Storage Manager Implementation

CMPE321- 2018 Spring

Project 2

Alper Çakan - 2016400000

1. Introduction

In a DBMS architecture, there exists three main levels, namely; the view level, the logical level and the physical level. The final one, also called the “internal level”, is concerned with the actual storage of the data and the metadata on the secondary storage. In other words, it is concerned with the storage of the records and the metadata on the files, which exist on (usually) the hard drive.

The above mentioned physical level has parts such as storage manager, file manager and disk drive. The most important of them all, the storage manager, can be thought as the interface between the DBMS and all the “physicality” at the low levels. Storage manager is responsible for retrieving a record, and the other parts of the DBMS are only concerned with the records, not with files, pages, disk and so on. Also, it is important to note that, storage manager is of fundamental importance to the performance of the DB, because generally the costliest (read slowest) operation is disk/page accesses. Therefore, it is important to design a storage manager as efficient as possible.

In this project, we are to implement the storage manager system that we designed in the previous project, which should have a system catalog (which stores metadata) and should store the actual data on multiple data files. Furthermore, it needs to support common data handling operations of DDL and DML, respectively;

1) Creating a record type,
2) Deleting a record type,
3) Listing all the types,
4) Creating a record,
5) Deleting a record,
6) Searching for a record (by the primary key),
7) Listing all the records of a given type.

1.1. Implementation Details

The project was coded in C++11. For documentation, Doxygen’s Javadoc mode is used (however, the Doxygen related files are not included in the submission). Google’s C++ coding styles were followed.

The code is structured as follows:

- main.cpp
  This contains the entry point of the program, command line argument parsing and the main logic (data manipulation, storage management and so on).

- Page class
  This class represent a disc page. Basically, it holds a byte array (having type char *, since C++ does not have an explicit byte type) of size 2048. It also has methods for fetching the consecutive page in the file and so on.

- Disc class
  This class is a utility class, hence is composed only of static methods. Basically, it has methods for reading from and writing to pages. Since there is no API for reading a page of a file or changing the page size of the disc according to our choice, and since the disc is not actually 10 megabytes; this class actually reads from and writes to files using C++ standard library's file classes in binary mode. And to read or write a page, it simply seeks to the relevant position in the file, calculating that position using the page size. And to restrict the total size to 10 megabytes, it simply counts the total number of pages used by the storage manager.

- constants.h
  This file contains some constants, such as error and help messages, file names, header sizes, type aliases and so on.

The page header is composed of three 64-bit unsigned integers. The first integer signifies the status of the page. If it is 0, this page is not being used. If it is 1, it is being used. The second integer signifies the category of the page, such as data page, type page and so on. The third integer is the global address of the page. Global address of a page is unique across all files, whereas the local address of a page is unique only in the individual file. The local address is basically the index of the page in the array of pages of a file. Note that both global and local address are starting from 1.

The record header is composed of only a single 64-bit unsigned integer. This integer signifies the used/not used status of the page.

We also have a logical unit called "cell" in pages. This is like a record, but since the word "record" is used for data records, we call structural units of system catalogue etc. as a cell. Note that the definition/usage of the word "cell" includes data records. Some cells, such as record type cells, have a header signifying their used/not used status. However, not all kinds of cells have headers.

The system catalogue is divided into three files, which are named "general", "types", and "field names". The general catalogue file keeps only the global address counter. This is used in order to continue from the last global address even when the program is closed and later started. The types file contains type name, field count and field page address for each type. The "field page address" is the address of the page which contains the names of the fields of

this type. The “field names” file contains the field names of the types, and it is divided as exactly one page for each file.

The records for each record type is kept in separate files. These files are named the same as the record types.

When creating a new type, first we find the first unused page in the field names file. Then, we write the field names of the type to this page. Then, we find the first unused cell in the type info file. Here, we insert the information such as the type name and the field count.

When deleting a type, we simply find the cell of the corresponding type info and mark it as unused. Then, we mark the corresponding field names page unused.

When listing all the types, we simply iterate over the cells of a type info page. To fetch the field names, we read the corresponding field names page whose address we have obtained from the type info cell.

