CSE510 - Project 1 Report

Group 11

Bala Sujith Potineni	bpotinen@asu.edu
Jackson Nichols	jcnicho2@asu.edu
Tracey Lott	talott@asu.edu
Vivian Roshan Adithan	vadithan@asu.edu
Vedanti Dantwala	vdantwal@asu.edu
Vraj Rana	vrana7@asu.edu

Abstract

This project served as an effort to translate the minibase code from a row stored Relational database to a Column stored database. Accompanying this design shift were the addition of new scanner classes, bitmap functionality, and a few programs designed to utilize these new features. Once everything was implemented the programs were executed on given sample datasets and the number of diskpage reads and writes were analyzed. Based read and write counts implementation works except query delete programs. Read and Write count gave deeper understanding about column store and more refined way to refined way to implement column store.

Acronym List

Acronym	**Term**
BM	Bitmap
DB	Database
DBMS	Database Management System

Introduction

Project 2 was intended to expand the minibase code base and implement behaviors and functionality of a Column stored DBMS. The Project description provided a minimum expectation for the classes, data members, and methods to be implemented to achieve this extended functionality [1]. The requirements laid out in that document served only as the minimum, and some additional classes were created in order to achieve the functionality of all the required methods. Among the minimum functionality expected during this phase were 4 programs that could be run from the command line and would utilize the updated Column stored DBMS.

Those programs were executed on some sample data sets. The input data set was using a provided test set containing 50,000 data entries. The program results were analyzed with a focus on the number of page reads and writes it took to perform the requested operation on the given index. At the start of each program we reset the read/write counters, and once the program completed its read and write count for the provided input arguments were output. This provided insight into performance of various implementation and design decisions.

Division of Labor

The team met twice near the start of February to discuss and divide up the workload for this project. After reviewing the scope of work, the efforts were bundled into the following groups of effort:

1. Tuple ID (TID) Class & ColumnDB Class
2. Value Class, Query Program & x Program
3. Batch_insert Program & Index program
4. Columnar Package containing Columnarfile Class & Tuplescan Class
5. File & Index Scans covering the iterator.ColumnFileScan class, global.Index type extension for Bitmap Index, and ColumnIndexscan class
6. Bitmap Package containing the BM Class, BitMapFile Class, BMPage Class, and HFPage functionality extension

The division of labor as a diagram was drawn up with very loose estimated completion dates to ensure the team was making continuous progress towards completing this phase of the project. That diagram was shared in the team's Discord as follows:

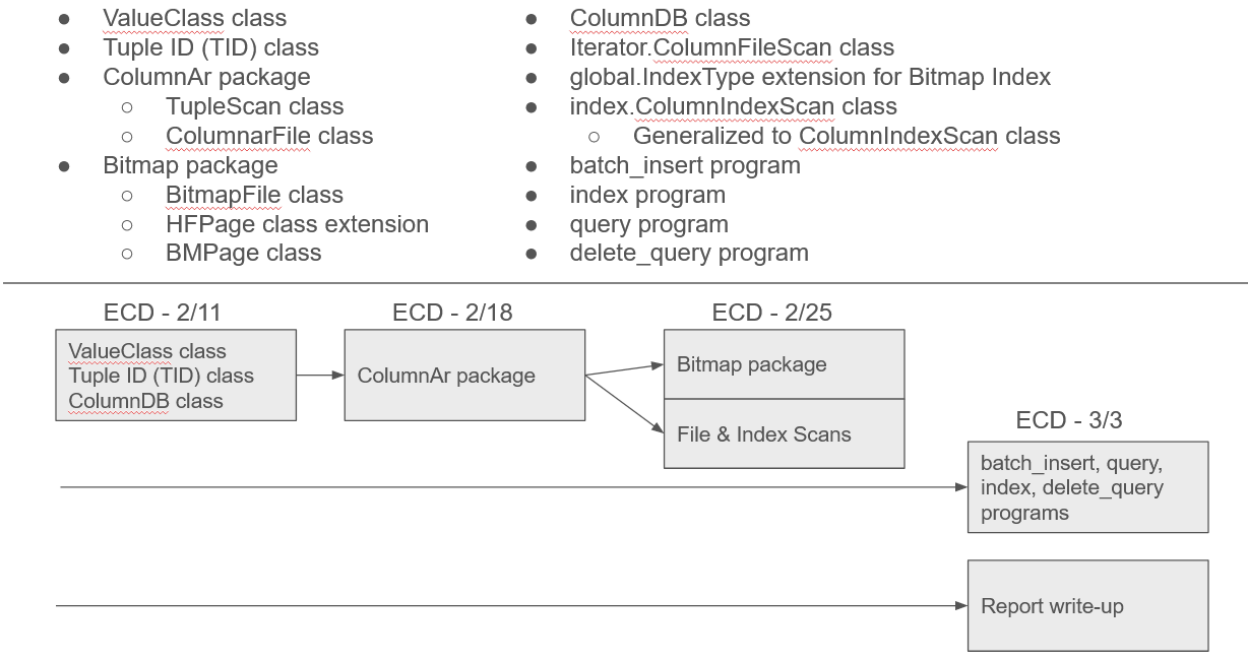

Figure 1: Projected Division of Labor

Team members volunteered and confirmed which group they were working on for Project 2 on February 8th. The division of labor was originally decided to be

- Roshan would work Group 1
- Tracey would work Group 2
- Vraj would work Group 3
- Sujith would work Group 4
- Vedanti would work Group 5
- Jackson would work Group 6

This was the intended plan for the division of work. In practice Roshan ended up doing Group 1, Group 4, & Group 5. He received some assistance from Jackson and Sujith for the later 2 groups of effort, but was the main contributor for those efforts.
Each team member summarized their design decisions for the groupings of work they implemented in the final report write-up.

