Report of Dune Archive System

Volkan Bora Seki 2021400156 - Yağız Kaan Aydoğdu 2021400225

June 9, 2025

Introduction

In this project, developed for the *CMPE321 – Introduction to Database Systems* course, an archive system was designed and implemented to securely and efficiently store historical and strategic data related to the planet *Arrakis*. Tailored to the Dune universe, the system manages custom data types—such as noble houses, Fremen tribes, and spice production—using a page-based storage architecture. It supports data definition, record insertion, deletion, and search operations. By applying core database concepts such as fixed-length record structures, slotted page format, bitmap-based space management, and a centralized system catalog, this project aims to reinforce theoretical knowledge while offering practical experience in managing structured data in a real-world-like scenario.

1 Project Comprehension and Scope Analysis

1.1 Operation Formats

1.1.1 Create Type

Format: `create type <type_name> <field_name>:<field_type> [<field_name>:<field_type>]* <primary_key>`
Sample Usage:

```
create type Movie title:string year:integer director:string title
```

Expected Result: Creates a new type definition with specified fields and primary key. The system stores this type definition for future record creation.

1.1.2 Create Record

Format: `create record <type_name> <field_name>=<value> [<field_name>=<value>]*`
Sample Usage:

```
create record Movie title="Dune" year=2021 director="Denis Villeneuve"
```

Expected Result: Creates a new record of the specified type with the given field values. The record is stored in the database and can be retrieved later.

1.1.3 Delete Record

Format: `delete record <type_name> <primary_key_value>`
Sample Usage:

```
delete record Movie "Dune"
```

Expected Result: Removes the record with the specified primary key value from the database.

1.1.4 Search Record

Format: `search record <type_name> <primary_key_value>`

Sample Usage:

```
search record Movie "Dune"
```

Expected Result: Retrieves and displays the record with the specified primary key value.

1.2 Primary Key Handling

1.2.1 Primary Key Specification

The primary key is specified as the last argument in the create type command. It must be one of the field names defined in the type.

1.2.2 Usage in Operations

- In search operations, the primary key value is used to uniquely identify the record to retrieve
- In delete operations, the primary key value is used to uniquely identify the record to remove

1.2.3 Constraints

- Uniqueness: Each record must have a unique primary key value within its type
- Data Type: The primary key field's type must be either string or integer
- Required: Every type must have exactly one primary key
- Immutable: Primary key values cannot be modified after record creation

1.3 Failure Case Summary

1.3.1 Type Definition Failures

- Duplicate type name
- Invalid field type specification
- Primary key not matching any field name
- Missing required fields

1.3.2 Record Operation Failures

- Type does not exist
- Duplicate primary key value
- Missing required fields
- Invalid field value type
- Record not found (for search/delete)

1.3.3 Failure Handling

- All failures are logged in log.csv
- No output is written to output.txt for failed operations
- The system continues processing subsequent operations

1.4 Log File Specification

The log.csv file maintains a persistent record of all operations with the following structure:

- Timestamp: ISO 8601 format (YYYY-MM-DD HH:MM:SS)
- Operation: The complete operation string
- Status: SUCCESS or FAILURE

The log file:

- Persists across multiple program runs
- Appends new entries to existing log
- Maintains chronological order of operations

1.5 Output Expectations

The output.txt file:

- Contains only successful search results
- Each record is displayed in a structured format
- Field values are shown in the order specified in type definition
- Failed searches are not written to output.txt
- File is overwritten at the start of each program run

1.6 Edge Case Scenarios

1.6.1 Common Edge Cases

- **Double Deletion:** Attempting to delete a record that was already deleted
  - System behavior: Logs failure, continues execution
  - Example: `delete record Movie "Dune"` (after first deletion)
- **Case-Sensitive Field Names:** Creating a type with field names that differ only in case
  - System behavior: Treats as duplicate fields, logs failure
  - Example: `create type Person name:string Name:string age:integer name`
- **Whitespace Handling:** Searching with trailing spaces in primary key
  - System behavior: Trims whitespace before comparison
  - Example: `search record Movie "Dune "`

