# Eventual Durability

Tejasvi Kashi
Kenneth Salem
University of Waterloo
Waterloo, Ontario, Canada
tejasvi.kashi@uwaterloo.ca,ken.salem@uwaterloo.ca

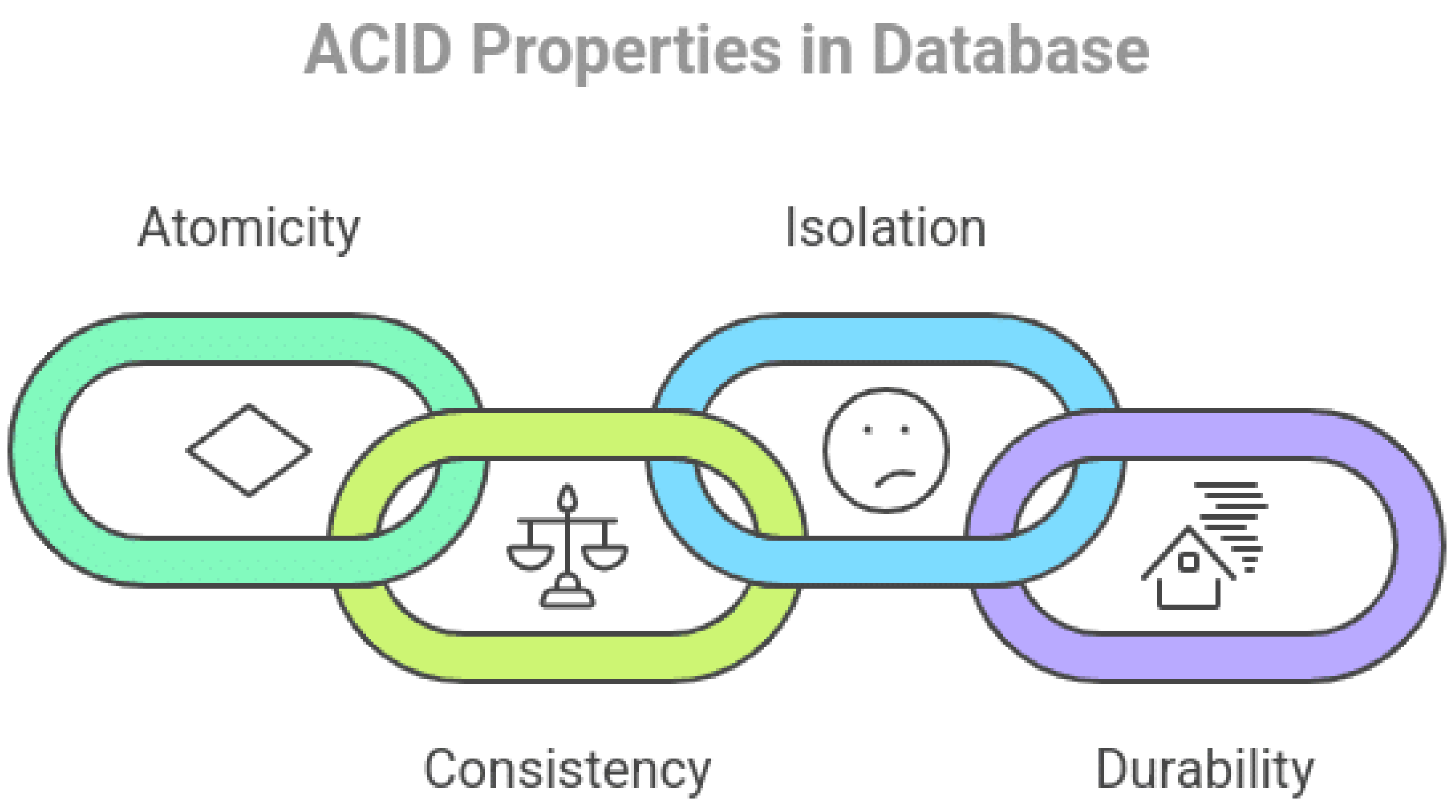
Jaemyung Kim
Khuzaima Daudjee
University of Waterloo
Waterloo, Ontario, Canada
jaekim.uw@gmail.com,kdaudjee@uwaterloo.ca

**Team Members**

| | |
|---|---|
| 200036T | Ampavila I.A. |
| 200332X | Kurukulasooriya K.V.R. |
| 200555H | Samrakoon R.S.A |
| 200623P | Subodha K.R.A |
| 200727M | Wijesinghe U.M. |

# Overview

# Background of the Study




- A **Transaction** is a sequence of database operations that are executed as a single unit of work, ensuring consistency and integrity.

