PAGE MODULE:

- **Page::Page()**
  Zero-initializes an in-memory 8KB page image. Does not set headers yet.
- **void init(page_no)**
  Formats the page: writes a PageHeader (magic, page_no) and sets the slotted-page pointers (lower = sizeof(PageHeader), upper = PAGE_SIZE).
- **void load_from(raw)**
  Loads a raw 8KB byte buffer (read from disk) into the page's internal data.
- **void store_to(raw)**
  Writes the current in-memory page image into raw (exactly 8KB) so the caller can persist it.
- **nitems()**
  Returns the current number of **slots** (ItemIds) on the page.
- **free_space()**
  Returns free bytes available between the ItemId array (lower) and the tuple area (upper). Invariant: free space is [lower, upper).
- **Result add_tuple(tuple_bytes)**
  Attempts to insert a variable-length tuple: copies payload at the **top** (decrement upper), appends a new **ItemId** (increment lower).
  Returns the **slot number** on success or NO SPACE if not enough room (needs payload + one ItemId).
- **Result get_tuple(slot_no)**
  Fetches the raw tuple payload for a **live** slot (USED and not DEAD). Returns NOT FOUND if slot is out of range or dead/unused.
- **Status mark_dead(slot_no)**
  Marks the given slot as **tombstoned** (sets ITEM_DEAD). Returns OK or NOT FOUND (invalid or unused slot).
- **void compact()**
  Reclaims space by **repacking** all live tuples tightly from the top (reset upper to end of page, rewrite live payloads, rebuild their offsets), and clears dead/unused ItemIds.
- **PageHeader header()**
  Read-only access to the page header (for debugging/inspection: see magic, page_no, lower/upper).

ROW MODULE:

encode_row(Schema schema, List: row)

Turns a row (Value per column) into raw bytes we can store on a page.

- Builds a **NULL bitmap** (bit i = 1 ⇒ column i is NULL).
- Appends payloads for **non-NULL** columns in schema order:
- Prepends a **TupleHeader** with totals (size, column count, nullmap length, flags).
- Returns a self-contained byte vector ready for Page::add_tuple.

decode_row(Schema schema, data, len)
Reconstructs a row (Values) from the stored bytes.

- Reads and validates the **TupleHeader** (size & column count).
- Reads the **NULL bitmap**; for each NULL bit sets Value::null(type).
- For non-NULLs, parses payloads in schema order (same encodings as above).

TupleHeader {

total_len; // full tuple bytes (hdr + bitmap + payload)

attr_count; // number of columns

nullmap_bytes; // bitmap length in bytes

flags; // bit0=dead (0 means live in Phase 0)

};

- Stored at the beginning of each tuple; used by both encode/decode.

HEAP MODULE:

- **HeapTable::HeapTable(path, Schema schema)**
  Opens (or creates) the heap file at path and remembers the table schema.
- **ensure_first_page()**
  If the heap file is empty, appends page 0 and initializes it as a fresh slotted page (lower/upper pointers set).

- **Result insert(row)**
  Encodes the row (encode_row), then tries to place it on the **last page**.
  - If there isn't enough space, appends a new page, initializes it, and inserts there.
  - Returns the tuple ID (**TID** = {page_no, slot_no}) on success.
- **Status remove(TID)**
  Loads the target page and marks the slot dead (tombstone).
  Then calls Page::compact() to reclaim space.
  Returns OK or NOT FOUND if the TID is invalid/unused.
- **scan()**
  Full table scan: for each page and each live slot, decodes the tuple and prints the tuples.
- **npages()**
  Returns current number of pages = filesize / PAGE_SIZE.
- **bool read_page(page_no, out)**
  Seeks to page_no * PAGE_SIZE and reads exactly one page into out.
- **bool write_page(page_no, buf)**
  Seeks to page_no * PAGE_SIZE and writes one page from buf, then flushes.
- **append_new_page()**
  Extends the file by one zero-filled page and returns the new page number.

CATALOG MODULE:

What the Catalog does?

- Keeps a persistent mapping **table name → (relfilenode id, schema)**.
- Stores metadata in a human-readable file: data/catalog.txt.
- Creates/removes the per-table heap file under data/base/<relfilenode>.heap.
- **Catalog::Catalog(datadir)**
  Initializes a catalog rooted at datadir. Ensures datadir/ and datadir/base/ exist; sets catpath_ = datadir/catalog.txt; starts next_id at 1.
- **load()**
  Reads catalog.txt into memory.
  - Loads the TableEntry in tables.
- **create_table(name, Schema s)**
  Allocates a new relfilenode (from next_id++), records {name → (id, schema)}, and **touches** the heap file data/base/<id>.heap so it exists. Returns the new id.

