1. Introduction of the bitmap indexes.

A bitmap index is basically an algorithm to flag a bit. In other words, 0 means false, and 1 means true for certain keys. For example, assume that we have 5 records such that

EmployeeID	Gender	Department
10000001	M	1
10000002	M	2
10000003	M	3
10000004	F	1
10000005	F	3

Then, the bitmap index for gender for example, we have 2 bitmap indexes such that M = (1,1,1,0,0) and F = (0,0,0,1,1). Depending on the value that we have on the records as the record positions are fixed, we can determine whether a key has appeared or not. We have a bitmap index for gender with the following vector format.

Key	Vector
M	11100
F	00011

The same thing for the Department, we have 3 keys in the bitmap indexes, where 1= (1,0,0,1,0), 2= (0,1,0,0,0) and 3= (0,0,1,0,1). In this case, the bitmap index for the EmployeeIDis not very efficient, since we have 5 different values for EmployeeID.

In order to implement bitmap indexes, we assume that the order of the records are fixed. It means that once we create the bitmap, the corresponding bits are located in the same position in the records so that we can trace where the specific record is located. Otherwise, we cannot use bitmap indexes. For example, using the same records above, when we get the table, if the order of records are not the same, i.e. EmployeeID 10000001 supposed to appear at first row, but it appeared second row, then, the bitmap index for EmployeeID is not going to work.

In terms of this project, bitmap indexes for Gender and Department are quite efficient. Since gender has only 2 possible keys, we will have only 2 possible bitmap indexes, and for the department, we will have only 10 bitmap indexes. However, for the case of employee ID, since the total possible length of employee ID is 8 digits, which means we can have 10^8 possible keys are possible when we have more than 1 million records and if keys are equally distributed. In our case, some records are duplicated, and the total bitmap index size could be reduced, but it is not very efficient. In terms of performance, please refer to the performance section but bitmap indexes algorithms for employee ID are very much slower than gender or department. However, in terms of size, compressed employee ID bitmap indexes can be considerably smaller compared to uncompressed employee ID bitmap indexes because if there is rare chance to have a duplicate, then most of bits are going to be 0 and only when it appears, it will turn to 1 so

by algorithm of compressed bitmap, the size will be smaller.

In terms of compressed bitmap indexes, compressed bitmap indexes are using how many consecutive 0 or 1 are appeared. Compressed bitmap indexes have two parts, which is unary that shows the length of data part, and binary part that shows the data. For example, when we have 11101101, in terms of the unary part, we select the first 0 that appears, so we know that 1110 is the unary part and this tells that the next 4 bits are used as the data part. Data part is 1101 so if we convert this binary number to integer, it becomes 13. This means we have 13 0's and 1 is followed after. Using this, for example, if we have 11101101 00 1011, as I show previously, 11101101 translated in to 13 digits of 0, and there are 0 at the first bit, which means 1 is followed right after, and then, it shows 1011, so there are 3 more 0's and then 1 is followed. The result is therefore, 0000000000000110001

In this project, we are required to develop a technique for removing duplicates using bitmap indexes; however, we also need to consider the date of each record. As we have 10 MB limits, we cannot store all information in a memory for the large size of data. If this is the case, we need to scan the file again to get the date and we only keep the recent ones.

We also need to sort the records based on employee ID as required additionally. If the bitmap indexes are within the memory size, we can easily use hashmap collections in Java and sort it quickly; however, since we have 10MB restriction, the best way to sort is using Two-Phase Multiway Merge Sort (TPMMS), which we implemented in our TPMMS project.

2. Implementation of the bitmap indexes.

For our implementation of this project, we use arguments to identify input and output files. For instance, <<-i C:\dev\dat\sample_01.txt C:\dev\ dat\sample_02.txt -o C:\dev\dat\sample_output.txt>> commands determines the 2 input files with argument -i followed, and -o for output file.

2.1 Files from TPMMS project

There is an Employee class, which stores the object of Employee when we read from the file, and this only store 4 attributes, ID, Gender, Department, and Date. Because if we separate all attributes such as address or department, the performance becomes too slow (around 2 times slower), so we decided to reduce it to have only 4 attributes. There are essential attributes for creating bitmap indexes which is ID, Gender and Department. In addition to this, we need Date attributes for removing duplicate.

EmployeeFile java file is used for storing static values, such as the size of Date, size of ID, Block Size, … etc. EmployeeFileReader file helps to read the records from the given data sets and EmployeeFileWrite file helps to write back to make temporarily sorted sublists or to make a resulting output file. They both use BufferStream (BufferInputStream and BufferOutputStream) so that we could emulate interaction with hardware and read/write with given input or output. Both files keep track of how many records, Disk I/O, and Block I/O there are.