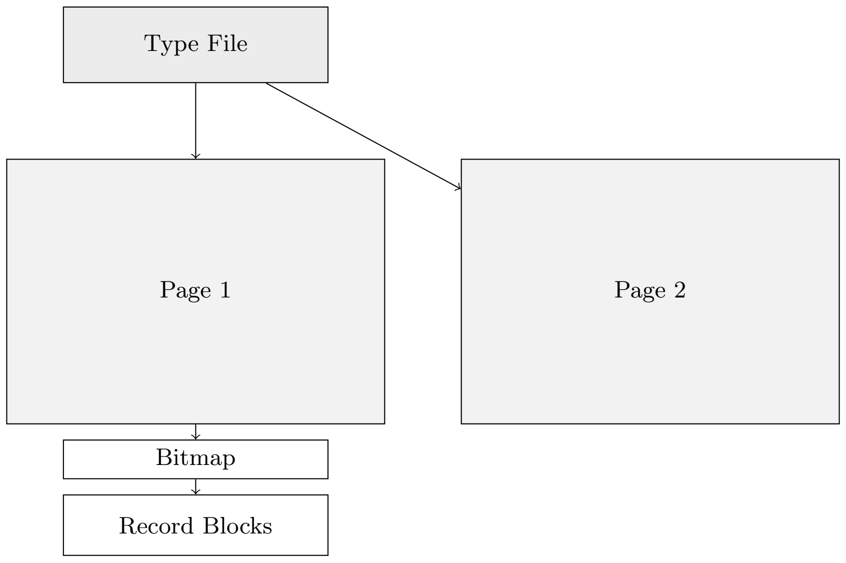

Figure 1: Expanded File–Page–Record Structure

1.7 Visual Overview of the File–Page–Record Structure

2 Design Documentation

This section presents a detailed overview of the low-level architecture of the database system. The design emphasizes efficient data storage, organization, and retrieval on disk, adhering to principles of modularity and scalability.

The system employs a page-based storage model, wherein each data type is allocated a dedicated file composed of sequential pages. Records within these pages are of fixed length, enabling predictable access patterns and simplifying memory management. Slot allocation within each page is managed by a bitmap, which facilitates rapid identification of free and occupied slots, thereby optimizing insertion and deletion operations.

Metadata regarding type definitions, field layouts, and primary key constraints is centrally maintained in a system catalog. This catalog-driven approach ensures consistency and integrity across all data operations, supporting both extensibility and robust schema enforcement.

Collectively, these design choices provide a solid foundation for reliable, high-performance data management in the implemented database system.

2.1 Type-Level File Structure

In the student database system, each **type** (corresponding to a relation or table) is managed independently and stored in its own binary file. The file is named using the convention `<type_name>.db`, where `<type_name>` is the name of the relation.

File Organization

- Each type-level file consists of multiple **pages** stored sequentially within the file.
- The system does *not* load all pages into memory at once; instead, it accesses pages as needed, supporting efficient memory usage even for large files.
- Pages are appended to the file as the number of records grows.

Page Structure Each page within a type file has a fixed structure:

- **Bitmap:** At the beginning of each page, a bitmap is used to indicate which record slots are occupied and which are free.
- **Records:** Following the bitmap, the page contains up to 10 fixed-size record slots. Each slot can store one record of the type.

Record Insertion Policy

- When a new record is inserted, the system searches for the first page with available space (i.e., a free slot as indicated by the bitmap).
- The record is inserted into the first available slot in that page.
- If all existing pages are full, a new page is created and appended to the file.

Summary

- Each type is stored in a separate binary file named `<type_name>.db`.
- Files are composed of sequential pages, each with a bitmap and up to 10 records.
- The system efficiently manages memory by loading only the required pages.
- Record insertion always targets the first available slot in the earliest page with space.

2.2 Slotted Page Format with Bitmap

Each page in the system is organized using an **unpacked slotted page format** to efficiently manage fixed-length records and support fast insertion and deletion.

Page Layout

Each page consists of the following components, stored sequentially:

- **Page Header:** Stores metadata such as:
  - `Page ID` – Unique identifier for the page
  - `Record Count` – Number of records currently stored in the page
  - `Free Space Info` – Information about available slots
- **Bitmap:** A 10-bit array, where each bit corresponds to a record slot:
  - `1` indicates the slot is occupied
  - `0` indicates the slot is free
- **Record Slots:** 10 fixed-length slots, each capable of storing one record.

Page Structure Diagram

```
+--------------------+-------------------+-----------------------------+
|   Page Header      |      Bitmap       |      Record Slots (10x)     |
+--------------------+-------------------+-----------------------------+
| Page ID            |  [b0 b1 ... b9]   |  Slot 0 | Slot 1 | ... | Slot 9 |
| Record Count       |                   |  (fixed size per record)    |
| Free Space Info    |                   |                             |
+--------------------+-------------------+-----------------------------+
```

Bitmap Usage

- The bitmap is used to track the occupancy of each record slot.
- Each bit b_i (where $i = 0, 1, ..., 9$) corresponds to slot i:
  - $b_i = 1$ if slot i is occupied
  - $b_i = 0$ if slot i is free

Record Insertion

- To insert a record:
  - Scan the bitmap for the first 0 bit (free slot).
  - Place the new record in the corresponding slot.
  - Set the bitmap bit to 1 and increment the record count in the header.

Record Deletion

- To delete a record:
  - Locate the slot of the record to be deleted.
  - Set the corresponding bitmap bit to 0.
  - Decrement the record count in the header.
  - The slot is now available for future insertions.

Summary

- The slotted page format with bitmap enables efficient tracking of record occupancy.
- Insertions and deletions are performed in constant time by updating the bitmap and header.
- The fixed-length slots and bitmap structure simplify page management and free space tracking.