Implementation Details

This section is aimed at providing a summary of how the different classes and packages were implemented in minibase. The details of implementation were kept to a higher level as to be supporting documentation for someone who has access to the code. Any class or package that was implemented with notable analysis of trade-off to other considered designs will have a brief explanation of the other designs and why they were not used in final implementation.

Value Class

The Value Class served as an abstract base class for the different types of input that could be stored in the Column Oriented DB. It served as a way to create a single method signature that could handle variable input types.
Its implementation consisted of 3 key member variables: an integer indicating the inherited type, a byte array containing the value, and an integer storing the length of the byte array. In implementation this parent class was extended to a StringValueClass and an IntegerValueClass.

Tuple ID (TID) Class

The TID class followed the specifications in the phase 2 document very closely. It consisted of a counter of the number of RIDs, a unique positional argument, and then the RIDs. Each RID pointed to the Page and Slot for that entries column value store. This class did not have much else in terms of functionality or design; it existed more so to be used by the other classes and packages.

RID:

Page No.	Slot No.

TID:

No. of RIDs	position	RID 1	RID 2	…	RID N

Columnar Package

The Columnar Package is responsible for storing, managing and also retrieving the columnar data. The Columnar package consists of a Columnarfile class and a Tuple class as per the specification.

Columnarfile Implementation details

The Columnarfile class is the main class of the package. It is responsible for storing and managing the columnar data. We have implemented the Columnarfile class in such a way that it consists of a Header Heapfile for the metadata, a Heapfile for every column and a deleted Heapfile to store the tuples that are marked as deleted. To insert, a tuple to be inserted is given as input and we separate the tuple into its columns and insert the columns into the respective Heapfiles. To get the tuple using TID then we have to get the values in the rid of each column's heapfile and combine them to form a tuple. To update the attribute of a tuple, we will update the attribute by deleting and inserting the value in the respective columns Heapfile. To update the entire tuple, we will need to call the update column for all the available columns. To get the tuple count we take any Columnar Heapfile and get its count. if the column is marked deleted we

store the deleted tuples TID in the deleted heapfile. To purge all the deleted tuples we will merge all the deleted tuples from the file as well as all from all index files. This implementation allows us to use all the already existing features of the Heapfile with only some minor modification. So this approach was chosen.

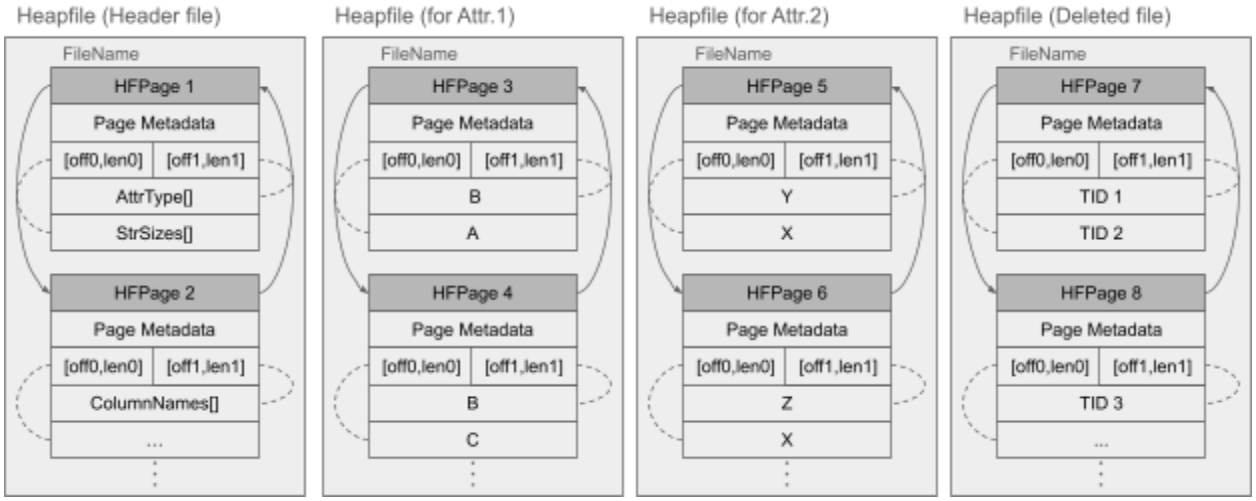

Figure 2: Columnar Heapfile Implementation Diagram

TupleScan Implementation Details

The TupleScan class is responsible for scanning the tuples in the Columnarfile. We have implemented the TupleScan class in such a way that it consists of a List of Scan objects for each column. We have implemented the openTupleScan in the Columnarfile class which opens the list of Scan objects for each column. We have implemented TupleScan in such a way that it mirrors the implementation of the Scan class in the heap package. If we want to move to the next tuple we call get_next for all the scan objects for the columnar heapfiles. Construct the TID and tuple and return it. Similar implementation is done for the position and close methods calling their respective methods for all the scan objects.

