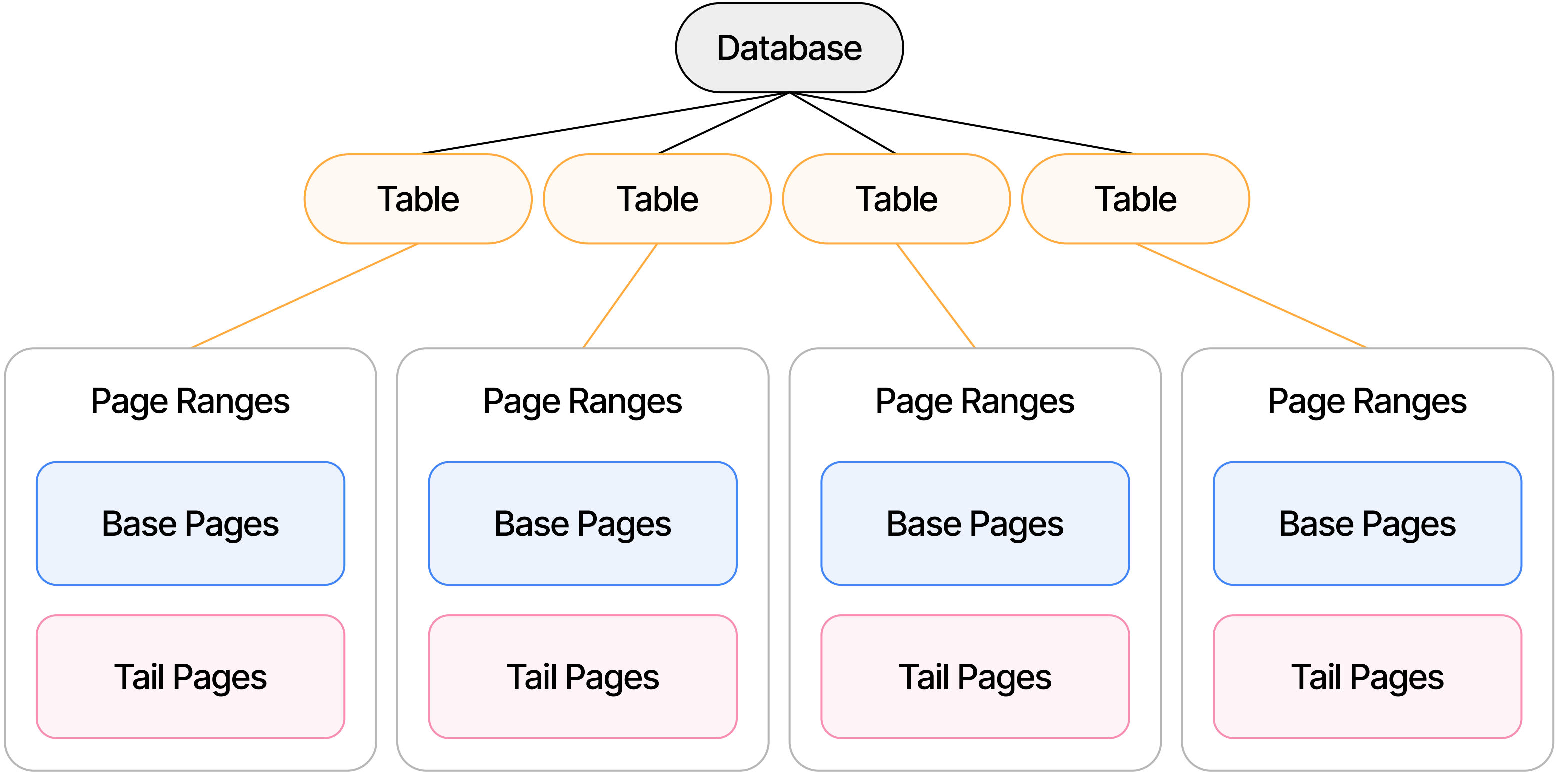
CowabungaDB

George Berdovskiy
Nate Buttke
Kevin Bao
Keyur Parikh
Marcin Wróblewski

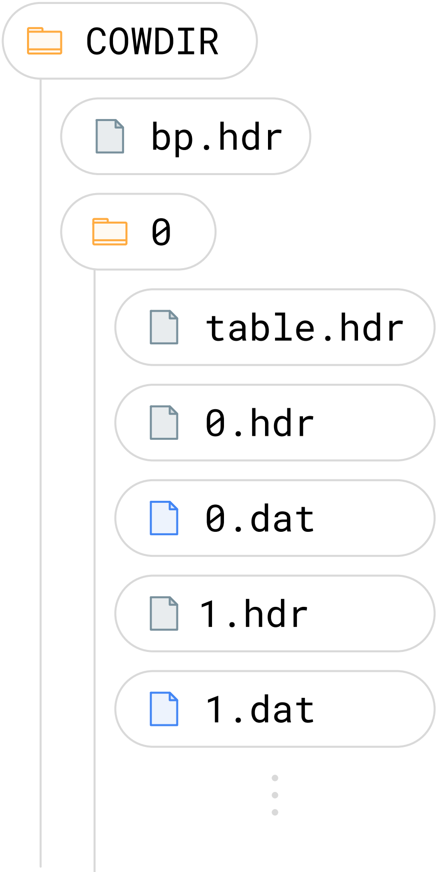
Previous Milestone Review

Overall Design

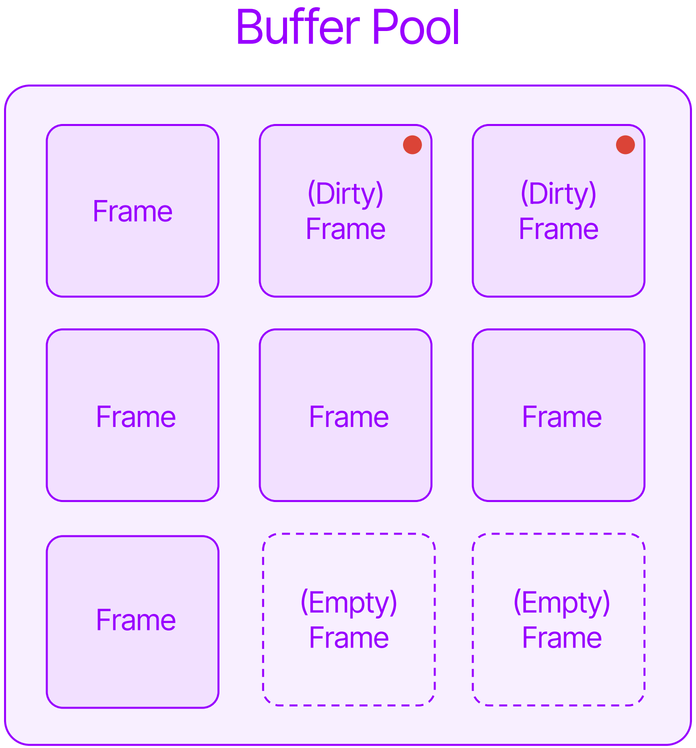


Persistence

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



Buffer Pool


Buffer Pool
Frame
(Dirty)
Frame
(Dirty)
Frame
Frame
Frame
Frame
Frame
(Empty)
Frame
(Empty)
Frame

Transactions

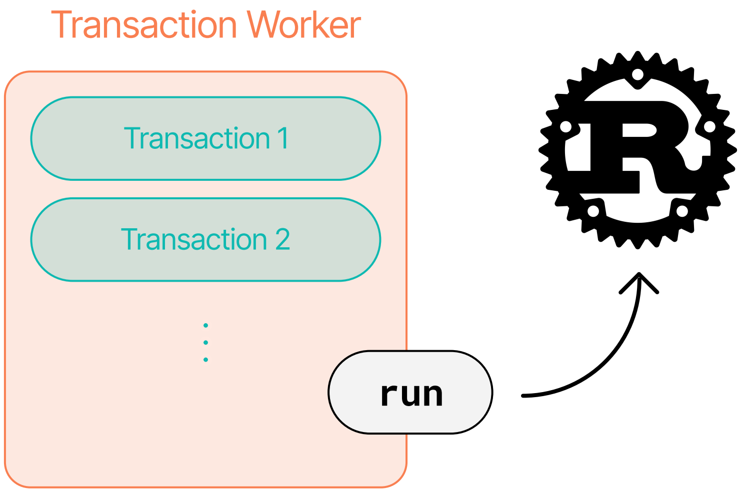
Transaction Worker, Transactions

Transaction Worker

