Project HALO: A CmpE 321 Project

Tolga Kerimoğlu, Gözde Ünver
2018400180, 2018400309

July 3, 2021

Contents

1 Introduction **3**

2 Assumptions & Constraints **3**
2.1 Assumptions 3
2.2 Constraints 3

3 Storage Structures **3**

4 Operations **6**
4.1 Descriptions of handling
add/delete/search/filter/update/read record operations 6
4.2 HALO Authentication Language Operations 8
4.3 HALO Definition Language Operations 8
4.4 HALO Management Language Operations 9

5 R & D Discussions **11**

6 Conclusion & Assessment **12**

1 Introduction

We have designed and implemented a functioning Database Management System from scratch using Python that supports various DDL and DML operations and **authentication**. The DBMS generates/deletes files and grows/shrinks the database as required and has a logging mechanism to record the outcome of each query into a log file.

2 Assumptions & Constraints

The below assumptions are generally in line with the project description. We have tried to restrain from making any additional constraining assumptions, only design choices we're considered.

2.1 Assumptions

- The files have limited page capacity and the system automatically generates and reallocates data once a file is full.
- The primary key of a record is assumed to be the second field of that record.
- The records in a file are stored in descending order of their primary keys.
- The records containing bigger primary keys for the same type are stored in files with names lexicographically smaller. For example, animalFile contains records with bigger keys than animalFile+.

2.2 Constraints

1. Maximum number of fields for a record is 12 (including the **planet** field).
2. Maximum length of a type name is 20.
3. Maximum length of a field name is 20.
4. It is assumed that no query conflicting with these constraints are entered i.e. we have not implemented error checking.

3 Storage Structures

In our design, we have tried to use principles of minimal complexity and maximum abstraction i.e. the abstractions of records do not spill out to pages and the abstraction of pages do not spill out to files. For example there is no information inside a page that stores which file it is in or to which type in the DB in belongs. Similarly, there is no information in a record about its place in its

current page etc. Let's get to the specifics.

Record Structure:
We have elected to go with the variable length record structure to avoid additional overhead. But it has a maximum length of 365 bytes because in the catalog file, records for a type information are stored and since we assumed that a type can have at most 12 fields with length 20 and type name of length 20 then a record must be at most 365 byte size. The first 3 bytes of a record is the offset of the data segment i.e. it marks the index where the record header ends and the fields data starts Following that, each 3 bytes is used for storing the offsets of the fields. For example the bytes 4-6 stores the beginning index of the second field, the bytes 7-9 store the beginning index of the third field etc. The final 3 bytes of the header points to the end of the record. Let's inspect a record with a single field with value **'CmpE321'**.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
0	0	6	0	1	3	C	m	p	E	3	2	1	

Table 1: An example record with indexes shown

$\underbrace{006}$ $\underbrace{013}$ $CmpE321$
Index of 'C' **Index to the right of the final '1'**

Page Structure: Our pages consist of a page header and then all the records inside the page stacked afterwards. Each page contains at most 10 records which comes out to a page size of about 3.6 KB. Let's walk through the page header. The first 3 bytes contain the page number for the file page is in. This is modified at the file level. For example, if there are 10 pages in a file, the 9th page would have the first 3 bytes set as '009'. The following byte (3rd byte) states the status of the page. If it is set to '1' the page is completely full, if it is set to '0' the page contains empty space. The next 10 bytes are for storing the status of the slots. For example, if the page contains only one record at the first slot, these 10 bytes would look like '1000000000'. If the page only has 1 slot empty at the end then we would have '1111111110'. Then the header also contains the primary keys of the first and the last record in the page. This is for sorting purposes and to check whether a record should be inserted in this page or not. These keys are preceeded with 6 bytes that state the offsets of the key values. Let's go over an example page header with 2 records, **('Kerimoğlu', 'Tolga')** and **('Ünver', 'Gözde)**. The superscripts denote the index of a byte.