Bitmap Package

This implementation covers the BM class, the BitMapFile class, and the BMPage class specified in the required list of classes and methods to implement. It also covers the BitMapHeaderPage class and BMFileScan class introduced to complete the implementation of those classes.
There were a few ways to implement a bitmap reviewed during this phase. The main two elements considered when designing the bitmap were its extensibility & efficiency. Simplicity of design was also considered, but not as heavily as the other two. Extensibility was a factor because the scope of bitmap functionality was understood to need to cover any integer value and any string value. Efficiency was considered and measured as the number of pages needed to manage a bitmap and the number of pages needed to perform operations on a bitmap. The final design was to create a single, doubly linked list of BMPages that could support variable length component bitmaps.

Bitmap Class Implementation details

The Bitmap file mirrored the BTreeFile rather closely in terms of functionality. IT had the specified methods from the project description [1] but also extend the IndexFile and implemented those abstract classes. This IndexFile extension was so the BitmapFile could be used by the FileScan class and ScanUtils class in a manner similarly to the BTreeFile. The Bitmap file kept track of four class variables: a pointer to the first BMPage in the doubly linked list for this bitmap, the name of the columnar file and column number this bitmap was mapping, and the type of value in that column being mapped.

The BMPage followed the HFPage very closely in terms of implementation details. The design was each bitmap value entry would be stored starting at the end of the page data array. Towards the head of data array would be a slot pointer, and this slot point would store a pointer to the start of the bitmap, the length of the bitmap, and the RID of the value it's mapped from the column heapfile. Each page would track how many bitmaps it had and how much free space was available for easy access on inserts.

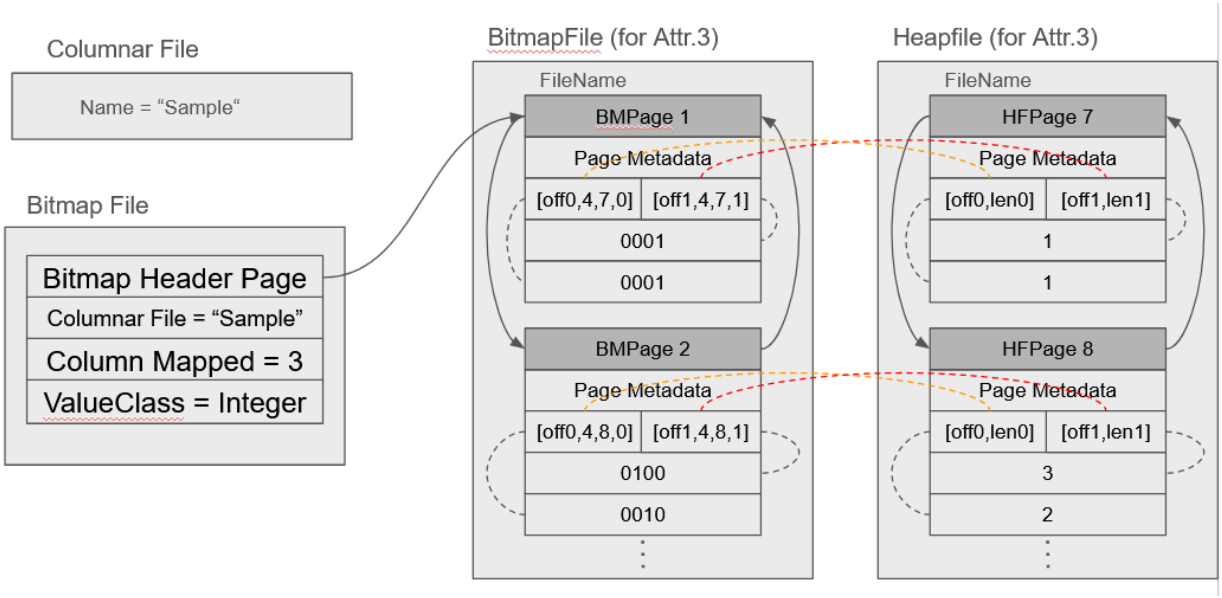

Figure 3: Simplified diagram of Bitmap implementation

When creating a new Bitmap Index File it is passed an argument for the Columnar File, column in the column, and the Value Class to map. Using that information the bitmap index scans each existing entry in the Columnar Heapfile and inserts each entry into the bitmap. When inserting the map it also associates the map back to a Page and Slot ID for the mapped column value. It uses the insertMap() method and this method has added functionality to prevent inserting the same RID multiple times by first verifying the RID isn't already mapped. If it isn't mapped already it will insert it into a page with space or create a new page for the map.

When deleting, the bitmaps follow a similar methodology as the HFPage where it will take in a position, which will be translated to an RID in the Heapfile, and find the corresponding RID. Then it will set the map slot to INVALID_PTR and delete the map from the page data array. After

deleting the mapping it will shift any other maps in the same page to free up space in that page for additional maps.

Bitmap Mapping Design

The implementation of the component bitmaps took different forms for integers and strings.
For Integers the components were broken up into 4 components, with each component representing a power of 256. This created a bitmap design similar to the following

[255, 0] * 256^3	[255, 0] * 256^2	[255, 0] * 256^1	[255, 0] * 256^0

However, integers can be mapped in the negative range and the above only handles positive integers. To address that, before mapping the value, the raw integer input had 2,147,483,647 (the negative INT_MIN value) added to it. This allowed for mapping any integer, positive or negative, into the map. This design, uncompressed, would take 1024 bits to represent (or 128 bytes). Since we know each component will only set 1 bit and the rest will be 0s, each 256 bits in a component could be compressed to a byte indicating which bit was set; reducing the size of the integer bitmap to 4 bytes in the BMPage.
Strings followed a similar methodology, but needed to be more dynamic since no two strings would be the same length.The mapping for strings was narrowed down to the idea that each character in the string could be any ASCII character with a byte value between 0 and 127. This led to the design of having each character be represented by a 128 bit bitmap and the BMPage would store in the bitmap length the consecutive number of maps to form a single string. This design preserved string ordering without introducing overhead of some sorted data structure. The string bitmap had the structure:

[0, 64]	A (65)	B (66)	C (67)	D (68)	E (69)	F (70)	[71, 127]
...	0	1	0	0	0	0	...
...	0	0	0	0	1	0	...
...	0	0	0	1	0	0	...