ExecTimer is used for recording the processing time and file generator is making sample data as explained above.

3.2 Newly added file for this project.

In addition to files from 3.1, we have added more files. Please note that since bitmap indexes are much more complex than TPMMS project, we have created many more classes.

In terms of bitmap index, we have implemented a BitSet of Java interface, which allows us to have a vector of bits. Therefore, this increases a lot of spare spaces compared to other data structures. Using this, we have an IndexBitSet class, which allows us to have a number of bits as BitSet class utilizes leftmost non-zero bit in calculation of bitset length.

We have EmployeePostion file which gets ID and help to build a BitSet in order to create bitmap.

3.3 Bitmap index creation

We are using two ways of creating bitmap indexes. If the bitmap indexes can fit in one run, in other words, if we can write them all in a main memory, then, we directly create bitmap indexes using main memory. The method is use in the main file, called CreateDenseBitMapIndex() in main method. Basically, we have used IndexBuilder class and then using IndexBitSet, we can create bitmap indexes. Basically, all department and genders in sample file can be used by this method, since the number of possible keys is much smaller than EmployeeID. Also, we have tested that it works for 1M tuples as well. The result of bitmap indexes will be a hashmap data structure with key – IndexBitSet relationship. Once this job is done, we create bitmap indexes and calculate the size of uncompressed bitmap indexes and compressed bitmap indexes.

However, the later is not feasible, then we need to apply method involving multiple runs. In order to do that, we create the list of each employee id and position in the record file. Using the PositionBuilder class, with the method BuildPseudoIndex(). For example, assume that we have the original tuples such that

249262252013-01-01Ynes Puttergill 0003325918281 Clarcona FL 32710 South
135210612013-02-01Noland Iacobo 0009214875375 Lochgelly WV 25866 South
724161182013-03-01Jan Fass 0008418304933 Lilburn GA 30048 South
135210612013-04-01Lorin Josham 1007121946888 New York City NY 10211 Northeast
735954162013-05-01Linn Luscott 1005419600217 Lock Haven PA 17745 Northeast
135210612013-06-01Robinia Jaycox 0009720978012 Hanlontown IA 50444 Midwest
295832512013-07-01Willa Andrus 0004131320092 Woodsfield OH 43793 Midwest
590022112013-08-01Jen Hebburn 0010691505122 Bloomfield NM 87413 West

And after running BuildPseudoIndex() method, we have a following output such that

0 24926225 0
0 13521061 1
0 72416118 2
0 13521061 3
0 73595416 4
0 13521061 5

0 29583251 6
0 59002211 7

The output is separated by space, and the first one is telling you whether or not it is processed, in other words, is positions for employee ID is recorded or not. The second one indicates the EmployeeID. And the third one is the position of the employeeID. For example, 24926225 appeared at the first row so we assigned as 0. 13521061 is appeared at the second row, … etc.

Once this job is done, we concatenate or merge all indexes. From PositionBuilder class, MergePositionIndexes() method exists, which can result from the previous example such that

1 24926225 0
1 13521061 1;3;5
0 72416118 2
0 73595416 4
0 29583251 6
0 59002211 7

So, the first column becomes 1 for two rows, it means that these employee ID is processed. As a result, for 13521061, we get all the positions concatenated. The positions are separated by semi-colon in our case. This is called iteration 1. And as we keep continuing iteration, we will have a following results.

Iteration 2:

1 24926225 0
1 13521061 1;3;5
1 72416118 2
1 73595416 4
0 29583251 6
0 59002211 7

Iteration 3:

1 24926225 0
1 13521061 1;3;5
1 72416118 2
1 73595416 4
1 29583251 6
1 59002211 7

From there, we now know all the employee ID's positions, then we can create a bitmap index. From PositionBuilder class, we have BuildBitMapIndex() method. This will result such that

24926225 10000000
13521061 01010100
72416118 00100000

73595416 00001000
29583251 00000010
59002211 00000001

We have saved them as a compressed bitmap index as well. But as a requirement, we only calculate size of compressed and uncompressed version of it. At this point, requirement 1 and 2 are already done. The timestamp and the size will be computed and display in the java console.

4. Remove duplicate and merge the records using bitmap indexes

From created bitmap indexes, in order to speed up, we are going to use the position information that we have created from 3.3. Also, we need to have date information so that we know which one is more recent compared to other records which have the same employee ID.

