ElVoidDB

1 project goal

ElVoidDB is a tiny SQL-like Database written in c++17 from Scratch. This Database is mainly inspired from Class CSE 562 Database Systems and Concepts used from TacoDB. My Goal of this project is try and build real working database from scratch. I built it step by step as I have learnt the concepts in Class and the Structure I have seen in TacoDB.

- **CLI + parser** – read a line, build command objects.
- **In-memory tables** – quick testing while disk layer matures.
- **Disk storage** – 4 KB pages, slotted-page layout.
- **Buffer pool** – LRU cache so pages stay in RAM.
- **Thread pool** – background tasks (flush, heavy I/O).
- **Background flusher** – writes dirty pages every 2 s.

Everything is version-controlled in Git (GitHub repo **ElVoidDB**).

2 Folder structure

```
ElVoidDB/
|– include/          <- public headers
|   |– Exceptions.hpp       basic error classes
|   |– Page.hpp             fixed-size 4 KB page + record API
|   |– Storage.hpp          BlockFile + TableFile +
FileManager
|   |– BufferPool.hpp       LRU cache + global gBufPool
|   |– ThreadPool.hpp       worker pool + global
gThreadPool
|   |– BackgroundFlush.hpp 2-second timer -> flush
job
|   |– Parser.hpp           SQL -> command objects
|   \ -- Commands.hpp        CREATE / INSERT / SELECT
classes
|– src/              <- implementation
|   \--  (one .cpp per header)
|– CMakeLists.txt     <- build script
\– build/            <- out-of-tree build (ignored by Git)
```

3 How the code flows

3.1 Main entry

src/main.cpp

```
BackgroundFlusher bg(2000);   // flush every 2 s
bg.start();

while (getline(cin, line)) {
   auto cmd = Parser::parse(line);
   cmd->execute();          // polymorphic call
}

bg.stop();
gBufPool.flushAll();         // final safety
```

3.2 Parser -> command objects

Simple token logic; semicolon is optional for now.

```
CREATE TABLE name (col1,col2)
INSERT INTO name VALUES (v1,v2)
SELECT * FROM name
EXIT / QUIT
```

Parser::parse() returns one of:

class	job
CreateTableCmd	make new .tbl file, add blank table to RAM.
InsertCmd	auto-load table if missing, check column count, push row, call TableFile::appendRow().
SelectCmd	auto-load table if missing, print header + all rows.

3.3 Disk Layer (Storage)

Type	Purpose	Depends on
Page	4 KB buffer, insertRecord() & forEachRecord().	-
BlockFile	read/write a page **through the buffer pool**.	gBufPool
TableFile	one data file per table. Page 0 = text header cols:id,name,. Pages 1..N = rows.	BlockFile
FileManager	map <table->TableFile*> so we open each file once.	TableFile

3.4 Buffer pool

LRU cache (default 64 frames)

```
Page&  gBufPool.get(path, pageNo);   // pins + returns
markDirty(path,pageNo);            // caller changed
frame
unpin(path,pageNo);                // pin-count--
flushAll();                    // write dirty frames
```

If every frame is pinned, get() throws **StorageError**.

3.5 Thread pool + background flush

One global worker pool.

```
// every 2 s:
gThreadPool.submit([]{ gBufPool.flushAll(); });
```

BufferPool::unpin() also submits an async flush when pin==0 && dirty==true.

4 Problems we solved (chronological)

Issue	Fix
Rows lost on restart.	Implement TableFile::appendRow() + loadAllRows().
Metadata & data mixed in page 0.	Never write rows to page 0; allocate page 1 first.
Seg-fault on first SELECT.	ensureLoaded() loads rows once and caches in RAM.
Double-loading caused duplicate “1Alice21 2Bob30”.	Load rows only when rows vector is empty.
Infinite recursion (BufferPool -> BlockFile -> BufferPool).	Use raw fstream inside BufferPool helpers.
Crash reading old corrupt rows.	Bounds-checked deserializeRow() -> bad rows skipped.
Column mismatch on VALUES((no space).	Parser patch (strip “VALUES(” token).
Git push refused plain password.	Switched to PAT / SSH authentication.

5 Build & run

```
mkdir build && cd build
cmake ..
cmake --build . -j
./elvoiddb
```

6 Quick demo

```
ElVoidDB> Create table user (id,name,age);
Table 'user' created.
ElVoidDB> Insert into user VALUES (1,Alice,21);
1 row inserted.
ElVoidDB> Insert into user VALUES (2,Bod,30);
1 row inserted.
ElVoidDB> select * from user;
id      name    age
1       Alice   21
2       Bod     30
ElVoidDB> quit
Bye from ElVoidDB!
```

7 Module dependencies (simplified)

```
main.cpp
 \-- Parser.hpp
     |-- Commands.hpp
     |    |-- Storage.hpp
     |    |    |-- BufferPool.hpp
     |    |    |    |-- Page.hpp
     |    |    |    \-- ThreadPool.hpp
     |    |    \-- Exceptions.hpp
     |    \-- Exceptions.hpp
     \-- Exceptions.hpp
```

8 Future work (next roadmap)

- **Presistence testing -** test the persistence of DB after quit.
- **Write-ahead log (WAL)** - crash-safe commits.
- **WHERE predicates** - simple expression engine.
- **B^+-tree index** - fast lookups.
- **Unit tests** - Catch2 for CI.
- **Replication prototype** - ship WAL to followers.