- Contains an array of `Transactions`
- When `run` is called, a new thread is spawned to run the transaction in Rust

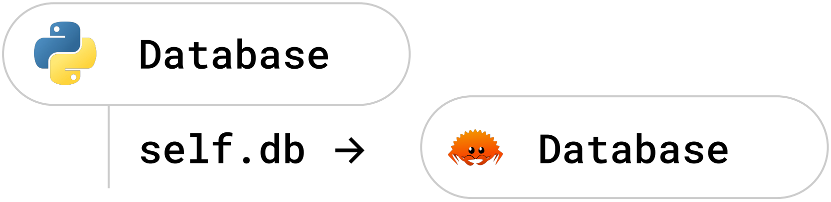
Transaction

- Contains an array of **queries** and their **arguments**



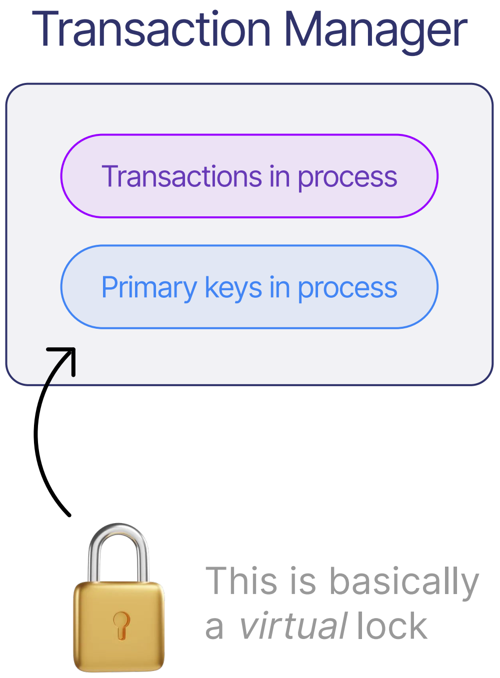
Database Rewrite

- Python and Rust - incompatible handling of ownership and memory → FFI issues
- Resolved by rewriting Python **`Database`** class in Rust
- Every database has its own buffer pool and transaction manager *(more on the next slide)*