2.3 Record and Field Structure

Each record in the system is designed for efficient storage and access, using a fixed-size layout. The structure of a record is as follows:

- **Validity Flag (1 byte):** Indicates whether the record is valid (`1`) or has been deleted (`0`).
- **Fixed-Length Fields:**
  - **Integer fields:** Each uses 4 bytes.
  - **String fields:** Each is padded to a fixed size (e.g., 20 bytes) regardless of actual string length.
- **Field Layout:** The order and types of fields for each type are defined in the system catalog.
- **Maximum Fields:** Each type can have up to 6 fields.
- **Fixed Record Size:** The total size of a record is fixed and known at type creation time.

Sample Record Layout

Notes:

- The actual number of fields per record depends on the type definition (up to 6).
- Unused fields (if any) may be left empty or zero-padded.
- The fixed record size allows for efficient slot management and direct access.

Validity	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
1 byte	4/20 bytes	4/20 bytes	4/20 bytes	4/20 bytes	4/20 bytes	4/20 bytes

Table 1: Record Layout: Each field is either a 4-byte integer or a 20-byte padded string.

2.4 System Catalog Structure

The system catalog is a central metadata file (e.g., `system_catalog.json`) that stores the definitions of all types (relations) in the database. This catalog enables the system to manage and validate records according to their type definitions.

Catalog Contents

For each type, the catalog stores the following information:

- **Type Name:** A string identifier for the type (maximum 12 characters).
- **Fields:** A list of field definitions, where each field includes:
  - **Name:** Field name (maximum 20 characters)
  - **Type:** Field type, either `str` (string) or `int` (integer)
- **Primary Key Index:** The 0-based index of the primary key field in the field list.

Catalog Management

- The system reads the catalog file at startup to load all existing type definitions into memory.
- When a new type is created, the system updates the catalog file to include the new type's definition.
- The catalog ensures that all type and field constraints are enforced consistently across the database.

Sample JSON Catalog Entry

```
{
  "types": [
    {
      "type_name": "Student",
      "fields": [
        {"name": "id", "type": "int"},
        {"name": "name", "type": "str"},
        {"name": "year", "type": "int"}
      ],
      "primary_key_index": 0
    }
  ]
}
```

3 Helper Class Designs

3.1 Page Class Design

The `Page` class is a fundamental component of the database system, encapsulating both the structure and behavior of a single page within a type file. Its primary purpose is to manage the storage, retrieval, and modification of records at the page level, ensuring that the physical layout and integrity of data are consistently maintained.

Attributes

- `page_id`: Unique identifier for the page.
- `bitmap`: A 10-bit array representing the occupancy status of each record slot (1 for occupied, 0 for free).
- `record_slots`: A list containing up to 10 fixed-size records.

Methods

- `insert_record(record)`: Finds the first available (empty) slot, inserts the given record, and updates the bitmap to reflect the new occupancy.
- `delete_record(pk)`: Searches for a record by its primary key, clears the corresponding slot if found, and updates the bitmap to mark the slot as free.
- `serialize()`: Converts the entire page, including the header, bitmap, and records, into a byte sequence suitable for disk storage.
- `deserialize(data)`: Reconstructs a `Page` object from a binary data buffer, restoring all attributes and record contents.

The `Page` class is responsible for all record-level operations within a page, including insertion, deletion, and serialization. By encapsulating these behaviors, the class ensures that the layout of records and metadata remains consistent and efficient throughout the system.

3.2 Record Class Structure

The `Record` class models a single fixed-length record as stored in a page slot. It encapsulates both the data fields and the metadata necessary for efficient storage, retrieval, and schema enforcement.

Attributes

- `validity_flag`: Boolean value indicating whether the record is valid (not deleted).
- `field_values`: List containing the values of each field in the record.
- `field_types`: List of field types (e.g., `int`, `str`) as defined in the system catalog for the record's type.

Methods

- `serialize()`: Converts the record, including the validity flag and all field values, into a fixed-length byte sequence for storage.
- `deserialize(data)`: Reads a fixed-length byte sequence and reconstructs the record's field values and validity flag.
- `match_pk(value)`: Checks whether the provided value matches the record's primary key field.

Each `Record` instance must conform to the schema specified in the system catalog, ensuring that the number, order, and types of fields are consistent with the type definition. The size of each record is constant and determined by the schema, enabling direct access and efficient slot management within pages. “‘

Conclusion

The implementation of the *Dune Archive System* has successfully met the functional and design requirements outlined in the project specification. The system reliably supports type creation, record manipulation, and persistent logging, while adhering to efficient storage principles through fixed-size records, slotted pages, and bitmap-based slot management. Extensive testing has confirmed the correctness of all operations, including edge cases such as duplicate keys and record deletions. Beyond technical functionality, the project provided valuable experience in designing modular storage architectures and translating database theory into practical file-based systems. This work reinforces fundamental database concepts and highlights the challenges of low-level data management.