9 Theory & Function Cheat-Sheet

Module / File	Function / Struct	What it does	Why we need it
Page.hpp	struct PageHeader	Two 16-bit numbers: slotCount and freeOffset.	Keeps the slotted-page layout tiny (4 bytes).
	class Page::insertRecord(std::string)	Packs [len][bytes] into free space, adds a slot entry, returns slot#, **–1** if page full.	Variable-length records without fragmentation.
	forEachRecord(cb)	Calls cb(payload,len) for every record in insertion order.	Table scans & recovery use this.
BufferPool.hpp / .cpp	gBufPool (global)	64-frame LRU cache; frames hold Page.	Avoids disk I/O on every read/write.
	get(path,no)	Pin & return a page; loads from disk if cache miss.	Centralised page fetch.
	markDirty(path,no)	Tells the pool the caller changed the frame.	Flush thread knows what to write back.
	unpin(path,no)	Decrements pin; if pin==0 && dirty, submits async flush to gThreadPool.	Auto-writeback when nobody is using the page.
	flushAll()	Writes all dirty frames to disk.	Final safety on shutdown.
ThreadPool.hpp / .cpp	gThreadPool	Fixed worker threads; runs lambdas.	Background flush; later for WAL fsync, scans.
	submit(f)	Push job into queue, notify one worker.	Fire-and-forget tasks.
BackgroundFlush.hpp	BackgroundFlusher	Spawns a timer thread; every 2 s submits gBufPool.flushAll().	Keeps dirty pages off the critical path.

Storage.hpp / .cpp	BlockFile(path,create)	Opens/creates <table>.tbl.	Low-level file holder.
	readPage(no,Page&)	Gets frame from buffer pool, copies to caller, unpins.	All reads go through cache.
	writePage(no,Page&)	Copies data into frame, marks dirty, unpins.	All writes go through cache.
	TableFile::appendRow(rowVec)	Serialize row → fit into last page or new page, update buffer-pool frame.	Durable INSERT.
	loadAllRows(destVec)	Walks pages ≥1, deserializes each record, pushes into destVec.	Lazy hydration after restart.
	columnList()	Reads page 0, splits "cols:..." into vector.	Loads table schema at runtime.
Commands.hpp / .cpp	ensureLoaded(name)	If gMemDB lacks the table, call columnList() + loadAllRows() once.	Guarantees RAM and disk stay synced.
	CreateTableCmd::execute()	FileManager.createTable, insert blank MemTable in RAM.	Fast table creation plus metadata page.
	InsertCmd::execute()	ensureLoaded -> column-count check -> RAM push -> appendRow.	Durable row insert.
	SelectCmd::execute()	ensureLoaded -> print header + rows.	Reads survive restarts.
Parser.cpp	stripSemicolon()	Drops trailing “;” and whitespace.	MySQL-style input flexibility.
	upper()	Uppercases a token for keyword comparison.	Case-insensitive grammar.
	parse(line)	Recognises CREATE / INSERT / SELECT / EXIT.	Converts raw text to command objects.
FileManager	createTable(name,cols)	Builds new .tbl, adds to cache.	One file per table.
	openTable(name)	Memoises TableFile objects.	Avoids reopening the same file.

Core design theory in simple terms.

- **Slotted page.**
  Fixed 4 KB block matches disk/SSD block size. Header + slot offset list → records can move without rewriting whole page.
- **Buffer pool.**
  Dirty pages sit in RAM; background flush or unpin flush keeps latency low. LRU is "good enough" for small systems.
- **Thread pool.**
  Separates latency-sensitive CLI from disk I/O. One pool is simpler than many ad-hoc threads.
- **Lazy loading.**
  We read pages only when a table is first touched, not at startup. Memory footprint grows with workload, not table count.
- **Background flusher vs. WAL.**
  Flusher reduces loss window to 2 s. WAL (next milestone) will reduce loss to **0 s** by fsyncing log before commit.

Dependency order (build)

```
Page  -> BufferPool -> Storage -> Commands -> Parser -> main
ThreadPool –/       |
BackgroundFlush–/
```

Common exceptions (Exceptions.hpp)

Class	Thrown when
StorageError	I/O fail, corrupt page, all frames pinned.
ParseError	Bad SQL syntax.
ExecutionError	Table missing, column mismatch, corrupted header.

All bubble up to main.cpp, which prints Error: … and keeps the REPL running.