$\underbrace{000}$ $\underbrace{0}$ $\underbrace{1100000000}$ $\underbrace{202530}$ $T^{20}olgaG^{25}özde^{29}$
First page in the file **Page is not full** **First 2 slots are used** **Offsets for keys**

File Structure: Files can have at most 1000 pages, which makes its size at most around 3.6 MB. Each file stores information about one type. Each file

has a name originated from the type of records it stores. The first file that is created when a new type is created in the system has '<type>File' as a name. Additional files for the same type that are created when the first file becomes full, have names as '<type>File{+...+}'. Each '+' is added for the additional file for the same type. files for the same type with less '+' in their names have records with bigger keys. Each file has a header. The header stores these information in order:

- 1 byte for the status of the file, '1' if all the pages of the file are completely full and '0' if the file is not full.
- 3 bytes for the number of existing pages in the file (i.e. if the file has 3 pages then '003' is stored)
- 1000 bytes for the status of pages. '0' if the page doesn't exist, '1' if page exists and not full, '2' if page exists and it is full

The file header has a fixed length of 1005 bytes.
When a record will be inserted into the file, the record is added to the correct page of the file where the key of the record is in the range of the page or lexicographically bigger than the biggest key in the page. When the record is added into the page, page status is updated both in the page header and in the file header. File status is also updated if all the pages become full after the insert. The program checks for the order of the keys while inserting records so even though the page is full and the record has a key bigger than the biggest key of the page, the record is inserted into the page and the record with the smallest key is shifted to the next page. This shifting may occur recursively until the insertion successfully ends. It might also be the case that file becomes full and there cannot be more pages, then a new file is created and the record with the smallest key in the old page is inserted into the new page.

Catalog and user files: At the initial boot-up of the system, a user and a catalog file are created. User file stores registered user records with their username and passwords. Catalog file stores name and fields of types in the system. User and catalog files have the same structure as a regular file in our system. They have a header and number of pages at most 1000. If the original catalog file/user file of the system which is 'CatalogFile'/'UserFile' is full, then a new one is created with '+' appended at the end of the catalog file/user file name. Catalog and user files don't store any fields in their headers just like any other file in the system.
For the catalog files:

- At the initial boot-up, 'User' type is also recorded in the file with fields 'Username' and 'Password' because it has a file in the system as well as other types.
- If a new type is created, the type name and its fields are stored in the catalog as a record like this:

" 'typeName' 'field1' 'field2' ... 'fieldN'
The record has typeName as the second element because in our record structure, the second field is unique for each record and since our catalog file has the same structure as any other file in the system, typeName must appear as the second field in the record. First element is an empty string because it has no use in storing record for a type information.

- When a type is deleted from the system, all of its files are deleted as well as its type information is deleted from the catalog

Records in the catalog and user files are also stored in decreasing order of their keys.

4 Operations

Write your DDL and DML operations in pseudocode by referring to the structures in your design.

4.1 Descriptions of handling add/delete/search/filter/update/read record operations

These operations are done in functions of file.py and their results are used in haloSoftware.py for halo definition and management language operations.

1. **Add record:** It is done in **file.py/add_record** function. A filename and a record are given as parameters. Empty spaces in a file are located at the end of the file so when a record will be inserted, pages are examined starting from the first page one by one via file seek(), read() and write() operations. First, the correct position to read the current page is located with file.seek() and data are read with file.read(PAGE_SIZE). If the page is full (page status is '2' in the file header) and the key of the record is bigger than the smallest key in the page, then the new record is inserted into the page anyway but the record with the smallest key value is popped and it is put into the next page. Record shifting is done recursively until all records find a place. If the page is not full but there are some records in it and the new record has a key value that is bigger than the smallest key in the page, then the record is inserted into the page without any shifting. If the record doesn't fit in existing pages in the file and there is still some empty space to create a new page, then a new page is created and the new record is inserted into this new page. When an insertion occurs, the modified page is written back to the file with first seeking the correct location and then writing PAGE_SIZE many data at that location. After the insertion is completed, page status, page number and file status in the file header are updated.