Transaction Manager

- Every transaction must lock the manager before beginning → transactions are *started* sequentially but run *concurrently*
- Notable fields...
  - **`transactions_in_process`** - map from transaction IDs to the primary keys they touch
  - **`pkeys_in_process`** - map from primary key to the *effect* a running transaction may have on it
    - Shared by all transactions!
- Transactions and associated primary keys are removed only when completed (strict 2PL)

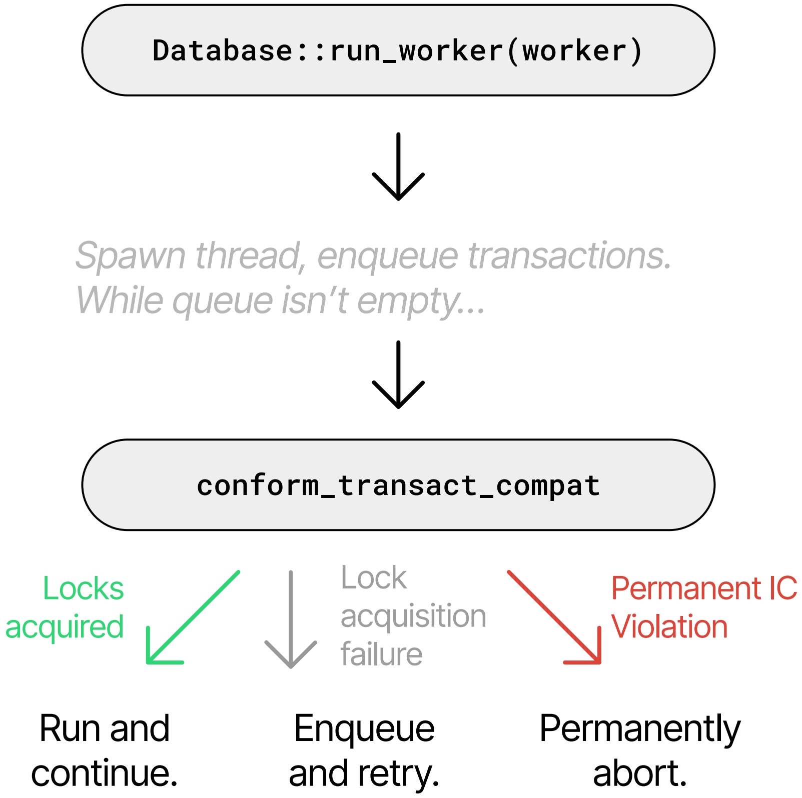


Concurrency

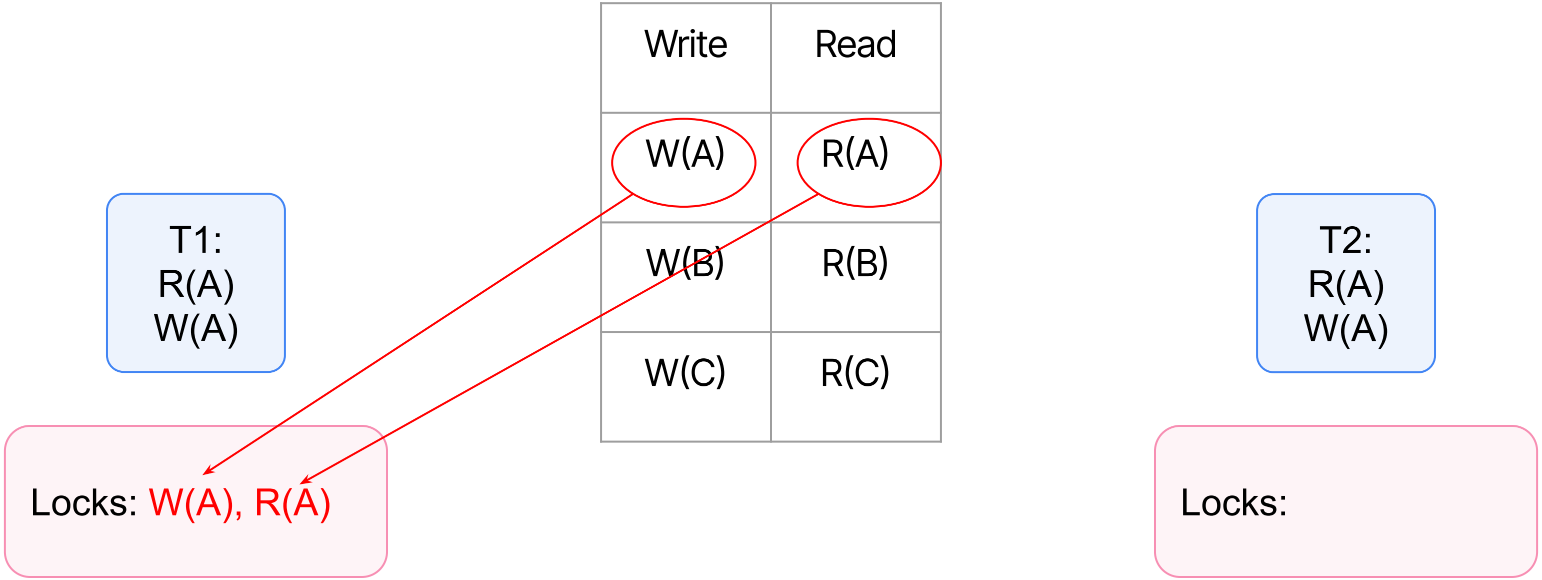