In this case, as with the integers, this mapping is very compressible because each character has 127 zeroes and a single one which took 16 bytes per character. So each map was compressed to a byte representing which bit in the map was set.

Single, Doubly Linked List Reasoning

The implementation described above puts each entire bitmap on a BMPage. The alternative approach considered was to create a BMPage list for each bit in the bitmap. There are possible

solutions, but the more these two extremes of implementation mix, the worse the efficiency becomes.
To best explain the reasoning of a single we will take a hypothetical situation where each BMPage can store 64 bytes of bitmaps. Our expected range of inputs is still any integer and any string. The pros and cons were considered in the following table

Criteria	**Single list of BMPages where the entire Bitmap entry is stored on a BMPage**	**List of BMPages where each page stores the single bit of the entry**
Number of doubly linked lists to manage	1	128 (8 components each with 16 bits is best sizing balance for full integer range)
Values that can be stored in a single BMPage	16 per page (64/4)	512 per page (64 * 8 bits)
Page reads to count matches for a value out of 1024 entries	64 Page Reads (1024/16 per page)	16 Page Reads ((1024/512 per page)*8 components)
Number of BMPages to represent 1024 entries	64 Pages	256 Pages
Page accesses to insert a value when pages have space	n+1 to find empty space (where “n” is current page count)	8 Pages
Page accesses to insert a value when pages don’t have space	n+2 to find empty space (where “n” is current page count)	8 Pages + 128 (to create next link in component lists)
Pages needed to read a value at some position	n/2 (where “n” is current page count)	128 Pages

Given the above analysis, this is why a single linked list storing the entire bitmap was used. The BMPage list per bit had notable improvements in equality search time, but severe impacts for page maintenance overhead. Since we didn’t know if the system was going to be used for equality searches, range searches, or scans we needed to take the path with the least overhead for the average case. We were also expected to handle any integer which introduced limitations in how the components could be set-up.
Dynamically updating the range of the map or components (ie, initial range was [0,7] but we insert 10 and now support [0,10] bitmapping) was also considered but the overhead to update old maps and unknown range bounds on inputs that could be inserted was undesirable.

File & Index Scans

ColumnFileScan

The ColumnarFileScan class is just a mirror of the FileScan class which is used for the Row implementation. We have just used TupleScan here instead of the Scan class used for the row implementation to get the tuple here.

ColumnIndex Scan & ColumnarIndex Scan

ColumnIndexScan Scans class here is used to create scan classes for BTree as well as BitMap implementation and they return the key,rid to the caller of the get_next_keyDataEntry. And the ColumnarIndex Scan class takes the key and rid and constructs the tid using helper functions like getPositionFromRid and getTidFromPosition which are present in the Columnarfile class. The constructed TID is then used to get the tuple which is projected according to the requirements of the user.

Bitmap Index Scan

The Bitmap index scan used the BTree index scan as a reference for implementation. It was observed that the BTree scanner call structure passed its arguments along to IndexUtil::BTree_scan(). This method set-up the conditions for the scan of the BTree to support conditional expressions, such as less-than and greater-than-equal-to type conditions, on matches for the scan. This seemed very useful for use in projections and range searches and was mirrored in the Bitmap index scan inside IndexUtil::BM_scan(). That is where the mirroring of implementation ends as the BitMapFile::new_scan could not function similarly to a BTree scan due to inherent differences in the two index architectures.

The Bitmap index used the condition arguments passed to the new_scan method and converted them into bitmaps of the same style as the Index it would be searching. This allowed for faster comparison of the component bitmaps; if the first component didn't match the conditional criteria then the rest of the components of the map could be ignored. The scan would then take the BitMapHeaderPage for this file and wait for a call to get_next() by the scanner object.

The get_next() method was implemented such that it would iterate map-by-map in each page of the doubly linked list of BMPages until it found the next entry that would satisfy the scan conditions. Once a match was found it would increment to the next map in the page, and then return the matched, un-bitmapped value in the form of a KeyDataEntry object. It also stores the RID of the match in the curRID member variable.

The delete_current() method can use the above curRID to quickly delete whatever was last returned by get_next() in the scanned bitmap index file.

A quick note about the keysize() method in the BMFileScan class. This method doesn't have a great deal of use compared to their counterparts in the BTreeFileScan class. keysize , as

returned by keysize(), doesn't serve much use in a bitmap. It was included because it was an abstract method in the IndexFileScan and needs implementation in its inheritor methods.

This class has a known weakness that could not be resolved before the submission date. When running a Bitmap Index scan the BMFileScan class pins the first page of the Bitmap file. At the conclusion of the index scan that page should be unpinned, but attempts to do that kept returning PAGE_PINNED error indicating it was still failing to unpin that page.