When creating a record, we simply insert the field values into the first unused record in the data file.

When searching for a record, we iterate over the records in the data file, and check if the primary key value matches with the one entered by the user. Note that, as specified in the first project report, we are setting the first field as the primary key.

When deleting a record, we first find search the record, and once we find it, we mark it as unused.

When listing all records, we iterate over all the records in the data file.

2. Changes From the Initial Design

There are many changes from the initial design. The foremost reason is that now we are using heap storage instead of indexing. The reason for this is that our disc capacity is limited to 10 MB. As a result, there is no need for indexing because not too many records can fit into 10 MB. So, there would be no performance boost by using indexing instead of unordered heap. Furthermore, using indexes would waste our precious storage with index file. Therefore, we opted not to use indexing. As a result, all of the DML and DDL manipulations needed minor changes. The details of the new algorithms can be found in the implementation details section above.

Another minor change is that we are starting the page addresses from 1 instead of 0. This way, we can have an illegal address, which is useful for error checking and distinguishing

pages which do not correspond to a page on the disc. We cannot use negative number because we are using unsigned integers for addresses.

Another change is this: we separated the system catalogue to three files. The details of these files are explained above. Since the information on these files are not strongly related, we decided to keep them in separate files instead of a single file.

Another minor change is that we are now using 8-bytes for marking a page or record as being used or not used. With this method, address calculations are easier. Furthermore, since the extra 7 bytes should always be 0, this becomes kind of a sanity/validity check.

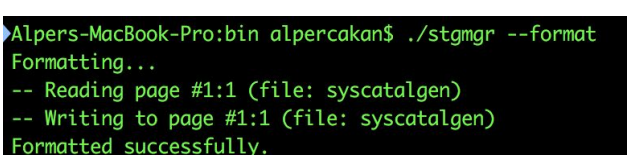
Another change is that we are now using a single page recording an individual record types field names. We decided to do this because of the storage restriction. It does not make sense to allow any number of fields for a record type, while we only have 10 MB disc. So now, at most ~32 fields are allowed for a record type.

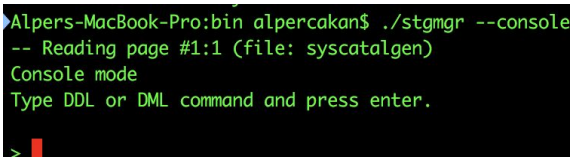
The last change is about the so-called "superdata" which is part of the system catalogue. The previous details of the superdata is given in the previous project report. However, due to the other changes explained above, none of the data mentioned the previous report is needed now. Instead, the next global page address is the only thing needed. So, we are now only storing the next global page address in the superdata, which is also now renamed to "system catalogue general".

3. Sample Usage and Outputs

Information about how to compile the source code to obtain the executable file can be found in the `README.md` file in the project directory. When the executable is ready, first run the program with the command line option --format to initialize the current directory as a disc drive containing an empty database.

Then, run the program with the command line option --console. In the console mode, you can run the DDL and DML commands, whose syntaxes are listed in the sections below in Extended Backus-Naur Form. The page address in the output are in the format "#global:local". When you want to exit, enter the command `exit`. Further info can be found in the README file and by running the program with argument --help.





A complete example execution (not including the compilation phase), user input in green:

```
$ cd bin
$ ./stgmgr --format
Formatting…
Formatted successfully.
$ ./stgmgr --console
-- Reading page #1:1 (file: syscatalgen)
Console mode
Type DDL or DML command and press enter.

> create_type Supplier s_id city_id country_id
-- Reading page #3:1 (file: syscatalf)
-- Reading page #5:2 (file: syscatalf)
-- Reading page #7:3 (file: syscatalf)
-- Reading page #9:4 (file: syscatalf)
-- Writing to page #9:4 (file: syscatalf)
-- Reading page #2:1 (file: syscatalt)
-- Writing to page #2:1 (file: syscatalt)
Supplier(s_id, city_id, country_id) is created!
> exit
$
```

3.1. Creating a Type

Syntax: create_type <field-name> {, <field-name> }

The command name for creating a type is create_type. After that, you list the field names, separated by whitespace. Since the only allowed type is 64-bit signed integers, this command is not taking field types. The first field will be the primary key. Hence, you should list at least one field. Further information about valid file names can be found in the first project's report.