Strict 2PL + No Wait

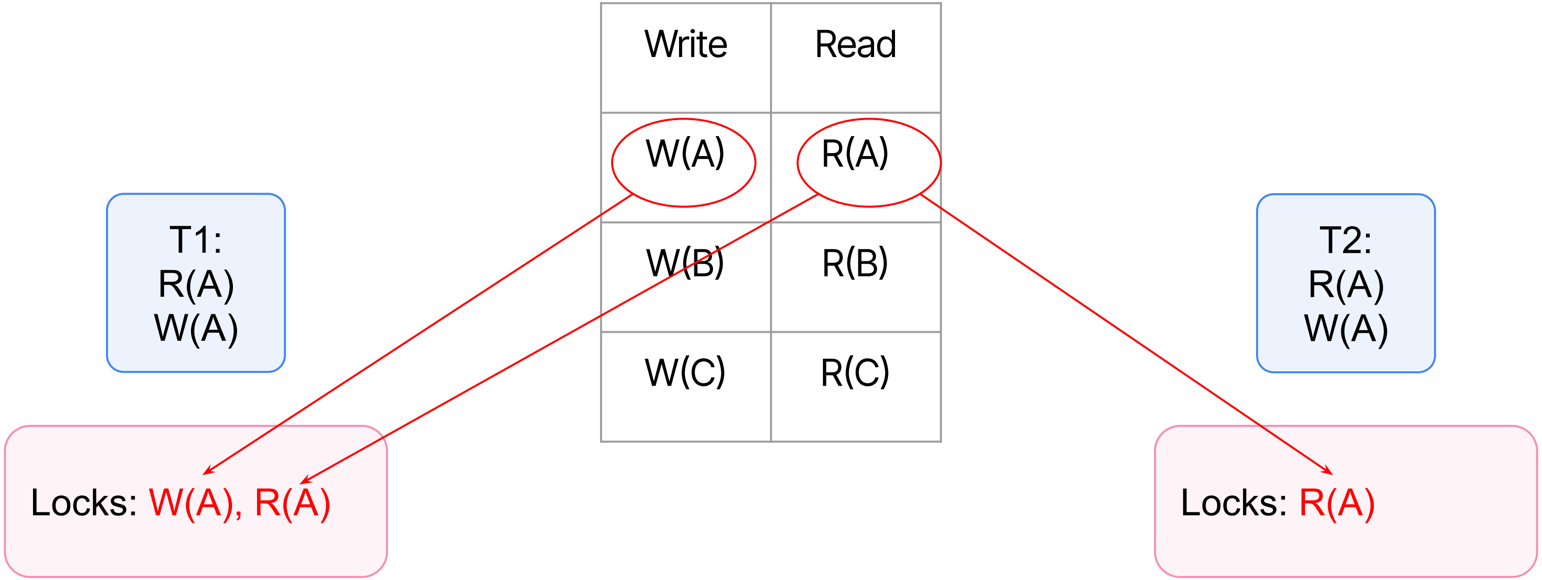
Overview and Implementation

- Before transaction executes, request sent to transaction manager → attempts to **gather locks** on all participating records
- If all locks can be gathered and "compatibility checks" pass (*conflict serializability*), transaction executes
- Otherwise, the **transaction is aborted**
  - Retries if due to lock acquisition failure or integrity constraint violation due to *other* transactions
  - Permanently aborts if it's due to violated integrity constraint within *this* transaction