- **bool drop_table(name)**
  Removes name from the in-memory map and deletes its heap file data/base/<id>.heap. Returns true if the table existed.
- **heap_path(relfilenode)**
  Builds the on-disk path for a table's heap file: datadir_/base/<relfilenode>.heap.
- **ensure_dir(p)**
  Creates a directory if missing (used for datadir/ and datadir/base/).

Data Members:

- datadir_ – root directory (e.g., "data").
- catpath_ – path to catalog.txt.
- next_id – next relfilenode to assign.
- map<string, TableEntry> tables_ – in-memory name→entry map.

SCHEMA:

- **Column**
  - **Fields:** string name; Type type;
  - **Column(name, Type t)** — Constructor. Sets the column's name, type.
- **Schema**
  - **Fields:** Ordered list of columns that defines the row layout.
  - **size()** Returns the number of columns in the schema.

When you "start the DB":

1. **Construct the Catalog**
   a. Catalog cat("data");
   b. Constructor ensures directories exist:

```
data/
  base/       (heap files live here)
  catalog.txt   (tiny metadata file)
```

c. Sets catpath_ = "data/catalog.txt" and next_id = 1.

2. **Load the catalog**
    a. cat.load();
    b. Opens data/catalog.txt if it exists.
    c. Builds a Schema (columns with Type).

        Records tables_[name] = { relfilenode=id, schema }.

So after startup we have:

- A memory map name → {relfilenode, schema}.
- A counter next_id for the next table id you'll allocate.

Creating a table

1. **Define a Schema**

```
Schema s;
s.cols.push(Column("id",    Type::INT);
s.cols.push(Column("name",  Type::TEXT);
s.cols.push(Column("age",   Type::INT));
s.cols.push(Column("active", Type));
```

2. **Create the table in the catalog**
    a. rid = cat.create_table("people", s);
    b. Inside create_table:
        i. Assigns relfilenode = next_id++.
        ii. Stores tables_["people"] = { rid, schema }.
        iii. Touches the heap file so it exists: data/base/1.heap (empty file).
    c. cat.save(); writes:

3. **Open the heap file**
    a. Look up the entry: cat.lookup("people", entry).
    b. Construct the heap:

```
HeapTable heap(cat.heap_path(entry.relfilenode), entry.schema);
```

c. HeapTable opens data/base/1.heap .
(It does **not** allocate the first page yet)

Inserting a row

Insert (1, "T", 26):

Row in memory

```
row = {
 Value::i v(1),        // id
 Value::s v("T"),      // name
 Value::i v(26),       // age
};
```

1) HeapTable::insert

- heap.insert(row) does:
  - **Ensure first page exists**
    - ensure_first_page() checks file size.
      - If empty: append_new_page() writes 8192 zero bytes (page 0), then:
        - Creates a Page, p.init(0) sets:
          - header.magic
          - header.page_no = 0
          - header.lower = sizeof(PageHeader) (start of ItemId array)
          - header.upper = PAGE_SIZE (8192)
        - store_to() → write initialized page 0 back to disk.
  - **Encode the row to bytes**
    - Calls encode_row(schema, row):
      - Builds a NULL bitmap (1 bit per column). Appends payloads in schema order:
        - Id, name, age
      - Prepends a **TupleHeader** (packed):
        - total_len = header + nullmap + payload

            - attr_count = 3
            - nullmap_bytes = 1
            - flags = 0 (live)
        - Returns one contiguous byte vector representing the tuple.
    - **Pick a page (last page) and add the tuple**
        - Reads page 0 into memory, constructs Page p from raw bytes.
        - Calls p.add_tuple(tuple_bytes):
            - Checks **free space**: upper - lower must be ≥ tuple_size + sizeof(ItemId).
            - **Places payload at the top**:
                - upper -= tuple_size
                - copies tuple bytes to data_[upper .. upper+tuple_size)
            - **Appends an ItemId** at the end of the item array:
                - Computes current slot = nitems()
                - Writes ItemId{ offset=upper, length=tuple_size, flags=ITEM_USED }
                - lower += sizeof(ItemId)
            - Returns the **slot number** (e.g., 0 on the first insert).
        - HeapTable writes the modified page back to disk.
        - Returns the **TID**: { page_no=0, slot_no=0 }.

Deleting a row

- heap.remove(TID):
    - Reads the page, p.mark_dead(slot) sets ITEM_DEAD in that slot's ItemId.
    - Calls p.compact():
        - Collects all **live** tuples.
        - Resets upper = PAGE_SIZE.
        - Rewrites live payloads tightly from the top, updates their ItemIds (same slot numbers).
        - Clears dead slots (flags=0, length=0, offset=0).
    - Writes the page back.