2. **Delete record:** the operation is done in **file.py/delete_record** function. All the non empty pages in the file are checked one by one (with file seek and read operations) to find the record and delete it. If after traversing all existing pages in the file, the record is not found, then the function returns False, if it is found and deleted then after the deletion this function returns True. When a page is read, the biggest and the smallest key in the page are used to compare with the key value of the record to be deleted. Those 2 keys are present in the header of the page. If the record is within the range of these keys (including these keys), then its records are read. If the given key value exists in these records then the record is removed from the page (the page is actually recreated in page.py/delete function using all the records in the page except the record to be deleted). If the modified page still has some records in it then it is written back to the file at the correct location. If the modified page now has 0 records in it, then the page is removed from the file. If there are pages that come after the deleted page, they are shifted to write with decreased page numbers. If the file doesn't have any more pages in it, then the file must be deleted. File deletion is done in haloSoftware.py if in file.py/delete_record deletes all of the pages in the file and in the file header status of all pages are 0. When a file is removed, its filename is also removed from catalog. After record deletion, file header is updated to store the current status of its pages.

3. **Search record:** Search in a file is done in **file.py/search_record** function. Each page is read one by one starting from the first page with file seek() and read() operations. If the key of the searched record is in the range of the biggest and the smallest keys of the current page (including these keys), then the records in the page are retrieved via page.py/ read_page function. If a record with the given key is found then the record is returned. After all the pages in the file is examined but a matching record is not found then the function returns False.

4. **Filter record:** It is done in **file.py/filter_records** function. Filename, field name, operation (<,>or =) and value to compare are parameters of the function. Each page of the given file is read one by one, starting from the first page via f.seek() and f.read(PAGE_SIZE) functions. First, the index of the field of the type is found because each value at that position in a record will be compared with the given value. For each record in the page, the field in question is tried to be converted to integer. If it cannot be converted, then it returns False. If it can be converted to integer, then it is compared with the value. If the record satisfies the condition, it is added to the list of returned results. If no record in a file satisfies the conditions, then an empty list is returned.

5. **Update record:** It is done in **file.py/update_record** function. It takes filename and record as parameters. It simply deletes the record with file.py/delete_record(filename,key of the record) first, if the return value is

False, then it means the record was not in this file so the function simply returns False. If the return value from the deletion is True (which means that the record used to be in this file) then the updated version of the record is added to the file with file.py/add_record(filename, record). It returns True after the updated version of the record is inserted.

6. **Read record:** It is done in **file.py/read_file** function. It takes filename and page number as parameters. It read the page of the file according to the given page number. It returns the list of records in that page.

4.2 HALO Authentication Language Operations

System starts with 'null' user information (username and password) before reading any instructions from the input file in the constructor of our haloSoftware.py file. If a login happens successfully then current user name and password is stored so that rest of the instructions can be done successfully. If no successfull login occurs then the rest of the instructions are not executed by our ddl and dml operations and became failure.
We handle authentication instructions in *def authHandler* function. First it checks if the instruction is login, logout or register, then does operations accordingly. If it is a login instruction, it checks if there is already logged in user. If there is, it is logged as failure else it checks if the user has already registered. If the user has registered already then the current user of the system becomes the given user, if not then it is logged as failure. If it is a register instruction, it checks if there is already registered user with the same username. If there is no such user, the given user is registered successfully (New user record is stored in the user file of the system) or else it is logged as failure. If it is a logout instruction, then the current user and its password of the system become null.

4.3 HALO Definition Language Operations

Definition language operations are handled in *def ddlHandler* function in our code. First the current user in the system is checked. If there is no logged in user, then the instruction immediately becomes failure. Different operations are done for 4 type of instructions. Each of them first checks if the given instruction has valid number of keywords otherwise it is immediately logged as failure. These instructions are:

1. **Create:** The new type is inserted to the correct catalog file with appropriate record shifting operations (according to their primary key). A new catalog file is created for the record with the smallest key value in all of the catalog files if the rest of the catalogs are full. After the type information is inserted to the catalog, a file for the type is created with only header and zero pages. The file name for the new type is <type>File.

2. **Delete:** Firstly, record containing information about the type is deleted from the catalog file that it was stored. Record deletion is done in file.py/

delete_record function. If after the deletion the catalog becomes an empty file, then it is removed from the system as well. If the type deletion from catalog was successful then it is logged as success and all the files containing records of the given type are deleted. If type deletion from catalog was not a success then it means that there wasn't any type with the given name in the system so it logs failure. If it was success, to delete all files for that type are retrieved via glob.glob(filename*) function. It returns all files with names containing 'filename' in it. For each of these files, they are removed from the system with os.remove().