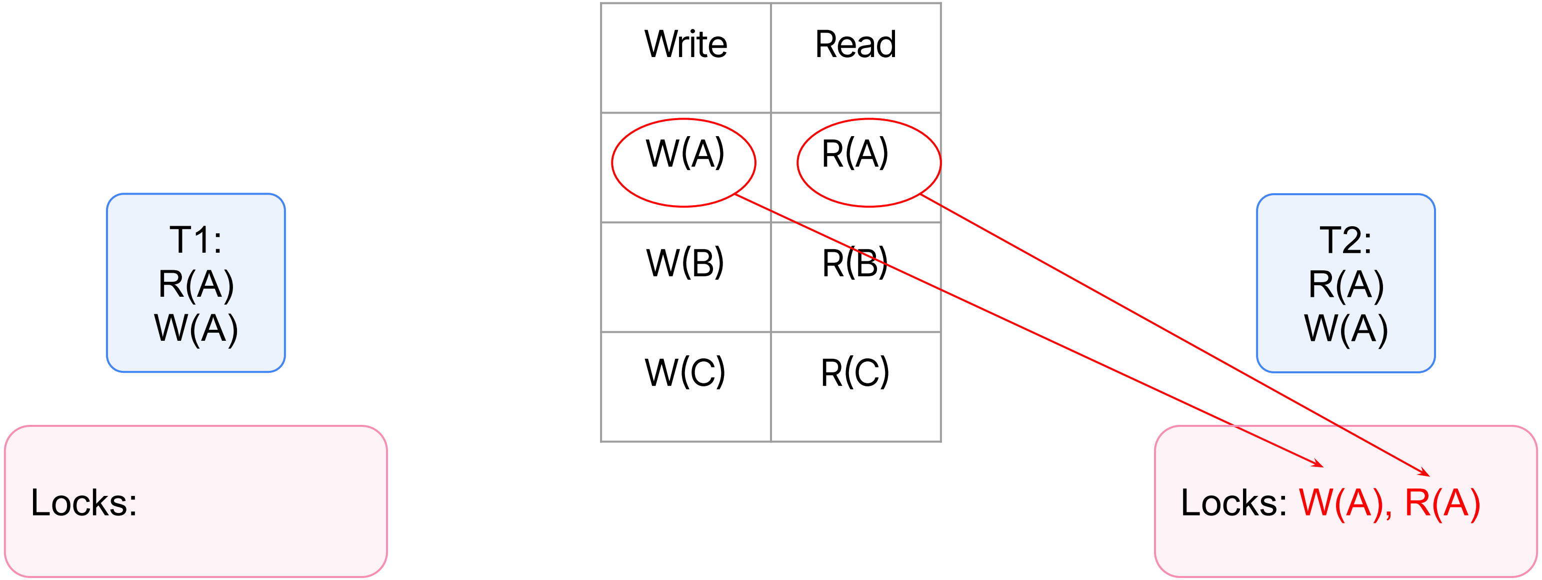


Transactions T1 and T2 are run





W(A) Lock cannot be obtained, abort T2 and retry later



T1 executes and locks are released

T1:
R(A)
W(A)

Locks:

Write	Read
W(A)	R(A)
W(B)	R(B)
W(C)	R(C)

T2:
R(A)
W(A)

Locks:

T2 executes and locks are released

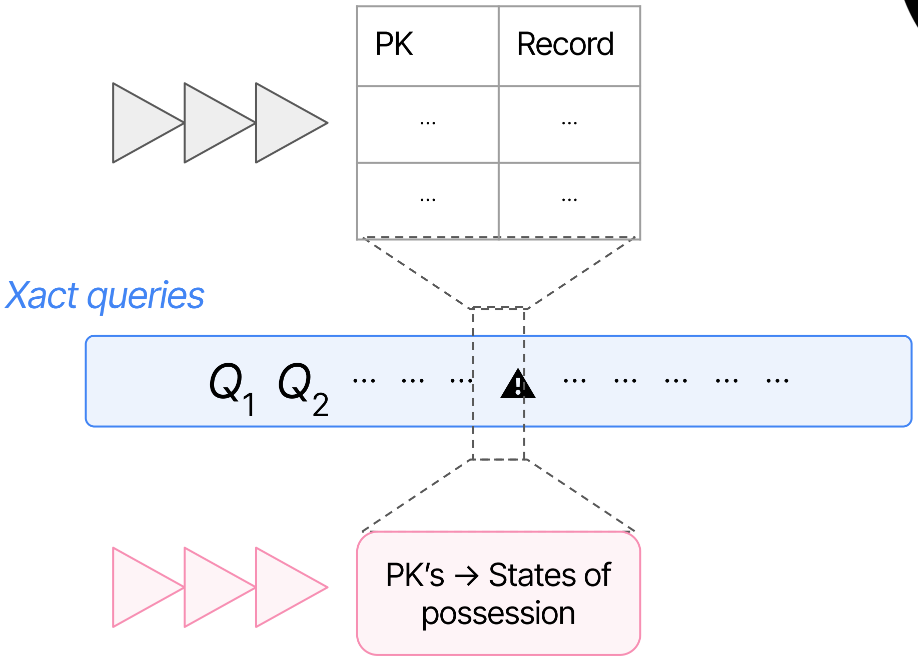
Correctness / Integrity constraints

Overview and Implementation

1. We have a set of possibly-participating primary keys held by existing, committed records

2. Any insert, update and delete queries along with any participating primary keys are checked in order by a bookkeeping algorithm

- This allows us to discover integrity constraint violations before any queries are executed (ensuring atomicity too)
- Also discovers operations on non-existent records



T1:
Update(Record 1's PK to 3)
Update(Record 2's PK to 3)