Programs

Batch Insert Program

This program is designed to facilitate the insertion of multiple records into a database with a single command, eliminating the need to manually insert each record. To execute the command, the user must navigate to the "tests" folder of the Minijava database. Attempts were made to enable execution from the "src" folder, however, this functionality was not achieved. An example command executed with sample data is provided below:
make batchinsert DATAFILENAME=datafile.txt COLUMNDBNAME=db
COLUMNARFILENAME=file NUMCOLUMNS=4

Once within the "tests" folder, the user is required to execute the above command to perform batch insertion. Upon execution of the "make" command, the Makefile will trigger the following sequence:

```
BatchInsert: BatchInsert.java
   $(JAVAC) BatchInsert.java TestDriver.java

batchinsert: BatchInsert
   $(JAVA) tests.BatchInsert $(DATAFILENAME) $(COLUMNDBNAME)
$(COLUMNARFILENAME) $(NUMCOLUMNS)
```

To execute "batchinsert", the Makefile necessitates the provision of variables as demonstrated in the above example; failure to provide these variables will result in an error. With the appropriate inputs, the Makefile initially compiles the "BatchInsert.java" class and then passes four arguments through the Command Line Interface (CLI).

Required Arguments:

1) **DATAFILENAME:** This is the source file from which data is read and subsequently inserted into the database. The program assumes that the file is located in the "tests" folder; alternatively, the user may provide the full path to the data file.

2) **COLUMNDBNAME:** This denotes the database into which the user wishes to insert records. The specified database may exist or not.
3) **COLUMNARFILENAME:** This represents the file within the database where the particular records reside. This file may or may not exist prior to the execution of the command.
4) **NUMCOLUMNS:** This variable indicates the number of columns the user wishes to save in the database from the data file. It should be an integer value between 0 and the number of columns present in the data file.

With the above assumptions, upon command execution, the program follows the steps outlined below to accomplish the task:

1. The program checks the length of the arguments, and if four different variables are not provided, it prompts the user to enter the required arguments and then terminates.

2. Upon receiving correct input, the program parses the input into four parts. It appends the ".minibase-db" extension to the COLUMNDBNAME input and converts NUMCOLUMNS into an integer type.

3. The program verifies whether the COLUMNDB already exists. If it does, the systemDef minibase restart flag is set to true, indicating the intention to open an existing database; otherwise, it is set to false, and a new database is created with 10,000 pages. Additionally, the program sets the maximum buffer allowed via the Gloab.NUMBUF variable and chooses the replacement policy of clock with the given COLUMNDB name.

4. Subsequently, the program reads the first line of the file. According to the provided sample data, the first line should contain "columnName: columnType" pairs with spaces as delimiters. If the first line does not adhere to this format, the program exits with a failure.

5. From the first line, the program extracts attribute types for all columns, column names, and the number of columns with string data types. This information is necessary for creating a new columnar file if required.

6. The program proceeds to check if the columnar file exists. If it does not, the program creates the columnar file with the specified COLUMNARFILENAME and metadata extracted earlier.

7. With a loop, the program reads each line from DATAFILENAME one by one, where values are separated by spaces and converted to attributes. These attributes are then inserted into the database by calling the insertTuple function. After inserting all the records, the program flushes the page to complete the write process and prints the number of write and read counts for the batch insert program.

InsertTuple Method:

This method is designed to insert one row at a time into the columnar database. It begins by preparing metadata. The method accepts a byte array containing tuple data, which is prepared in the program via tuple.getTupleByteArray(). From the byte array input, the insertTuple method retrieves attribute data and its size. Other metadata, such as the number of columns and column data types, are accessed via the columnar file instance instantiated during the program's check for file existence. Next, it creates or accesses NUMCOLUMNS heap files, reads metadata to determine if there is available space and its location. For each column, it calculates the offset, inserts data into the proper place, and obtains record IDs which are used later in queries. After obtaining record IDs from all columns, it creates a tuple ID containing the number of columns, heap page ID, and record ID, which is then inserted into the columnar file.

As per the design rationale, this method of batch insertion is considered inefficient. A more optimal approach involves opening a heap file, inserting one column at a time, generating record IDs, storing them in memory (if space permits, contingent upon system capabilities), and subsequently creating combined tuples for insertion into the columnar file. This method significantly reduces the number of reads and writes, resulting in a more efficient batch insertion process.

Index Program

The Index Program provides users with the capability to create indexes on columns of their choosing within the database, predominantly offering two index types: 1) Btree and 2) Bitmap. Execution of the command necessitates the user to navigate to the test folder of the Minijava database.

To create an index, users are provided with the following command structures:

1) For BTREE index: make index COLUMNDBNAME=db COLUMNARFILENAME=file COLUMNNAME=C INDEXTYPE=BTREE

2) For BITMAP index: make index COLUMNDBNAME=db COLUMNARFILENAME=file COLUMNNAME=C INDEXTYPE=BITMAP

Upon execution within the test folder, the Makefile orchestrates the following sequence of commands:

```
Index: Index.java
   $(JAVAC) Index.java TestDriver.java

index: Index
      $(JAVA) tests.Index $(COLUMNDBNAME) $(COLUMNARFILENAME) $(COLUMNNAME) $(INDEXTYPE)
```

This command necessitates four arguments to be passed through the Command Line Interface, otherwise, the Makefile will not execute the index program.

Required Arguments:

1) **COLUMNDBNAME:** This denotes the name of the database where the columnar file and table exist.
2) **COLUMNARFILENAME:** This signifies the name of the column file containing the data.
3) **COLUMNNAME:** This refers to the name of the column on which the index is built.
4) **INDEXTYPE:** This specifies the type of index, either **BTREE** or **BITMAP**.