3. **Inherit:** First the record that stores field information of the source type is found from the catalog file. If there is no such source type in the system then it logs failure and exits. If there is, adding the additional fields of the inherited type to the fields of the source type, the new type information is stored in catalog file. If there is already existing type with the same name, then it logs failure and exits. A new catalog is created to store the record with the smallest key in catalog files if after the insertion existing catalog files are not sufficient. When the type information is successfully added to the catalog, a file for the type is created.

4. **List:** For all catalog files in the system, each page is read one by one to get records. Since the types must be listed in ascending order, pages are read in reverse order because the last page stores the records with the smallest key values. Page reading is done by file.py/read_file function. For each page, records are also read in reverse order because in a page, last record has the smallest key value. for each record it is checked whether the key contains the substring 'User' because catalog also stores a record for user type. All the types are written to the output file. There is a boolean check variable and it stores True when there is at least 1 type in the system other than user type, so that system logs success, else system logs failure.

4.4 HALO Management Language Operations

First it is checked whether there is a current user in the system. If there is no logged in user, then halo management language operations are not performed and logged as failure. There are 6 type of operations. Before executing any of them, validity of the instruction is checked. If it is not valid then it is logged failure and instruction is not executed. Those instructions are:

1. **Create:** First, it is checked if there is a type information in the catalog for the given new record. If there is no record for that type then it is logged failure. If there is a type information the execution continues. The planet value 'E226-S187' is appended at the beginning of the new record's value list. All files for that type are retrieved via glob.glob(file for that type) function. If there exists at least one file for that type, then this new record is tried to insert into one of those files. Insertion is tried starting from the

file that contains records with bigger primary keys so the relative order between the records are maintained. Retrieving those files in an order that files containing the bigger keys appear first, is provided with sort() function because files with bigger primary keys have alphabetically smaller names. Record insertion is done via file.py/add_record function and each insertion attempt is checked according to the return value of this function. If the return value is 'False' then it means there exists a record with the same primary key so the new record is not inserted and it is logged as failure. If the return value is 'True' then the record is successfully inserted so its log is written as success. Lastly, if the return value is a list, which means the function returned a record, then this record must be shifted to the next file because the current file is full. This returned record may be either the new record or an old record with the smallest key in the file because the new record has been inserted but now there is no space for the returned record. The returned record becomes the new record to be inserted and this record insertion will continue recursively until all the records find a place. If there is still a record that cannot find a place, then a new file is created and the last record is inserted into this new file. Name of the file is '<name of the file that contains records with smallest keys of that type>+'. Last record is inserted into this file.

2. **Delete:** For each file that stores records for that type, its pages are examined. If there is no file for that type in the system, then it is logged as failure. If there are files for that type, for each of those files, file.py/ delete_record deletes the record with the given key by examining each page one by one in the file. If the return value from the delete_record function is 'True' then system logs as success and continues to check whether the all the number of pages in the file is 0. If it is 0, it means that file is now empty thus it is removed from the system.

3. **Search:** For each file that stores records for that type, its pages are examined. If there is no file for that type in the system, then it is logged as failure. If there are files for that type, for each of those files, its records are searched via the file.py/search_record method, page by page, according to the given key. If the return value of the search_record function is not an empty list, then it means that the record is found and it is logged as success. Even after traversing all the files in the system for that type, the record with the given key is not found, it is logged as failure.

4. **Update:** For each file that stores records for that type, file.py/update_record function is called on them. If the update_record function returns 'True' then it means that the record to be updated was in that file and it was updated so it is logged as success. All files for that type are checked until update_record function on one of them returns 'True'. If none of them returns 'True', then it is logged failure.

5. **List:** Files that contain records for that type are found and sorted so that files that store records with bigger primary keys appear in the file

list first. For each file, records in its pages are read by file.py/read_file function, starting from the first page to the last. All the records that are returned from this function (these records are also returned in decreasing order of their keys because they are stored in the file in this order) are written to the output file. If there is at least 1 record for that type, then it is logged as success otherwise failure.

