Storage Design

> Designed by ZHU Yueming on April 27th, 2025.
> Project framework provided by Zhang Ziyang.

Main Storage Structures

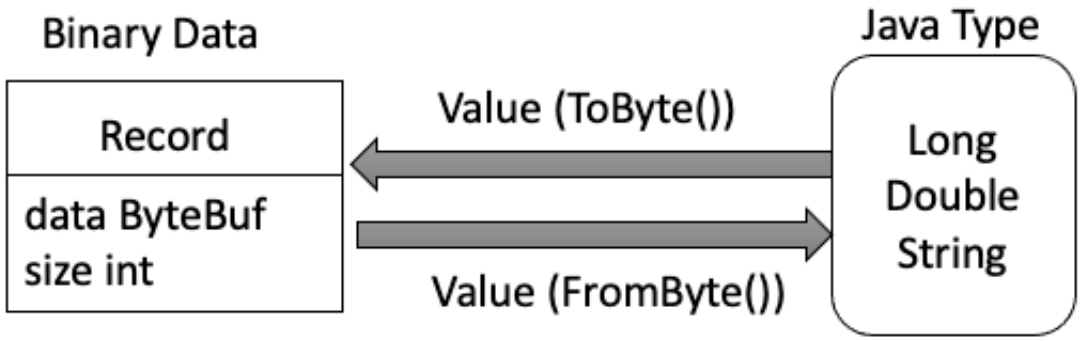
1. `ByteBuf` Data Type

`ByteBuf` is a binary data container provided by the Netty framework. In a database storage engine, it is used to represent binary records, raw page data from disk, etc. Essentially, it's a dynamic byte buffer that can store arbitrary binary data such as integers, strings, or serialized objects.

Usage in this project includes:

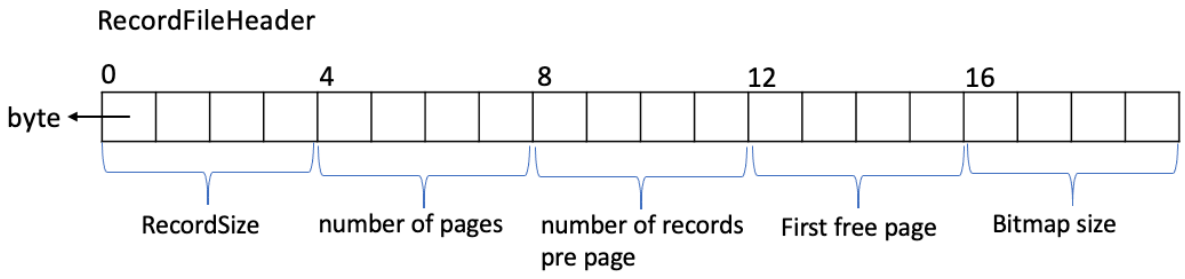
- **Record.data**: Stores a single database record in binary form.

  If `Record` is the container for binary data, and `Value` is the container for Java types, then we need a bridge between the two. That bridge is the `ToByte()` and `FromByte()` methods in the `Value` class, as shown below:



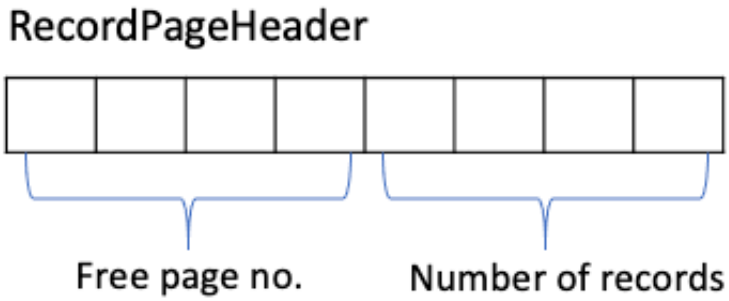
- **Page.data**: Raw byte data of a disk page.
- **RecordFileHeader.header**: Header information of the data file.

  The `header` field of `RecordFileHeader` stores content in binary form. Structure is as follows:



- **RecordPageHeader.data**: Page-level header information.