And then, using bitmap indexes, we get all the possible dates from the same employee ID. For example,

24926225 10000000
13521061 01010100
72416118 00100000
73595416 00001000
29583251 00000010
59002211 00000001

24926225 has only 1 record, so we do not need to remove duplicate; however, for the case of 13521061, we need to get comparing with dates. From extracted dates, we compare which one is the most recent, and then we remove the tuples which is not the most recent from the original file and save it to temporary file.

It means that we have another temporary file for dates such that

2013-01-01
2013-02-01
2013-03-01
2013-04-01
2013-05-01
2013-06-01
2013-07-01
2013-08-01

Please note that actual temporary file for dates does not have dash in order to save the file size, and I show it here in order to understand how it works easily. For the case of 13521061, we have 1 bit for 2nd, 4th and 6th, so we take the values in date temporary files of this position. As a result, we have 2013-02-01, 2013-04-01, and 2013-06-01. Since 2013-06-01 is the most recent one, we delete 2nd and 4th rows, and we will have a resulted output with,

249262252013-01-01Ynes Puttergill 0003325918281 Clarcona FL 32710 South

724161182013-03-01Jan Fass 0008418304933 Lilburn GA 30048 South
735954162013-05-01Linn Luscott 1005419600217 Lock Haven PA 17745 Northeast
135210612013-06-01Robinia Jaycox 0009720978012 Hanlontown IA 50444 Midwest
295832512013-07-01Willa Andrus 0004131320092 Woodsfield OH 43793 Midwest
590022112013-08-01Jen Hebburn 0010691505122 Bloomfield NM 87413 West

We keep processing until the last bitmap index has been passed. This job is done in BitmapIndexDeDup class with process() method.

There is also an important notation which determines how many index limits per round. Since we have only 10MB of main memory, the bitmap index cannot be stored all in by one run. So depending on the bitmap size, we have decided to limit of indexes. For the actual limit indexes, you can refer to the Figure 2. In this case, we have also recorded disk I/O and how long it takes to do it as well.

And from the resulted output, we observe that the resulted output is not ordered. Therefore, we need to sort this result with any algorithms. For that, please refer to following section, or section 5.

5. Sorting Algorithms

We have used the TPMMS algorithms from TPMMS project; however, we eliminate the duplication remove process as we already remove duplicates from the previous part. As the result is observed, the sorting algorithms do not take too much time compared to remove duplicates using bitmap.

Phase 1

1. Tpmms.Process() in loop reads records from input data files - one record at a time. Here record granularity is an abstraction provided by EmployeeFileReader.ReadRecord(); internally EmployeeFileReader reads data from disk in blocks - one block at a time Please note that there is no means to directly control hardware, particularly from within Java.
2. Every record read in (1) is placed into memory - into a Tree-like structure (TreeMap), where ID is used as a key. Tree assures that all records within it are in sorted order. As there can be more than one record with given ID records are placed within the array in a Tree. In essence, this is a MultiMap. Whenever the number of records in the tree is about to exceed maximum memory size for the TPMMS list, the content of a tree is saved into a file and the tree is freed.
3. In the end, all the input (-i attributes) files are saved into temporary ordered list files of specified maximum size.

Phase 2

1. Initially, the first record of each list file is read. Here again - EmployeeFileReader is the only abstraction that hides implementation in which reading is done at block granularity.
2. Every read record is placed into a priority queue. Priority queue uses a customized

comparator EmployeeComparator which ensures that records in the queue are ordered by ID. It is important to note that records saved in the queue are wrapped into EmployeeQueueContext structure which bedside record itself contains a reader (EmployeeFileReader) associated with list-file generated in Phase 1 (2).

3. After the priority queue is initialized with the first record of every list, queue processing loop commences. In the loop, on each iteration, a head element of the priority queue is popped and saved into a variable holding a current record (lTopValue). Additionally, there is another variable that holds record read on previous iteration (lLastTopValue) and is initially initialized to NULL. Then, the current record is compared with a record read on the previous iteration - if ID_previous<ID_current then the previous record is written to the final result; In case if lLastTopValue is NULL it is simply initialized with lTopValue. At the end of the loop reader reference from lTopValue is used to read another record from the corresponding list. This record (wrapped into EmployeeQueueContext) is pushed to the priority queue. Loop runs till priority queues are not empty. At the end of the process, you will see the output file with all sorted data.

From the previous result, we get following result after TPMMS sorting algorithms,

135210612013-06-01Robinia Jaycox 0009720978012 Hanlontown IA 50444 Midwest
249262252013-01-01Ynes Puttergill 0003325918281 Clarcona FL 32710 South
295832512013-07-01Willa Andrus 0004131320092 Woodsfield OH 43793 Midwest
590022112013-08-01Jen Hebburn 0010691505122 Bloomfield NM 87413 West
724161182013-03-01Jan Fass 0008418304933 Lilburn GA 30048 South
735954162013-05-01Linn Luscott 1005419600217 Lock Haven PA 17745 Northeast

The result is now sorted. For more test cases, please refer to the next section, or section 6.

