CMPE 321, Introduction to Database Systems
Spring 2021
Project 4: Project HALO

Adalet Veyis Turgut - 2017400210
Ufuk Arslan - 2017400219
Zuhal Didem Aytaç - 2018400045

1 Introduction

Project HALO is a database design and implementation project that provides basic structure and functionalities of real life database systems. The design of the project were implemented according to the assumptions and constraints given in the project description. The storing assets of the database form a hierarchy. Each record is stored inside pages, those pages are stored in files, and files are organized according to their types. The program builds those aspects in nonvolatile storage with system folders and files, so as records and types get updates the data can be reached after the termination. To simplify the project, all the search operations are made with respect to the primary key of data types. The pages inside folders are stored in a stored manner according to their primary keys, which decreases search times significantly. This project offers a great variety of database functionalities from creating records to listing all records of any given type. The program reads user input line by line, processes through the database operations, and writes the requested result to a log file. For the implementation of the project Python3 were used. This report further explains the design and the implementation of the project.

2 Assumptions & Constraints

Here are the assumptions and the constraints of the system:

- Maximum length of a field name = **19 characters**
- Maximum length of a field value = **19 characters**
- Size of a field = **fixed 19 bytes**
- Maximum number of fields in a type = **10 (including Planet)**
- Size of a record = (number of fields) * (size of a field) = 10 * 19 = **190 bytes**
- Size of a page = **fixed 3192 bytes**
- Maximum number of records in a page = **3192** // (size of the type of a record)
- Size of a file = 3192 * 8 = **25536 Bytes**
- Maximum number of pages in a file = **8**

3 Storage Structures

- Types
  - Types are stored inside the `files.json` file. Each object key of `files.json` is the name of a specific type. Inside each type we store related files as json objects.
- Files
  - Files are stored inside the `files.json` file under their corresponding types. Each file object contains an array of page information. Those information consist of the page id (pointer to the page), start and the end of the interval of the primary keys inside that page. We store the end and start intervals to make searching and sorting faster.
- Pages
  - Pages are .csv files stored inside folders named as `file<file-id>`, and those file folders are stored inside the folders named as `<type-name>`. So, each page of the same type are stored together inside multiple folders. Page keys inside `files.json` file works as page headers as we can reach any page data from any type and file.
- Records
  - A record is basically a row inside `page<page-id>.csv` file. The fields of a record is given in the header of the corresponding .csv file. Start and end intervals of primary keys of records inside `files.json` file works as record headers as we can reach any record from any type, file, and page.

Listing 1 shows the content of an example `files.json` file.
Figure 1 shows an example directory structure of the database.

```json
{
    "_comment": "Comment explaining file.json structure",
    "animal": {
        "fields": [
            "planet",
            "id",
            "name",
            "age",
            "height",
            "weight"
        ],
        "file1": {
            "page1": {
                "intervalStart": "8",
                "intervalEnd": "1",
                "blankSpace": 28
            },
            "page2": {
                "intervalStart": "16",
                "intervalEnd": "9",
                "blankSpace": 28
            }
        }
    },
    "human": {
        "fields": [
            "planet",
            "id",
            "name",
            "age",
            "height",
            "weight",
            "alias",
            "occupation"
        ],
        "file1": {
            "page1": {
                "intervalStart": "8",
                "intervalEnd": "1",
                "blankSpace": 18
            }
        }
    }
}
```

