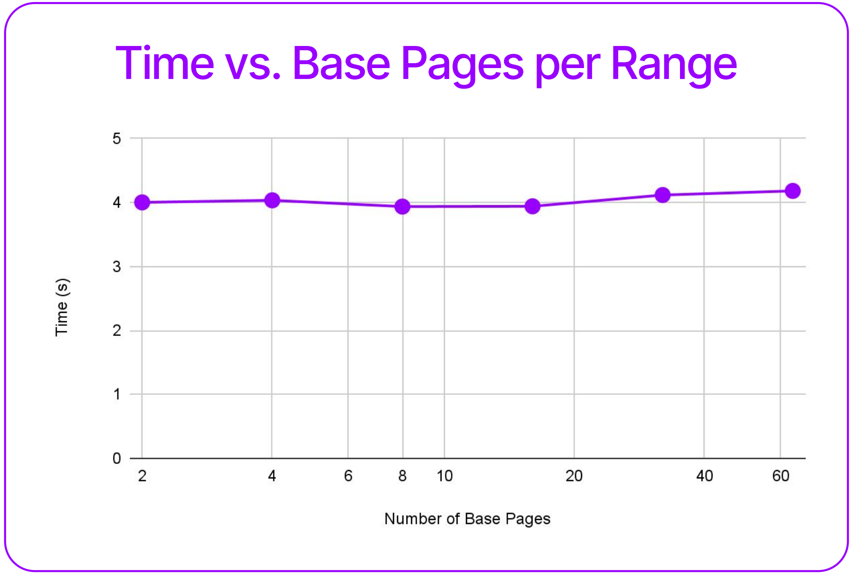
Conclusion - Regardless of database size, queries take the same amount of time!

Physical Page Size



Ran 1,000,000 insert queries using several different page sizes. **There was no *significant* difference in the total runtime.**

Base Pages per Range



Ran 1,000,000 insert queries using several different base page counts. **There was no *significant* difference in the total runtime.**

Demonstration