Stored in binary form as well. Structure is:



---

2. BitMap

The core function of the bitmap is **space-efficient slot management**. Each bit represents the state of a slot:

- 1 → Slot is **occupied** (contains a valid record)
- 0 → Slot is **free** (no record, ready for insertion)

Key methods:

```
// Initialize bitmap (set all bits to 0)
BitMap.init(bitmap);

// Set the 5th slot as occupied
BitMap.set(bitmap, 5);

// Check if the 3rd slot is occupied
boolean occupied = BitMap.isSet(bitmap, 3);

// Find the next free slot
int freeSlot = BitMap.firstBit(false, bitmap, recordsPerPage);
```

---

3. File and Page

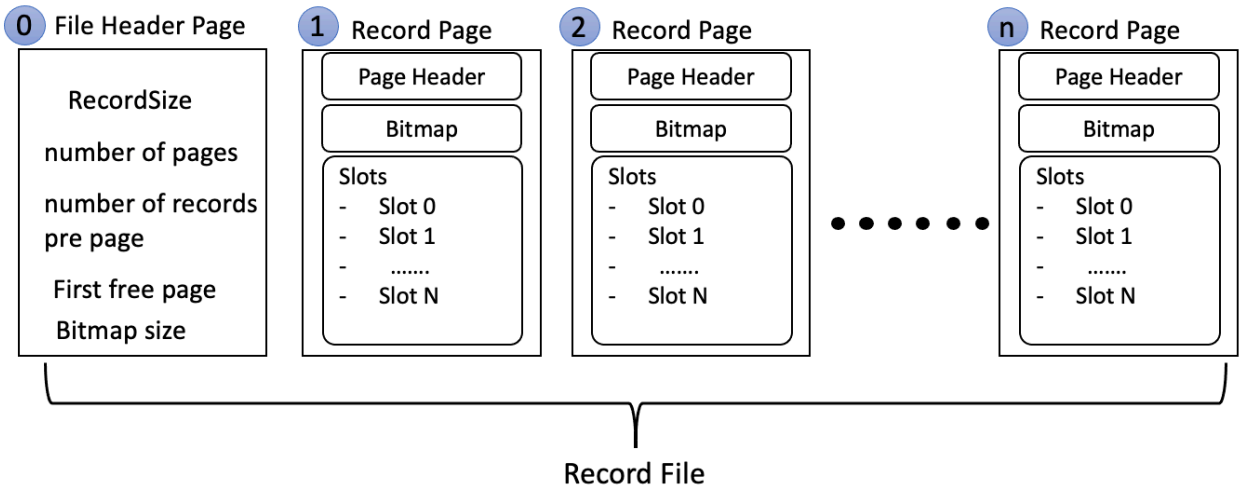
3.1 Structure Overview

In this project, each table is stored in a binary file. See `CS307-DB` → specific table folder → `data` file.

Each data file is split across multiple pages. The **first page** is the **file header page**, followed by **data pages**. Each data page has 3 components:

- Page header (first 8 bytes)
- Bitmap (tracks usage of each slot)
- Slots (each slot may store one actual Record)

Visual structure:



Note:
PageID is **not stored** in the page itself, but is managed in memory via offset:

```
public int getPageID() {
    return position.offset / DEFAULT_PAGE_SIZE;
}
```

PageID is computed like this:

- Offset 0 → PageID 0
- Offset 4096 → PageID 1
- Offset 8192 → PageID 2

RID:

`RID` is a Java in-memory object representing the position of a record: it contains the page number and slot number.

```
public class RID {
    public int pageNum;
    public int slotNum;
}
```

Summary:

- Valid data pages start from **PageID 1**
- Valid slot numbers start from **SlotNum 0**

3.2 Useful Methods

- How many pages does the file contain?

  ```
  fileHandle.getFileHeader().getNumberOfPages();
  ```

- How many records per page?

```
fileHandle.getFileHeader().getNumberOfRecordsPrePage();
```

- How to check if a slot contains a record?

```
BitMap.isSet(pageHandle.bitmap, currentSlotNum);
```

- How to get the Record object from a slot?

```
fileHandle.GetRecord(rid);
```

---

4. Record to Value Conversion

A `Record` stores one row in **binary form**. Understanding its layout is key:

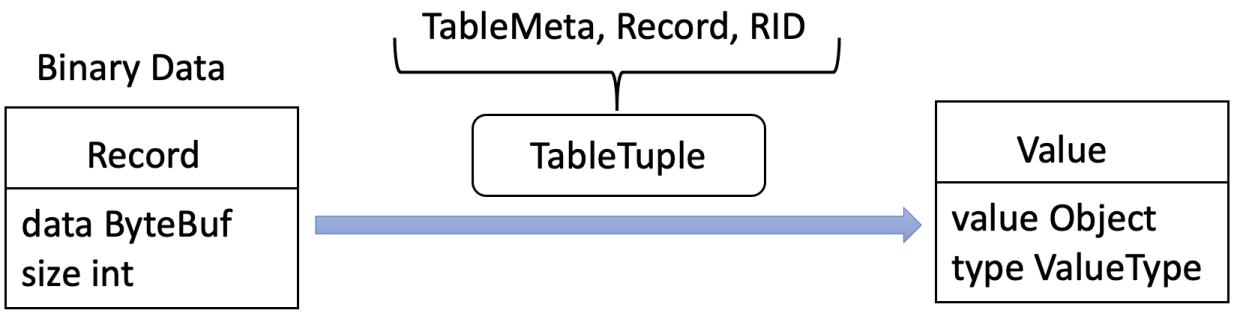
Byte Offset	Content
0	Column 1 value (depends on column type and length)
...	Column 2 value
N	Column N value

Each column has:

- `offset`: byte offset within the record
- `len`: number of bytes occupied by the column

Conversion is done via the `TableTuple` class, which uses the `TableMeta`, `Record`, and `RID` to extract `Value`s.

Structure diagram:



Exercise

Scan a table and count all rows, printing their values.

Write code to traverse each page in a file, and each slot in a page, to detect records. If a slot is occupied, extract and print its values. You may use the `BitMap` as a helper.

For example, given the table:

```
create table t( id int, name char, age int, gpa float);
insert into t (id, name, age, gpa) values (1, 'a', 18, 3.6);
insert into t (id, name, age, gpa) values (2, 'b', 19, 3.65);
insert into t (id, name, age, gpa) values (3, 'abb', 18, 3.86);
insert into t (id, name, age, gpa) values (4, 'abc', 19, 2.34);
insert into t (id, name, age, gpa) values (5, 'ef', 20, 3.25);
insert into t (id, name, age, gpa) values (6, 'bbc', 21, 3.20);
```

Complete the code below:

```
public class ScanExercise {
    public static void main(String[] args) {
        try {
            Map<String, Integer> disk_manager_meta = new HashMap<>
(DiskManager.read_disk_manager_meta());
            DiskManager diskManager = new DiskManager("CS307-DB",
disk_manager_meta);
            BufferPool bufferPool = new BufferPool(256 * 512, diskManager);
            RecordManager recordManager = new RecordManager(diskManager,
bufferPool);
            MetaManager metaManager = new MetaManager("CS307-DB" + "/meta");
            DBManager dbManager = new DBManager(diskManager, bufferPool,
recordManager, metaManager);

            RecordFileHandle fileHandle =
dbManager.getRecordManager().OpenFile("t");
            int pageCount = fileHandle.getFileHeader().getNumberOfPages();
            int recordsCount =
fileHandle.getFileHeader().getNumberOfRecordsPrePage();
            // TODO: complete the code here

        } catch (DBException e) {
            Logger.error(e.getMessage());
            Logger.error("An error occurred during initializing. Exiting....");
        }
    }
}
```

Expected output:

```
Page 1 Slot 0: is set. Values: a 3.6 1 18
Page 1 Slot 1: is set. Values: b 3.65 2 19
Page 1 Slot 2: is set. Values: abb 3.86 3 18
Page 1 Slot 3: is set. Values: abc 2.34 4 19
Page 1 Slot 4: is set. Values: ef 3.25 5 20
Page 1 Slot 5: is set. Values: bbc 3.2 6 21
The total count is : 6
```