Upon execution with proper arguments, the program follows the sequence below to generate the index:

1. The program first checks if the correct number of inputs is passed via the command line interface. If not, it prompts the user to provide the correct number of arguments.

2. After validating the arguments, the program opens the database using the inputted COLUMNDBNAME. If the database does not exist, the program terminates with the message "DB does not exist." It appends the ".minibase-db" extension to the COLUMNDBNAME input, requiring users only to provide the database name without the extension for proper program execution.

3. Subsequently, the program attempts to open the COLUMNARFILENAME columnar file. If the file does not exist, the program terminates with the message "Columnar File COLUMNARFILENAME does not exist."

4. The program initializes several objects based on the COLUMNNAME input:
- **Column Index**: Obtained from metadata stored in the columnar file, necessary for opening the heap file containing column data.
- **Column Attribute Type**: Also retrieved from the columnar file metadata, required for Bitmap creation. Retrieving this information directly from the columnar file ensures accuracy.
- **Value Class**: Created from the retrieved Column Index and Attribute Type, which is essential for Bitmap creation.

5. The program performs a sanity check, ensuring that the column exists. If the column does not exist, the program terminates with the message "Column Doesn't Exist. Provide the Column Name which exists in the table."

6. If the column exists, the program calls the createBTreeIndex() method with the columnIndex or createBitmapIndex() with the columnIndex and Value Class based on the INDEXTYPE provided by the user.

7. Upon successful index creation, the program flushes pages to complete writing data to disk and prints read & write counts for the program.

createBTreeIndex() Method:

This method creates a B-tree with the columnar file name and column index combination. Based on the column attribute type, it creates a maximum key size attribute, which is 5 for integers and the maximum available size for strings as defined in the Global Constant (50). Btrees do not support float types, resulting in an error. After creating the Btree file, the program scans the heap file for data insertion into the Btree map, closes the scan, and returns.

createBitmapIndex() Method:

This method is explained in detail in the Bitmap Index Scan section.

Query Program

The query program accepts input from the command line in the form *"query COLUMNDBNAME COLUMNARFILENAME [TARGETCOLUMNNAMES] VALUECONSTRAINT NUMBUF ACCESSTYPE".* Value Constraints lists the selection criteria and takes the form *"{COLUMNNAME OPERATOR VALUE}"*. These arguments correspond to the name of the database, the name of the columnar file in the database, the target columns to scan, the query constraints, the maximum number of buffer files that can be used, and the access type that should be used for the scan.

Initially, it parses the arguments, checks if the Database exists, initializes the buffer manager using NUMBUF, and fetches the desired columnarFile. For the sake of modularity, query processing for each of the four access types are divided into their own methods. These methods are called using a switch statement on the access type argument. At the end of each query processing method, the number of reads and writes are displayed. These are discussed and analyzed in the

The filescan query is implemented using the columnFileScan (as referenced in the File & Index Scan section). A projection was built with the target columns to get a map of what input fields go where into the output tuple. The value constraint argument - consisting of the aforementioned column name, operator, and key value - was parsed to create the selection condition expression. The first operand of the selection is a symbol, based on an inner join to the target column. The second operand attribute type is assessed and then assigned based on whether it is a float, integer, or string. These - along with a list of attribute types, the total number of columns in the input and output tuples, and the file name - are passed to create a new ColumnarFileScan object. The tuples matching the query are fetched using the get_next() method and printed.

Column scan query access is implemented similarly with the openColumnScan method within the columnarFile. This initializes a sequential scan (implemented using a heap file) on the page

corresponding to the target column. The column is then fetched, the values within the column are checked against the constraint value, and matched tuples are printed using scan.getNext() in a loop. With queries with column scan access, the current implementation returns the entire selected value constraint column. Future refactoring is needed to parse based on the given operator and return each of the targeted columns when such is greater than one.

When calling a query using BTree access, a BTree index must be created first. This is done using a method implemented in the columnar package. An index is created on each column, creating a BTreeFile with the naming scheme of columnarFileName.btree{column number}. The index passed on the target column (listed in value constraints) is retrieved using this naming convention. Subsequently, a BTree scan object is created with a corresponding Int/StringKey based on the constraint value. Various if statements assign the low and high key values for the scan based on the constraint operator. At the moment, BTree queries that rely on StringKeys are partially functional. It fails on queries that attempt to return strictly greater or less than the key. This is because a method to increment or decrement string keys was not identified in time.

The implementation of Bitmap is still outstanding. Proper implementation of the bitmap queries relies on the bitmap index scan class (refer to bitmap index section).

Query_Delete Program

The Query delete program (called using *query_delete*) takes in the same arguments as the query program with the addition of *PURGEDB* which is a boolean representing whether all previously deleted tuples should be purged from the database or not.

Each access type query is performed the same way with slight modification. Instead of printing the retrieved tuples, the TID is used to mark each result as deleted. At the end of the query processing, if the PurgeDB boolean is true, then the purgeAllDeletedTuples method is called from the columnar file.

In the current iteration of the columnar DBMS, the query delete throws a page-pinned exception on BTree-based queries when attempting to flush the buffer pages.

Much of the query and query delete programs have pending implementations. As we had to redistribute workload due to team dynamic changes, it was not prioritized and its implementation relies heavily on other classes.

Program Results

After all the changes outlined above were complete, the programs were executed. The analysis of their performance was expected to compare the read and write performance of the different indexing types implemented.

Test Set

The Batch Insert program tested on a given sample dataset with 49,999 records and four columns.

The Index program tested on a given sample dataset with 49,999 records db with C and A columns.