6. Test results/ test cases,

We have tested that there are 20,000, 50,000, 100,000, and 200,000 tuples as their record sizes. More specifically, for 20,000 tuples, it means we have a total sample size of 20,000, and it could be 10,000 + 10,000, or any other combinations. For our report, we divide them evenly. In sample size on Figure 1, shows two 10,000 which have different file numbers and the total of them are 20,000 tuples. For more than 1 million tuples, since we have limited with 10MB main memory restriction, each run can only fit 25 of bitmap so this makes it too slow compared to other operations. On the other hand, if we compare to 20,000 tuples only, we have smaller bitmap indexes which we can fit more values in the main memory, so this makes removing the duplicate process much faster. For more results, please refer to Figure 2. Please note that in the column name, there is ID (A), and ID (C). (A) means the size of uncompressed or actual bitmap indexes; on the other hand, (C) means the size of compressed bitmap indexes.

As we observe in Figure 1,bitmap size becomes too large for employee ID, since there are many possible different keys for bitmap indexes as there are not a lot of duplicates. However, in this case, compressed bitmap indexes are the best since there will have a lot of 0’s and this gives much smaller size in employee ID for compressed bitmap indexes. On the other hand, for gender, there are only 2 possibilities (0 or 1) and compressed bitmap indexes actually make a

larger size than uncompressed bitmap indexes. For department, even though there are 1000 possibilities on this field, but given example data has only 10 distinct department, so there will have 10 bitmap indexes. Uncompressed is beneficial since it reduces as half size; however, it is not very beneficial compared to employee IDs as well.

In Figure 2, it shows the result of removing duplicate using bitmap. The time it takes in figure 1 is around doubling up as the sample size is doubling; however, removing duplicate process, it becomes more than double, and looks like it goes exponentially. Since we have more tuples sizes, each bitmap indexes will have large size as well. For example, for the size of 10,000 tuples, it will have a bitmap index size with 10,000 bits for 1 key; however, if the size is 100,000 tuples, it will have a bitmap index size with 100,000 bits for 1 key. Therefore, As we observed, the index limit which means how many bit indexes we can pass by one run for 10 MB memory restriction becomes getting smaller and smaller. This means that for 2 million records, it will have only 25 index limit, and if we assume that duplicates are very rarely happens, then it requires to pass 2,000,000 / 25 = 80,000 runs to do the task, which makes impossible to observe the result.

In Figure 3, it shows the result of TPMMS algorithms for sorting, and this takes less time compared to any other process. For example, for 20,000 tuple size, phase 1 and 2 takes less than 0.1 seconds, whereas creating indexes or removing duplicate takes much more times compared to do this. We do not see significant different in terms of disk I/O for each phase, since there is no duplicate in each file (which already removed from previous steps). Also, in figure 3, total process shows. It means that from all steps which showed in figure 1, 2 and 3 are cumulative and tells the total processing time.

In Figure 4, it is the result in TPMMS project and I will compare this result in section 7.

Sample Size	File Number	Total Sample	Creating indexes									
			ID (A)	ID(C)	Time (ms)	Gender (A)	Gender (C)	Time (ms)	Dept (A)	Dept (C)	Time (ms)	Total Time (ms)
10,000	File 1	20,000	55,530,000	231,166	756	20,000	26,380	33	100,000	62,422	27	1,261
10,000	File 2		55,580,000	231,202	395	20,000	26,376	24	100,000	62,400	22	
25,000	File 1	50,000	368,175,000	622,372	1,394	50,000	65,948	59	250,000	156,136	60	2,492
25,000	File 2		368,550,000	622,480	862	50,000	65,944	56	250,000	156,062	58	
50,000	File 1	100,000	1,388,050,000	1,304,162	2,288	100,000	131,908	85	500,000	312,258	111	4,490
50,000	File 2		1,389,300,000	1,304,402	1,798	100,000	131,904	103	500,000	312,152	82	
100,000	File 1	200,000	5,550,800,000	2,726,242	5,288	200,000	263,818	244	1,000,000	624,528	236	10,325
100,000	File 2		5,558,000,000	2,726,802	4,238	200,000	263,814	161	1,000,000	624,342	155	

Legend A = Actual bitmap size
C = Compressed bitmap size
Please note that size of bitmap is all in bytes (i.e. ID (A) means actual bitmap size of employee ID which has size of X)