6. **Filter:** Field, operation and value are extracted from the given filter condition. It is checked from the catalog whether type information is stored for the given type. If there is no record for that type in catalog, the it is logged as failure otherwise execution continues. For all the files in the system the filter operation is applied via file.py/filter_records function. If this function returns a non empty list, then those records in the returned list are written into the output file. Lastly, it is logged as success.

5 R & D Discussions

HALO Cluster
Execution Approach: Building a HALO Cluster entails building many HALO centers at different locations and executing the queries among all of them. When building such a structure, two different approaches come to mind. Firstly, we can run multiple queries in parallel but this approach has many problems, mainly about coherence. We would need to preemptively determine which queries could be run in parallel and produce the same results as if they were run in sequence i.e. which queries can be parallelized without creating an inconsistent state of the database. This is not a trivial task and gets exponentially more difficult with more centers. The second approach is running a single query in parallel. When thinking of a query as a collection of relational algebra operators cascaded in a certain way, one can see why it is inherently amenable to parallelization. For example consider a simple query such as
'SELECT *
FROM Students S, Professors P
WHERE S.gpa $\geq$ 3.5 AND P.citations $\geq$ 1000 AND S.interest = P.interest".
We can select all the students with gpa higher than 3.5 in one center while in parallel we can select professors with over 1000 citations in another center. Afterwards, the results can be pooled in a single center and the join operation on the 'interest' field can be performed. Even with a few centers operational, you can see how big of a speed-up this will yield for many different types of queries. This would be our approach to building a HALO cluster. Our approach to distribution would be the **Replication** approach since it allows for faster query evaluation in general (the queries that need not be parallelized can be performed on the local copy rather than retrieving data from a remote HALO center) and for increase availability of data. For a missions as critical as ours, it is paramount that even if a center goes down, we still have many copies of the up to date data the cluster can keep functioning without a problem.

Brief Architectural Discussion: On the physical implementation side exists 3 alternatives. The centers can have a shared global memory, each center could have its own memory but share a disk with all the others or they can share nothing and communicate just through an interconnection network. Both the shared disk and shared memory architectures suffer from **interference.**. It has even been shown that with such networks, after passing a certain number of centers, there is no additional performance increase and the performance can even start to decrease. Therefore, as it is currently mostly adopted by the popular DBMS, we would choose the 'share nothing' architectural approach since we have no information on how big the cluster will become and shared approaches may underperform with lots of centers. Note that for distributed centers it is not feasible to have a shared memory or disk since they can be very far apart. You can assume that the above discussion was conducted in a futuristic context.
Logging operation: Our approach of running a single query in parallel also produces a simple solution for any logging collisions. After the query and the sub-operations are run on the distributed centers, the final operations could be performed on the center that the query was initially run from. If any sub-operations produced an error in any of the centers, this error message would also be pooled into the original center. Then this original center would evaluate the final output (success/failure and returning tuples) and log the operation accordingly. This log file would be kept coherently among all centers and after each update, the updating center could send the new log to all other centers to keep each log synchronous and updated.
Query Optimization: Query optimization on parallel and distributed systems seem to have been a very deep topic with more than what meets the eye, at least from our look into it. Therefore, due to the limited time we've had, we can't confidently say we've learned all the ins and outs of it. Nevertheless, we've already presented some examples of query optimization. Firstly, as in the example we have presented above, the independent operations can be evaluated in parallel. When dealing with a distributed system however, the cost of retrieving data from remote centers can also be a problem and it can pose additional challenges for the optimizer to compute the cost of a query evaluation plan.

6 Conclusion & Assessment

Our program might be working slow for very large inputs because of record shifting in files but it gives correct result.
One of the most prominent design choices we have made is that all the file I/O operations we conduct are on the byte level i.e. we do not read single bits or extract any bit level information. For example, if we intend to store the status of a file (full or not) we write a single byte containing '1' or '0' whereas this obviously can also be represented as a single bit so do have lots of overhead in many cases. This was a convenience based design choice we've made early on that became too hard to replace without redoing the entire project. Overall, we have put in a serious amount of work and are happy with the outcome. It

was a very educative project that allowed us to learn about and combine many different parts of database management systems.