The query and query delete were tested using the first 24 tuples from the provided sample data. It consists of two string and two integer columns - A, B, C, and D respectively.

For the full input file and query parameters, refer to the appendix (A and B).

Test Results

Table 1.a summarizes the Batch Insert Page Read totals:

BATCH INSERT READS	
24 Tuples, 4 Columns, and no existing DB	3
49999 Tuples, 4 Columns, and no existing DB	2436596

Table 1.b summarizes the Batch Insert Page Writes totals:

BATCH INSERT WRITES	
24 Tuples, 4 Columns, and no existing DB	17
49999 Tuples, 4 Columns, and no existing DB	240553

Table 2.a summarizes the Batch Insert Page Read totals:

INDEX READS		
Tuple Count	BTREE	BITMAP
49999 Tuples	841	14,975,223

Table 2.b summarizes the Batch Insert Page Writes totals:

INDEX WRITES

Tuple Count	BTREE	BITMAP
49999 Tuples	2072	46930

Table 3.a summarizes the Query Page Write totals:

QUERY WRITES				
	ACCESS TYPE			
QUERY CONSTRAINT	**filescan**	**columnscan**	**btree**	**bitmap**
A = Connecticut	0	0	0	-
D > 6	0	0	0	-
C <= 10	0	0	0	-

Table 3.b summarizes the Query Page Read totals:

QUERY READS				
	ACCESS TYPE			
QUERY CONSTRAINT	**filescan**	**columnscan**	**btree**	**bitmap**
A = Connecticut	10	3	10	-
D > 6	10	2	11	-
C <= 10	10	2	11	-

Table 4.a summarizes the Query Delete Page Write totals:

QUERY DELETE WRITES				
	ACCESS TYPE			
QUERY CONSTRAINT	**filescan**	**columnscan**	**btree**	**bitmap**
A = Connecticut w/ purge	13	-	-	-
A = Connecticut w/o purge	3	-	-	-
C <= 10 w/ purge	-	-	-	-

Table 4.b summarizes the Query Delete Page Read totals:

QUERY DELETE READS	
	ACCESS TYPE

QUERY CONSTRAINT	filescan	columnscan	btree	bitmap
A = Connecticut w/ purge	13	-	-	-
A = Connecticut w/o purge	12	-	-	-
C <= 10 w/ purge	-	-	-	-

Results Analysis

Batch Insert Program:

The write count is the result of writing a record into an individual heap file, writing into a columnar file. So total write count should be around (number of columns +1) * number of records. For given sample data it 49,999 * 5 approximate considering buffer.

The read count is the result of continuous reads for finding free space into heap files for inserting data into heap files. Due to the huge number of pages the buffer can not keep the page in memory so when it tries to enter data into one column it releases the page and starts from the beginning which makes read count exponentially.

Which also explains, the initial 25,000 to 30,000 records are swiftly inserted within the first minute. However, the insertion process slows down significantly for the remaining records. This delay is attributed to the necessity of scanning and writing the initial records, thereby impeding the overall insertion speed. Exact reads and writes counts are given in the result 1.a & 1.b table.

Index Program:

1) **BTREE Index:**

For Btree index creation, the program first reads the columnar file for metadata, then scans through the heap file for data insertion into the Btree map. It reads all the pages with the particular column, resulting in a read count for all heap pages and one columnar file. For the given sample data with 49,999 records and an integer type column C, the read count is 841. The read count may increase for string data types as they occupy more space in file pages, necessitating more file access. The write count occurs because the program inserts data into the Btree, updates keys in nodes, and, for the given implementation, the write count is 2072. A table in the results section details the read and write counts.

2) **BITMAP Index:**

For Bitmap index creation, the program calls Bitmap constructor which interacts with db to get heapfile pages, create heap files for Bitmap index. Same as Btree Index scans all the pages with a particular column, checking for duplicate RID to avoid multiple mappings to the same

value column, writing into bitmap files which result in huge read count. Write count on other hand is very small or equivalent to record number in column which in result table.

Exact read and write count outlined in table 2.a and 2.b.

Most notably and unsurprisingly the column scan, in comparison to each other access type, performs fewer page reads. Since columnar databases store each column contiguously by page, columnscan (which scans sequentially) may offer better performance for queries targeting specific columns or ranges of values within columns. The number of page accesses increases when linear with the number of target columns to return.

When executing a delete query that also purges the database of the deleted tuples, it is expected that the number of page writes is higher than if the database is not purged. For each tuple, the database has to be written over to remove them. As outlined in table 4.a, this is observed behavior (see query delete for table context). It is also expected to see a higher write count in access methods that maintain an index map (such as btree and bitmap) as these maps also have to be updated.

Conclusion

Analysis of the program results indicate that there is room for improvement across the board for all classes, packages, and methods introduced as part of the development effort for Phase 2 of this project. Some improvements could be simple, such as a possible reduction for batch insert to go tuple column-by-column instead of tuple by tuple. Other improvements could be more involved, such as needed overhauls in bitmap foundation in order to resolve the large amount of reads needed to insert unique mappings. It also goes without saying that there is functionality in this submittal of code that has been admitted to not working as intended.
Going forward, which areas of improvement will be tackled should be analyzed in a cost benefit analysis. The team will need to implement all the functionality that could not be achieved during phase 2 in order to continue forward for phase 3. As far as efficiency improvements, those will need to be considered in tandem with the expansions minibase will need for phase 3 and where efforts are best spent.