It is recommended, but not required, that you use InitialCapsCamelCase for type and field names.

Note that this command will fail if the disc drive is full.

```
> create_type Supplier SId CityId TaxId
-- Reading page #3:1 (file: syscatalf)
-- Writing to page #3:1 (file: syscatalf)
-- Reading page #2:1 (file: syscatalt)
-- Writing to page #2:1 (file: syscatalt)
Supplier(SId, CityId, TaxId) is created!
```

3.2. Deleting a Type

Syntax: delete_type <type-name>

The command name for deleting a type is delete_type. The only argument is the name of the type.

```
> delete_type Product
-- Reading page #2:1 (file: syscatalt)
-- Reading page #5:2 (file: syscatalf)
-- Writing to page #5:2 (file: syscatalf)
-- Writing to page #2:1 (file: syscatalt)
Product is deleted!
```

3.3. Listing All Types

Syntax: list_types

The command name for listing all the types is list_types. It takes no argument.

```
> list_types
-- Reading page #2:1 (file: syscatalt)
-- Reading page #3:1 (file: syscatalf)
-- Reading page #5:2 (file: syscatalf)
Supplier(SId, CityId, TaxId)
Product(PId, Barcode, StockCount, IsImported)
```

3.4. Creating a Record

Syntax: create_record <type-name> <field-value> {, <field-value> }

The command name for creating a record is create_record. The first argument is the type of the record to be created, and the other arguments are the values of the fields (in the order that was given when creating the type).

```
> create_record Supplier 12 2432 5435341231
-- Reading page #4:1 (file: Supplier)
-- Writing to page #4:1 (file: Supplier)
Supplier(12, 2432, 5435341231) is created in page #4:1
```

3.5. Deleting a Record

Syntax: delete_record <type-name> <field-value>

The command name for deleting a record is delete_record. The first argument is the type of the record to be deleted, and the second argument is the primary key value of the record to be deleted.

```
> delete_record Supplier 6435
-- Reading page #4:1 (file: Supplier)
-- Reading page #2:1 (file: syscatalt)
-- Reading page #3:1 (file: syscatalf)
-- Writing to page #4:1 (file: Supplier)
The record is deleted from page #4:1
```

3.6. Searching for a Record

Syntax: search_record <type-name> <field-value>

The command name for searching for a record is search_record. The first argument is the type of the record to be searched, and the second argument is the primary key value of the record to be searched.

```
> search_record Supplier 12
-- Reading page #4:1 (file: Supplier)
-- Reading page #2:1 (file: syscatalt)
-- Reading page #3:1 (file: syscatalf)
The record is found in page #4:1
Supplier(12, 2432, 5435341231)
```

3.7. Listing All Records of a Type

Syntax: list_records <type-name>

The command name for listing all the records of a type is list_records. The only argument is the name of the type whose records are to be listed.

```
> list_records Supplier
-- Reading page #4:1 (file: Supplier)
-- Reading page #2:1 (file: syscatalt)
-- Reading page #3:1 (file: syscatalf)
Supplier(12, 2432, 5435341231)
Supplier(6435, 423412, 12343242)
```

4. Conclusions and Assessment

The reasons behind our design decisions has already been elaborated in their respective sections. To summarize the general design, we can say that our design is not very restrictive, so it would be flexible enough to be used in a wide range of cases. As a downside, some of the restrictions make this DBMS not usable for some rare tasks, such as storing some machine learning data: because of reasons such as the 64 fields limit. However, the project description says our disc is 10 MB, so those kinds of tasks are not suitable already. Because of the small disc size, 64 fields is actually more than enough. Also, the 32 character limit for

field and type names can also be considered to be too restricted; however, again the small disc size justifies this limit.

One might also consider the performance. Since we are using heap storage and not indexing, B-tree etc., one may worry about speed. However, since the disc is 10 MBs, even when the DB is full, there would not be too many records to be fetched in an unreasonable amount of time. Therefore, using indexing or B-tree would waste our most precious resource (which is the storage) with index files.