From Indexer

PK	Record
1	1
2	2

Hash

In the algorithm, an update is treated as a delete followed by an insertion

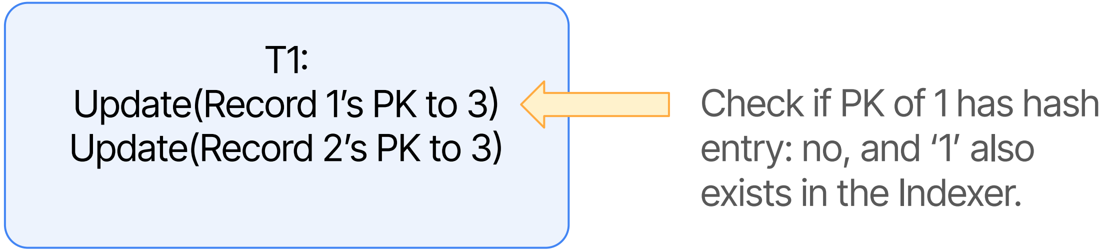
T1:
Update(Record 1's PK to 3)
Update(Record 2's PK to 3)

Check if PK of 1 has hash entry:

From Indexer

PK	Record
1	1
2	2

Hash



From Indexer

PK	Record
1	1
2	2

Hash

T1:
Update(Record 1's PK to 3)
Update(Record 2's PK to 3)

Add new participating
PK (1) to hash (delete)

From Indexer

PK	Record
1	1
2	2

Hash

1 → Not held

T1:
Update(Record 1's PK to 3)
Update(Record 2's PK to 3)

Add new participating PK (1) to hash (delete)
Check if PK of 3 has hash entry:

From Indexer

PK	Record
1	1
2	2

Hash

1 → Not held

T1:
Update(Record 1's PK to 3)
Update(Record 2's PK to 3)

Add new participating PK (1) to hash (delete)
Check if PK of 3 has hash entry: no, and '3' also does *not* exist in the Indexer (insert)

From Indexer

PK	Record
1	1
2	2

Hash

1 → Not held
3 → Held

T1:
Update(Record 1's PK to 3)
Update(Record 2's PK to 3)

Add new participating
PK (2) to hash (delete)

From Indexer

PK	Record
1	1
2	2

Hash

1 → Not held
3 → Held
2 → Not held

T1:
Update(Record 1's PK to 3)
Update(Record 2's PK to 3)

Add new participating PK (2) to hash (delete)
Check if PK of 3 has hash entry:

From Indexer

PK	Record
1	1
2	2

Hash

1 → Not held
3 → Held
2 → Not held

T1:
Update(Record 1's PK to 3)
Update(Record 2's PK to 3)

Add new participating PK (2) to hash (delete)
Check if PK of 3 has hash entry: yes, and a PK of 3 is held (insert)

From Indexer

PK	Record
1	1
2	2

Hash

1 → Not held
3 → Held ⚠
2 → Not held

For insertions we may also find a conflict in the Indexer

- *Don't abort permanently*, this was triggered only by incompatibility with current records (no self-incompatibility so far is known)

For deletions, we might either notice

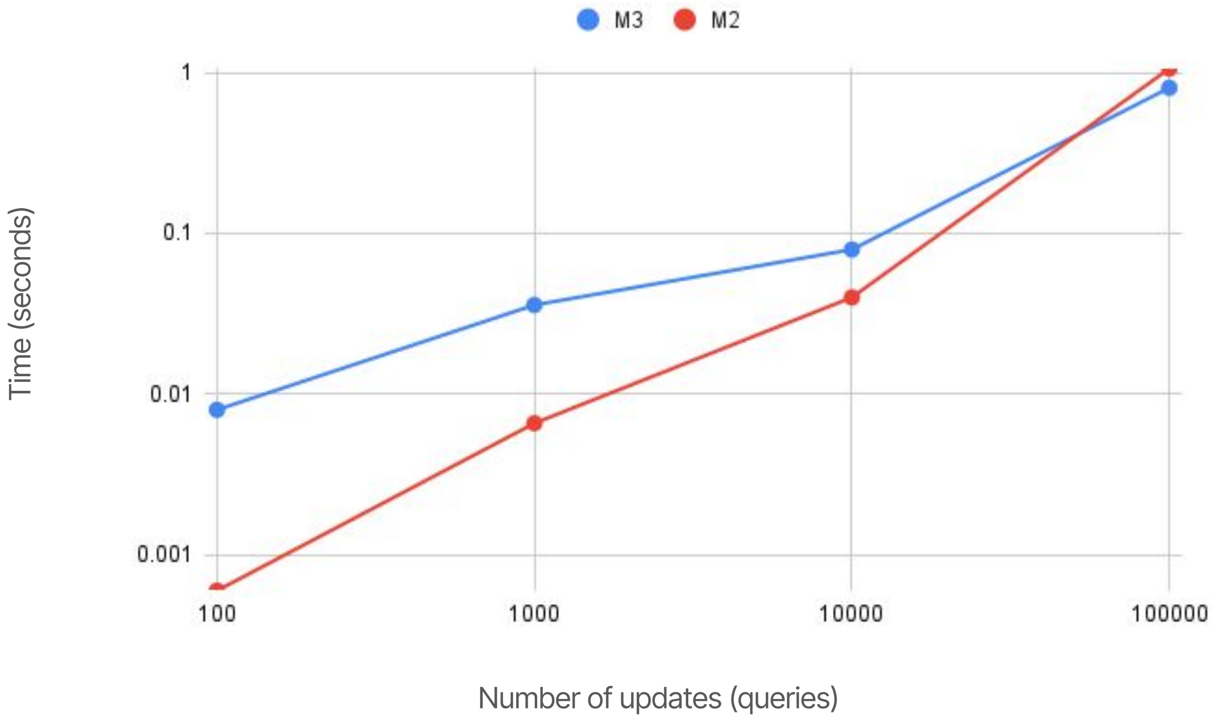
- 'Nonexistent' in the hash (self-incompatible double deletion)
- The PK is missing from the Indexer (deletion of never-existing record, not self-incompatible)

No matter the starting records, T1 is self-incompatible and so is permanently aborted.