- Example: Transferring money between bank accounts. If the debit is successful but the credit fails, the whole transaction is rolled back to maintain data integrity

# Problem Statement

(a) Classical Transaction Timeline

Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure.

Durability is slow, limits performance. The delay caused by waiting for durability

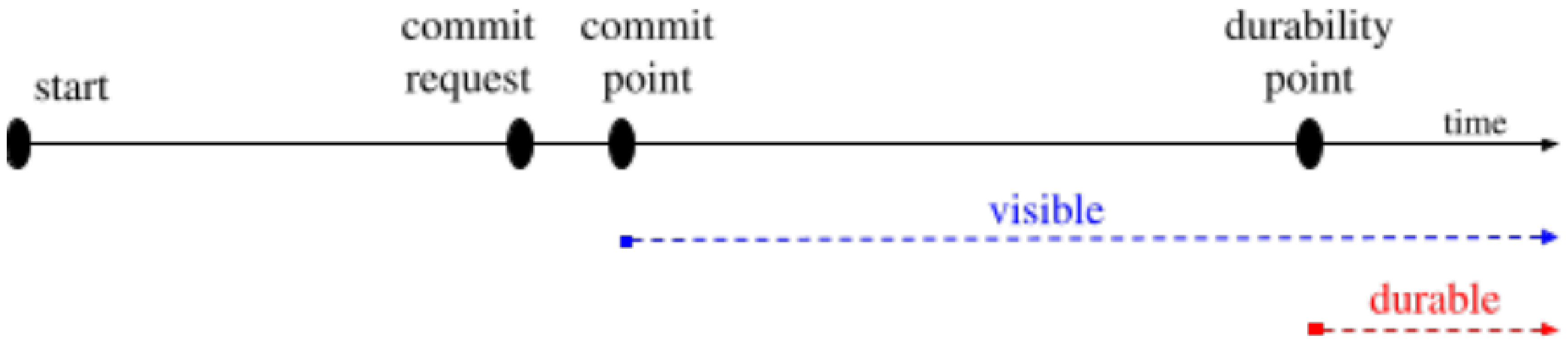
**How can systems ensure strong durability without sacrificing performance and scalability?**

# Framework

**Key Idea:**
**Decouple commit from durability.**

01 Commit first

02 Make Durable Later




**(b) Eventually Durable Transaction Timeline**

Need to consider 2 Types of Transactions :

**Fast Transactions**

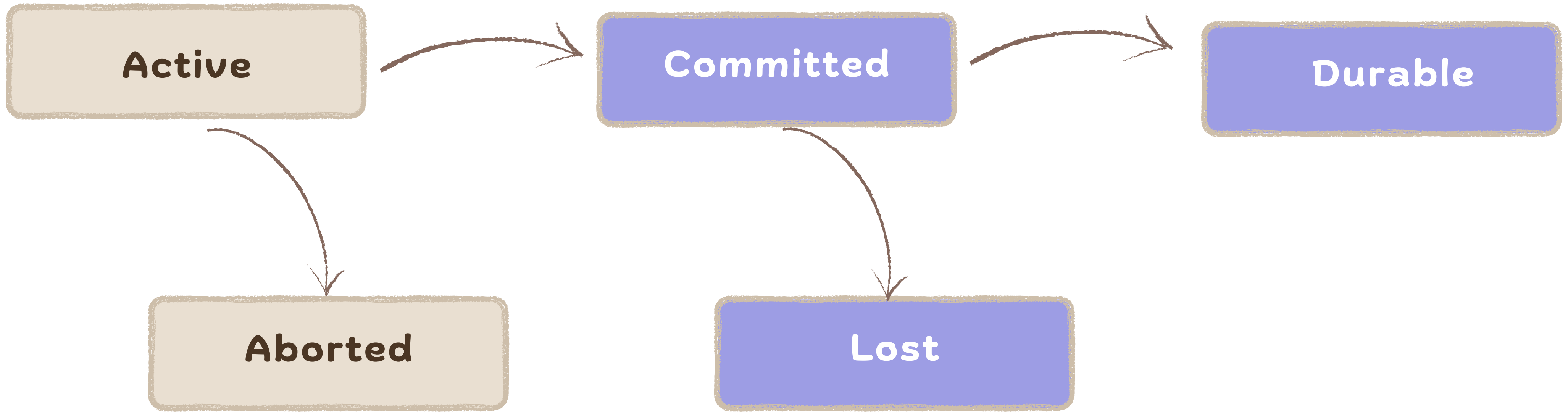
- Commit quickly
- Durability is not guaranteed.