Listing 1: Example `files.json` file

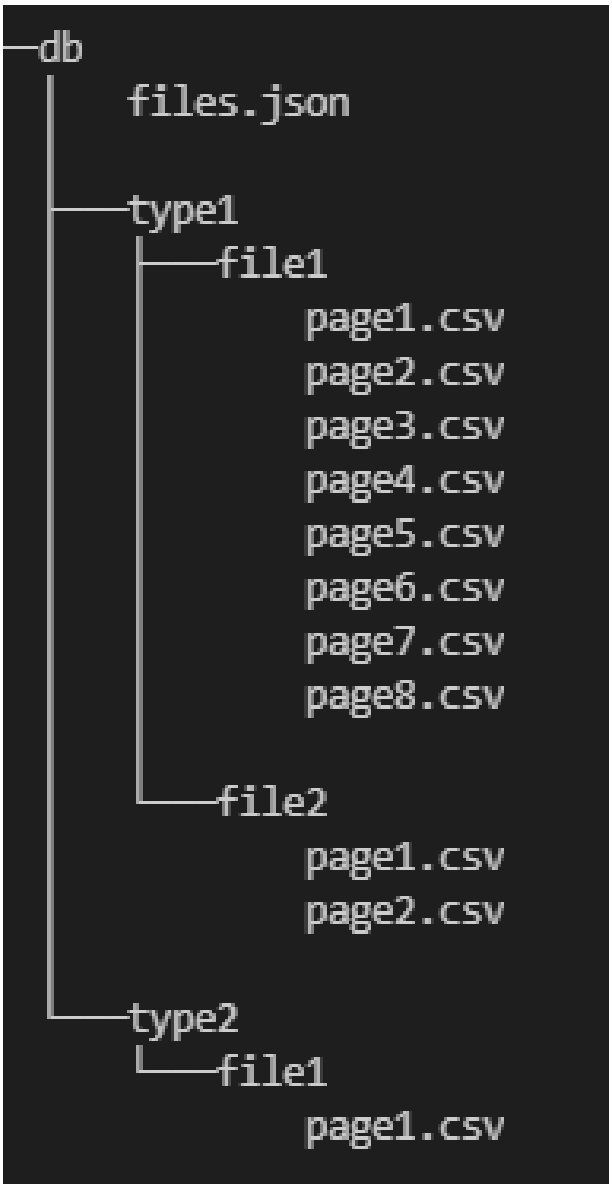

Figure 1: Example directory structure of the database

4 Operations

- **HALO Definition Language Operations**
  - **Create**
    - Creates a new type with given name that has the given fields.
    - Input format: create type <type-name> <number-of-fields> <field1-name> <field2-name>...
    - Output format : None
    - The type name can be at most 19 characters long.
    - The number of fields can be at most 9 (with default field Planet it can be at most 10).
    - The field names can be at most 19 characters long.
    - A page for the new type is created.
  - **Delete**
    - Deletes the given type and its records from the system.
    - Input format: delete type <type-name>
    - Output format : None
    - The type should exist in the system
    - The pages and files of the given type is deleted.
  - **Inherit**
    - Creates a new type with given target name that has the given fields and the fields of the source type.
    - Input format: inherit type <target-type-name> <source-type-name> <additional-field1> <additional-field2>...
    - Output format : None
    - The source type should exist in the system
    - Follows the same procedure with Create Type; type-name being target type name, the fields being the additional fields plus the fields of source type.
  - **List**
    - Lists the names of all types.
    - Input format : list type
    - Output format : None
- **HALO Management Language Operations**
  - **Create**
    - Creates a record of the given type with the given field values.
    - Input format: create record <type-name> <field1-value> <field2-value>...
    - Output format : None

    - The type should exist in the system.
    - The number of fields should match with the number of fields of the type.
    - The field values can be at most 19 characters long.
- **Delete**
    - Deletes the given record from the system.
    - Input format: delete record <type-name> <primary-key>
    - Output format : None
    - The type should exist in the system.
    - There should exist a record of the given type with the given primary key.
    - The record is deleted from the page it appears. If the page is empty after the delete the page is also deleted.
    - If the deleted page was the last page of a file, and there exists at least two files, that file is also deleted.
- **Update**
    - Updates the given record's fields with the given values.
    - Input format: update record <type-name> <primary-key> <field2-value> <field3-value>...
    - Output format : None
    - The type should exist in the system.
    - There should exist a record of the given type with the given primary key.
    - The number of fields of the type should match with the given field values.
    - The field values except the value for field Planet are updated.
- **Search**
    - Returns the field values of the given record.
    - Input format : search record <type-name> <primary-key>
    - Output format : <field1-value> <field2-value>...
    - The type should exist in the system.
    - There should exist a record of the given type with the given primary key.
- **List**
    - Lists the field values of all records of the given type.
    - Input format : list record <type-name>
    - Output format :
      <record1-field1-value> <record1-field2-value>...
      <record2-field1-value> <record2-field2-value>...
      ...

    - The type should exist in the system.
- **Filter**
    - Lists the field values of all records of the given type that satisfy the given condition.
    - Input format : filter record <type-name> <condition>
    - Output format :
      <record1-field1-value> <record1-field2-value>...
      <record2-field1-value> <record2-field2-value>...
      ...
    - The type should exist in the system.
    - The conditions should be valid.

5 R & D Discussions

In order to handle multiple HALO centers we may need to change the connection between operations and the storage. We could have a single storage for multiple centers with the current storage structure, however since there may be concurrent operations happening at multiple centers we would need to consider and avoid insert, delete, and update anomalies.

Distributed operations can be performed by implementing unclustered indexes. So, we may choose to use unclustered hash or B+ tree indexes. Since unclustered B+ tree indexing is estimated to give faster results on searching, we would implement unclustered B+ tree indexing on HALO systems. Since mission critical values are mostly known beforehand, we could create an unclustered index on frequently used fields of critical missions.

Our concerns regarding the distributed operations would be the the general disadvantages of implementing unclustered indexing. Our primary concern would be the slower search times of unclustered indexing compared to clustered ones.

6 Conclusions & Assessment

In conclusion, the design of the HALO project is very simple, yet very efficient. Storage structures of HALO were established in a well organized and consistent manner. The assumptions and constraints of the system like choosing length of a field's name and value made life a lot easier during implementation phase. However, those assumptions and constraints may limit user requirements.

While designing a database structure reducing search times is very essential since searching is used in many functionalities like updating, deleting and listing records. Since inputs are always given with the primary keys of records,

and records stored in files are indexed according to primary keys, average search operation is always faster compared to searching non-primary fields. However, in real life situations we will need to search/filter records based on a specific field of a type. Also, HALO does not provide many of the useful aspects of modern database systems like foreign keys, cascading, triggers and so on.