Evaluation

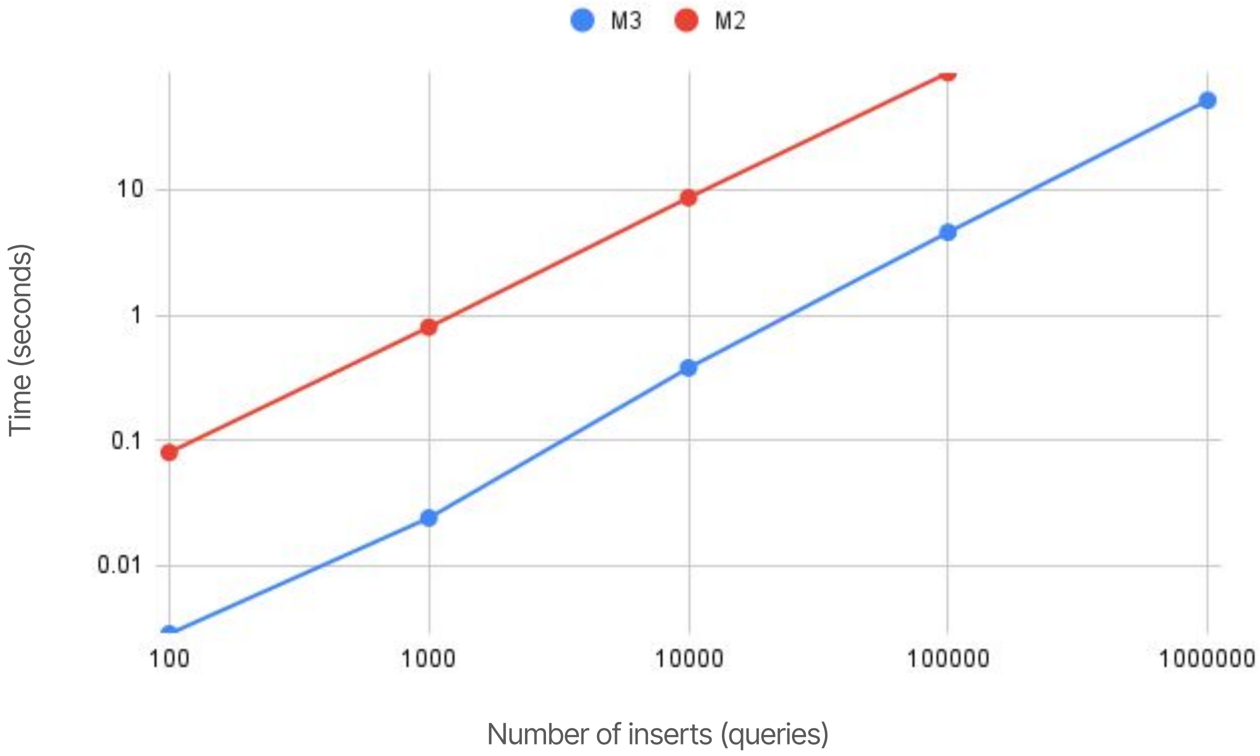
Comparing against Milestone 2

Selects



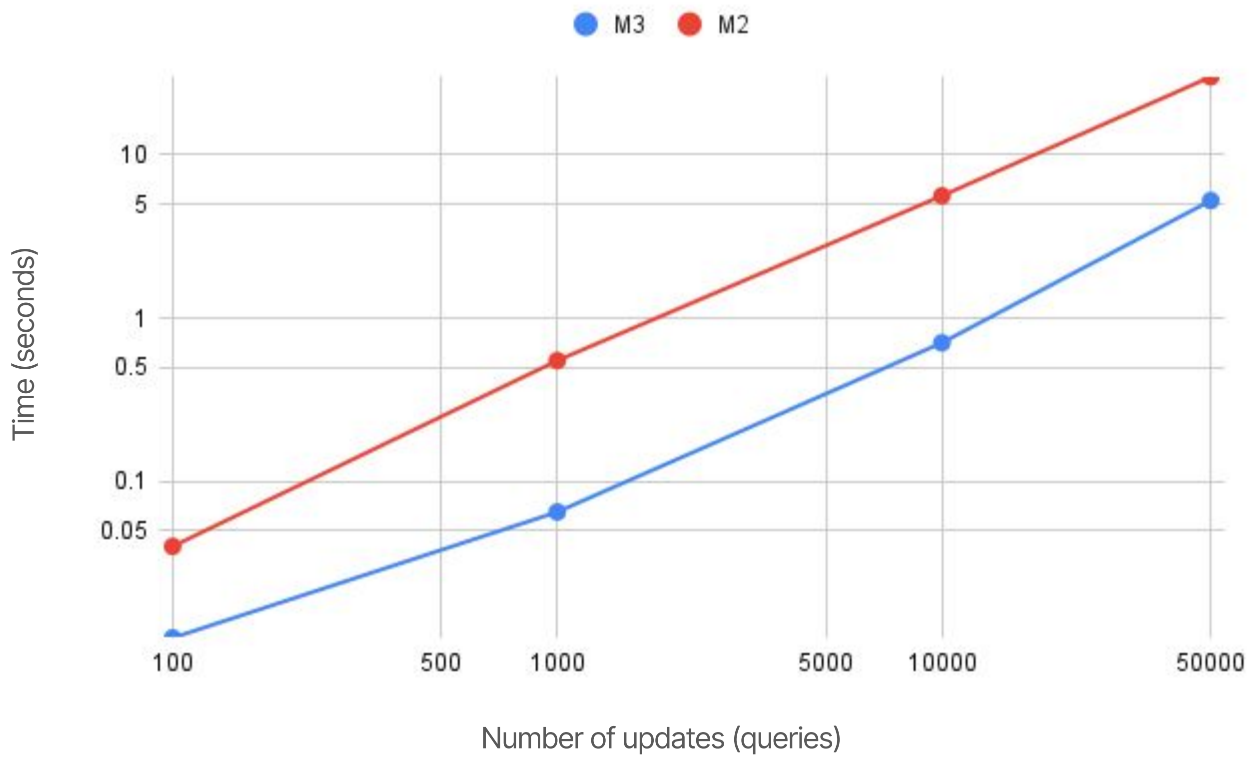
Comparing against Milestone 2

Inserts ~ 15-20x boost



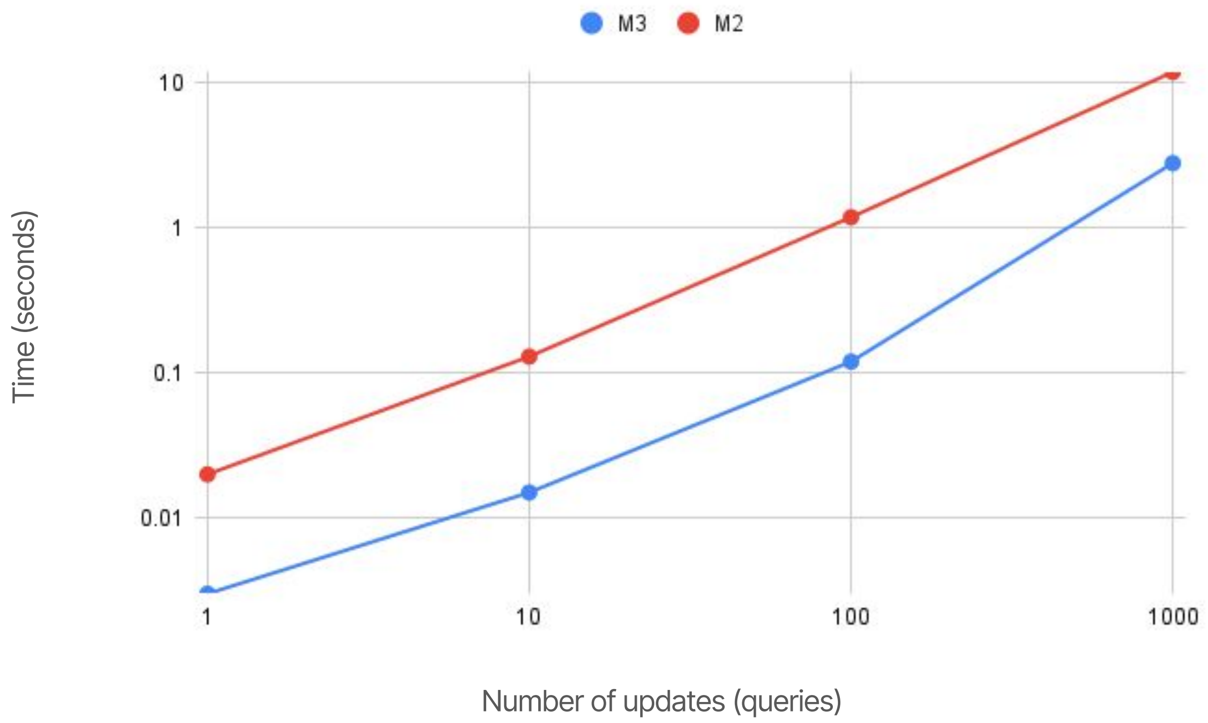
Comparing against Milestone 2

Updates ~ 4-6x boost



Comparing against Milestone 2

Deletes ~ 5-8x boost



Conclusion

Looking Back...

- Using Rust was an excellent choice
  - *Very* fast and memory safe... not a single segmentation fault!
  - Only bugs were *logic* bugs
  - Types → easier to understand
- Teamwork is important!
  - Met in person 2 - 3 times per week
  - Food fuels productivity 🍕
- Understanding is critical for implementation
  - Some features worked almost on the first try due to hours of thorough discussion
- Around 115+ commits, 20 closed PRs 🎉

Demonstration