**Safe Transactions**

- Commit only when durability is ensured

# Framework

## Transactions move through three states:




- **Committed:**

Changes are logged and visible, but not yet fully stored on disk.

- **Durable:**

Changes are safely written to disk.

- **Lost:**

Otherwise, they may be lost on failure.

# *Our Implementation*

We attempted to implement eventual durability using **two** main approaches.

01

Our **MiniSQL** Implementation

02

MySQL

# MiniSQL Implementation

```
├── src
│   └── main
│       ├── java
│       │   └── edu
│       │       └── mora
│       │           └── db
│       │               ├── catalog
│       │               │   └── Catalog.java
│       │               ├── engine
│       │               │   └── CLI.java
│       │               ├── executor
│       │               │   └── SimpleExecutor.java
│       │               ├── parser
│       │               │   └── SQLParser.java
│       │               ├── sql
│       │               │   ├── CreateTableStatement.java
│       │               │   ├── DeleteStatement.java
│       │               │   ├── InsertStatement.java
│       │               │   ├── SelectStatement.java
│       │               │   ├── Statement.java
│       │               │   └── UpdateStatement.java
│       │               ├── storage
│       │               │   ├── BufferPool.java
│       │               │   ├── DiskManager.java
│       │               │   ├── Page.java
│       │               │   ├── RecordId.java
│       │               │   ├── TransactionManager.java
│       │               │   └── WALManager.java
│       │               └── table
│       │                   ├── Schema.java
│       │                   ├── Table.java
│       │                   └── Tuple.java
```

MiniSQL File Structure

```
MiniSQL> select * from users
+----+-------+
| id | name  |
+----+-------+
| 1  | Udara |
| 2  | Inuka |
+----+-------+
MiniSQL> insert into users
```

MiniSQL Sample Database command execution

# MiniSQL Implementation

**SECOND APPROACH :** Built a SQL database from scratch (in Java)

The main components of the database system:

### Engine (CLI)

- Serves as the **interactive shell** for MiniSQL
- Initializes subsystems and manages recovery
- Handles user input and ensures safe, durable shutdown

### Storage

- Uses Pages and BufferPool to manage table data
- **LRU strategy** for in-memory page management
- Supports transactions and recovery via **Write-Ahead Logging (WAL)**

### Supported Operations

- SELECT
- CREATE
- INSERT
- UPDATE
- DELETE

# Eventual Durability in MiniSQL

MiniSQL detects fast transactions via a special SQL hint: /*+ FAST */

**Write-Ahead Log Flushing**

- Fast transactions don't flush WAL immediately at commit
- A separate thread handles delayed flushing
- Simulates I/O-heavy scenarios targeted by fast transactions

```
MiniSQL> /*+ FAST */ INSERT INTO users VALUES (3, 'Radith')
```

# Benchmarks

To benchmark the performance of our implementation we followed the below steps,

1. Start an in-process MiniSQL engine and pre-load a 256-row key-value table.
2. Run a 5-second write-intensive workload with 8 parallel threads; each thread repeatedly executing UPDATE kv SET v = v + 1 … on random keys.
3. Execute the workload twice: once with /*+ FAST */ (durability deferred) and once with /*+ SAFE */ (fsync + page-flush on every commit).
4. Collect the number of successfully committed transactions and the nanoseconds spent per commit to compute throughput (tx/s) and average latency (µs).