Figure 1 Creating index result

Sample Size	Index Limit	Supporting Files ID					Supporting Files ID					Supporting Files ID					Total
		Disk Read	Disk Write	Disk I/O	Record read	Record Write	Disk Read	Disk Write	Disk I/O	Record read	Record Write	Disk Read	Disk Write	Disk I/O	Record read	Record Write	Time (ms)
20,000	2,500	1,145	824	1,969	40,000	25,559	500	48	548	20,000	20,000	323	138	461	65,559	5,559	1,478
50,000	1,000	2,862	2,087	4,949	100,000	64,745	1,250	122	1,372	50,000	50,000	2,305	368	2,673	764,745	14,745	4,716
100,000	500	5,725	4,121	9,846	200,000	127,791	2,500	244	2,744	100,000	100,000	14,560	694	15,254	5,627,791	27,791	13,815
200,000	250	15,334	12,126	27,460	520,383	375,951	5,000	488	5,488	200,000	200,000	110,128	1,389	111,517	44,455,568	55,568	52,448

Figure 2 Removing duplicate result

Sample Size	Phase 1					Phase 2					Total Process Time
	Disk Read	Disk Write	Disk I/O	Record R/W	Time (ms)	Disk Read	Disk Write	Disk I/O	Record R/W	Time (ms)	
20,000	138	138	276	5,559	94	138	138	276	5,559	60	2.927 sec
50,000	368	368	736	14,745	192	368	368	736	14,745	139	7.592 sec
100,000	694	694	1,388	27,791	350	694	694	1,388	27,791	265	19.005 sec
200,000	1,389	1,389	2,778	55,568	652	1,389	1,389	2,778	55,568	502	64.077 sec

Figure 3 Sorting using TPMMS result

Memory (MB)	Sample Size	Phase 1							Phase 2							Total Time (s)
		Blocks Read	Disk Read	Blocks Write	Disk Write	Disk I/O	Num of Runs	Time (s)	Blocks Read	Disk Read	Records write	Block Write	Disk Write	Disk I/O	Time (s)	
10	**20,000**	500	500	500	500	1,000	4	**0.412**	500	500	5,553	138	138	638	**0.073**	**0.485**
10	**50,000**	1,250	1,250	1,250	1,250	2,500	6	**0.792**	1,250	1,250	14,745	368	368	1,618	**0.214**	**1.006**
10	**50,000**	2,500	2,500	2,500	2,500	5,000	12	**1.295**	2,500	2,500	27,791	694	694	3,194	**0.413**	**1.708**
10	**200,000**	5,000	5,000	5,000	5,000	10,000	22	**2.145**	5,000	5,000	55,508	1,387	1,387	6,387	**0.524**	**2.669**
10	**1,000,000**	25,000	25,000	25,000	25,000	50,000	104	**9.569**	25,000	25,000	277,120	6,928	6,928	31,928	**2.543**	**12.112**
10	**2,000,000**	50,000	50,000	50,000	50,000	100,000	206	**20.580**	50,000	50,000	553,308	13,832	13,832	63,832	**5.296**	**25.876**
10	**10,000,000**	250,000	250,000	250,000	250,000	500,000	1030	**111.200**	250,000	250,000	2,729,857	68,246	68,246	318,246	**36.950**	**148.150**

Figure 4 result for TPMMS project

7. Comparing the results with TPMMS

Since This project does not require to compare the result with 20 MB, we directly compare with 10MB section in TPMMS project. As mentioned in section 6, if we increase the size of tuples such as millions of tuples, because of 10 MB memory restriction, the number of available indexes per run becomes too small and this makes to run a lot of time. However, in TPMMS project, we have a result showing with 1 million, 2 million and 10 million with 12.112 seconds, 25.876 seconds, and 148.150 seconds which make a reasonable processing time.

Even smaller numbers of tuples, genuine TPMMS algorithms are much faster in TPMMS project result. For example, for the sample size of 20,000 tuples, using bitmap indexes takes total 2.927 seconds whereas in genuine TPMMS takes only 0.485 seconds. The result for 50,000 tuples, 100,000 tuples, and 200,000 tuples are showing that TPMMS are much faster than bitmap indexes process.

However, as we use the TPMMS sorting algorithm for bitmap index as of section 5, if we genuinely compare the result of TPMMS algorithm only, TPMMS in bitmap index is slower. This is because in bitmap index, we do not need to remove duplicate unlike in TPMMS project. This make it the time spent only for TPMMS less time. Bitmap indexes are not the best tool for eliminating duplicates and that they are more suitable for operations on columns with low cardinality.