Bibliography

[1] K. Selcuk Candan, “Database Management System Implementation Phase 2”, ASU, Tempe, AZ, 2024
[2] Raghu Ramakrishnan and Johannes Gehrke. Database Management Systems. McGraw-Hill, 2002.

[3] Michael Stonebraker, Paul Brown, Dorothy Moore, and Carl Staelin. Minibase: A self-configuring database system. ACM SIGMOD Record, 20(2), 1991

Appendix

This section exists if we want to include anything we want to refer to without inserting it directly into a section of the report.

This might be a good section to outline what input set(s) we tested the programs on for our analysis

Appendix A - Query Test Data

A:char(25)	B:char(25)	C:int	D:int
South_Dakota	West_Virginia	6	3
Connecticut	Delaware	8	8
New_Hampshire	Kansas	2	1
Connecticut	Puerto_Rico	4	9
Montana	District_of_Columbia	2	8
Vermont	West_Virginia	8	6
New_Hampshire	District_of_Columbia	6	6
Kentucky	Singapore	1	0
Connecticut	Indiana	2	8
Delaware	Singapore	8	6
Colorado	Connecticut	9	0
Vermont	Puerto_Rico	3	3
Puerto_Rico	Business	0	4
Indiana	West_Virginia	1	5
Business	Vermont	0	4
Nevada	Oklahoma	0	7
Nevada	District_of_Columbia	8	3
Puerto_Rico	Indiana	2	1
Montana	District_of_Columbia	6	9
South_Dakota	Singapore	2	0
North_Dakota	Singapore	4	2
Colorado	District_of_Columbia	0	6
Oklahoma	Oklahoma	7	1
Delaware	Vermont	3	9

Appendix B - Queries and Responses

```
make query COLUMNDBNAME=testDB COLUMNARFILENAME=testFile TARGETCOLUMNNAMES=[A,B,C,D]
VALUECONSTRAINT='{D \> 6}' NUMBUF=4 ACCESSTYPE=BTREE
cd tests; make query
make[1]: Entering directory '/afs/asu.edu/users/t/a/l/talott/cse510g11/minjava/javaminibase/src/tests'
/usr/bin/javac -classpath .:.. QueryProgram.java
/usr/bin/java  -classpath .:.. tests.QueryProgram testDB testFile [A,B,C,D] {C \> 6} 4 BTREE
Open the DB
```

```
Replacer: Clock

File exists
Executing Btreescan query...
testFile.btree3
testFile.btree3
[Oklahoma, Oklahoma, 7, 1]
[Connecticut, Delaware, 8, 8]
[Vermont, West_Virginia, 8, 6]
[Delaware, Singapore, 8, 6]
[Nevada, District_of_Columbia, 8, 3]
[Colorado, Connecticut, 9, 0]
Number of disk pages read: 11
Number of disk pages written: 0

make query COLUMNDBNAME=testDB COLUMNARFILENAME=testFile TARGETCOLUMNNAMES=[A,B,C,D]
VALUECONSTRAINT='{C \<= 10}' NUMBUF=4 ACCESSTYPE=COLUMNSCAN
make[1]: Entering directory '/afs/asu.edu/users/t/a/l/talott/cse510g11/minjava/javaminibase/src/tests'
/usr/bin/javac -classpath .:.. QueryProgram.java
/usr/bin/java  -classpath .:.. tests.QueryProgram testDB testFile [A,B,C,D] {C \<= 10} 4 COLUMNSCAN
Open the DB
Replacer: Clock

File exists
Executing Columnscan query...
6
8
2
4
2
8
6
1
2
8
9
3
0
1
0
0
8
2
6
2
4
0
7
3
Number of disk pages read: 2
Number of disk pages written: 0

make query COLUMNDBNAME=testDB COLUMNARFILENAME=testFile TARGETCOLUMNNAMES=[A,B,C,D]
VALUECONSTRAINT='{A = Connecticut}' NUMBUF=4 ACCESSTYPE=FILESCAN
```

```
cd tests; make query
make[1]: Entering directory '/afs/asu.edu/users/t/a/l/talott/cse510g11/minjava/javaminibase/src/tests'
/usr/bin/javac -classpath .:.. QueryProgram.java
/usr/bin/java  -classpath .:.. tests.QueryProgram testDB testFile [A,B,C,D] {A = Connecticut} 4 FILESCAN
Open the DB
Replacer: Clock

File exists
Executing Filescan query...
Connecticut
[Connecticut, Delaware, 8, 8]
[Connecticut, Puerto_Rico, 4, 9]
[Connecticut, Indiana, 2, 8]
Number of disk pages read: 10
Number of disk pages written: 0


make query_delete COLUMNDBNAME=testDB COLUMNARFILENAME=testFile
TARGETCOLUMNNAMES=[A,B,C,D] VALUECONSTRAINT='{A = Connecticut}' NUMBUF=4
ACCESSTYPE=FILESCAN PURGEDB=true
cd tests; make query_delete
make[1]: Entering directory '/afs/asu.edu/users/t/a/l/talott/cse510g11/minjava/javaminibase/src/tests'
/usr/bin/javac -classpath .:.. QueryDeleteProgram.java
/usr/bin/java  -classpath .:.. tests.QueryDeleteProgram testDB testFile [A,B,C,D] {A = Connecticut} 4 FILESCAN true
Open the DB
Replacer: Clock

Connecticut
Deleted Tuples:
[Connecticut, Delaware, 8, 8]
[Connecticut, Puerto_Rico, 4, 9]
[Connecticut, Indiana, 2, 8]
Number of disk pages read: 12
Number of disk pages written: 13
```