Below are the type of results we get when we ran the benchmark, validating the claim the fast transactions improve perofrmance in certain scenarios.

```
FAST commit → 83,692 tx  |  throughput 16738.4 tx/s  |  avg 477.5 µs
SAFE commit → 62,288 tx  |  throughput 12457.6 tx/s  |  avg 641.9 µs
```

# MySQL Implementation

🔧 **Global server variable added:**

- eventual_durability_mode (ON / OFF)
- Configurable at startup via --eventual_durability_mode
- Implemented in sys_vars.cc and registered via sys_var API

✨ **Commit path conditionally modified in trx_commit_low():**

**Fast (non-durable) path:**

- Calls log_write_up_to(commit_lsn, false)
- WAL written to OS buffer only (no flush)

**Safe (durable) path:**

- Calls log_write_up_to(commit_lsn, true) (default behavior)

📊 **Runtime counters added:**

- Fast_transactions_committed
- Safe_transactions_committed
- Exposed via SHOW STATUS LIKE '...';
- Registered with add_status_vars(...) in mysqld.cc

# MySQL Database - Safe Trx

```
mysql> SET GLOBAL eventual_durability_mode = OFF;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO ed_test VALUES (1000, 'safe'); SHOW ENGINE INNODB STATUS;
Query OK, 1 row affected (0.01 sec)
```

```
---
LOG
---
Log sequence number          21019612
Log buffer assigned up to    21019612
Log buffer completed up to   21019612
Log written up to            21019604
Log flushed up to            21019604
Added dirty pages up to      21019612
Pages flushed up to          21019324
Last checkpoint at           21019324
Log minimum file id is       6
Log maximum file id is       6
27 log i/o's done, 1.35 log i/o's/second
```

# MySQL Database - Fast Trx

```
mysql> SET GLOBAL eventual_durability_mode = ON;
Query OK, 0 rows affected (0.00 sec)


mysql> INSERT INTO ed_test VALUES (501, 'fast'); SHOW ENGINE INNODB STATUS;
Query OK, 1 row affected (0.00 sec)
```

```
---
LOG
---
Log sequence number          21022222
Log buffer assigned up to    21022222
Log buffer completed up to   21022222
Log written up to            21022198
Log flushed up to            21021111
Added dirty pages up to      21022222
Pages flushed up to          21020440
Last checkpoint at           21020440
Log minimum file id is       6
Log maximum file id is       6
```

# Benchmarks

To benchmark the performance of our implementation we followed the below steps,

1. Connect via the Unix socket (./mysql.sock) and drop/recreate the ed_bench database.
2. Create a table bench_table(id INT PRIMARY KEY, val VARCHAR(100)).
3. Enable eventual durability using: SET GLOBAL eventual_durability_mode = ON; and run 5 000 sequential INSERT statements.
4. Disable eventual durability using: SET GLOBAL eventual_durability_mode = OFF; and run 5 000 sequential INSERT statements.
5. Compute throughput (txns/sec) = N / (duration / 1000) and report total duration in ms.

Below are the type of results we get when we ran the benchmark, validating the claim the fast transactions improve perofrmance in certain scenarios.

```
⚡ Running 5000 fast transactions...
🔒 Running 5000 safe transactions...

✅ Benchmark Results:
Fast commit throughput : 192.152 txns/sec (26021 ms)
Safe commit throughput : 177.16 txns/sec (28223 ms)
```

# Other Attempted Changes

1. If transaction T2 depends on T1 it should not be durable before T1
2. Check dirty pages in the buffer pool for durability. If the corresponding transactions were not durable the flushing of these dirty pages needed to be delayed.

# *DEMO*

# Thank You

MySQL Edited Source Code: https://github.com/inuka-00/mysql-server/tree/ed-mysql-8.0.36
MiniSQL Source Code: https://github.com/RadCod3/customdb