

**COMP - 6521**

**Advanced Database Technology and Applications**

**Project Report**

**On**

**Lab Assignment 2: Bitmap Indexing**

**Professor**

**Dr. Nematollaah Shiri**

**Team Members**

|  |  |  |
| --- | --- | --- |
| **Student Name** | **Student Id** | **Email Id** |
| Yash Pandya | 40119272 | y\_pandy@encs.concordia.ca |
| Himen Sidhpura | 40091993 | h\_sidhpu@encs.concordia.ca |
| Sucheta Sudhakumari | 40080543 | s\_ijaya@encs.concordia.ca |

**Index**

1. Bitmap indexing 3
2. Program Description 3
3. Steps to run the program 4
4. Algorithm 4
5. Experiment Results 6
6. Results of TPMMS 8
7. Comparison of results to LA1(TPMMS) 10
8. Coding Standard 10
9. Class Description 10
10. Group Member Contribution 11
11. Results 11
12. **Bitmap Indexing**

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

Indexes are primarily built on specific keys /fields of the data base even though some databases extend the power of indexing by letting developers create indexes on functions or expressions. A bitmap index is built on table attributes such that bitmap index for a field F is a collection of bit-vectors of length n, one for each possible value that may appear in the field F. The vector for value u has 1 in position i if the ith record has v in field F, and it ha5 0 there if not.

The number of bit vectors created for a field will be equal to the number of distinct values for that field and the length of the bit vector will be equal to the number of rows/tuples in a table.

1. **Program Description**

**Phase 1 –Program Initialization:** Program execution is triggered by the ProgramController class. Constant values including block size and file I/O paths are initialized. The input files are ready block by block into the program, starting with T1 followed by T2.

**Phase 2 – Creation of bitmap index and sorting index:** A 2D array of data type: long and size : (number of tuples )\*(number of tuples +1 ) is created to store the bit vectors .The first column holds the distinct values of the field while the remaining columns signify the bits corresponding to each tuple. The row size of the array is set to be equal to the number of tuples, assuming the maximum row numbers possible in case no field values are repeated across the block. Every time a unique value is read, the 1st column in the most recent row is set to that value after which the corresponding bit vector is set to 1. The key values are then sorted using quicksort and the bitmap index for each block is then written into a sublist.

**Phase 3 – Merging index blocks:** The block wise bitmap index sublists, corresponding to each file are merged recursively into a single file using TPMMS technique. At the end of this point, we get 6 separate index files corresponding to the 3 fields and 2 files. For each distinct value of the index, the columns with bit vector 1 indicate duplicates tuples.

**Phase 4 – Compression of Index:** Each of the index files are then compressed using run length encoding algorithm.  
  
**Phase 5 – Merging the files:**  The bitmap indices of employee file are read into the memory using TPMMS algorithm. For each unique employee number, corresponding tuples are located from data file and checked for duplicates. In case of duplicates, the data is sorted using date and the latest tuple is then written into the output file.

1. **Steps to run the program**
2. Import the project Bitmap-Indexing to the IDE.
3. Set constant values including the block size, memory allocation size, I/O file paths in the constants.java class.
4. Run the program from ProgramController.java class.
5. The execution will be performed phase by phase and results will be printed on the console.
6. **Algorithms**

**PHASE - 1: Program initialization:**

* Start.
* Allocate main memory.
* Allocate block size, file I/O paths and other utility constants.

**PHASE - 2: Bitmap Index Creation:**

* Start
* Read the file block by block.
* For each block, create a 2D array of size equal to the number of tuples in a block, to store the bitmap indices for the corresponding field.
* For each unique field value read, add a row to the array.
* For each field value read (new or existing in the array), set the corresponding bit vector to 1 in the array.
* Add the key value into a list of key values already read.
* Sort the key list.
* Write the bit vectors for each block into a file, fetching tuples in sorted order (This creates a sorted index).
* Continue till all blocks are read.
* Repeat the steps for indexes on other fields on the same file as well as on file T2.
* Stop

**PHASE – 3: Merging Bitmap Index blocks:**

* Start.
* Read bitmap index sublists of the same type, file by file.
* Taking 2 indices at a time, merge them into a single file .
* Repeat the process until a single index file remains for each index type.
* Identify duplicates using the index and eliminate them from the data file.
* Repeat the steps for indexes on other fields as well as for file T2.
* Stop

**PHASE – 4: Compression of Bitmap indices:**

* Start.
* Read bitmap index blocks one by one into the program.
* Using Run Length Encoding technique perform index compression.
* Repeat the steps for indexes on other fields as well as for file T2.
* Stop

**PHASE – 5: Merging the data files:**

* Start.
* Using TPMMS technique, read the index for employee field into the program.
* For each employee IDs read, identify the indices with bit vector 1 which correspond to duplicate tuples.
* Find the corresponding block of data from the data blocks.
* Scan the data block and locate all tuples with the same employee ID, adding them to a list.
* Sort the list according to date and write the latest one to the output file.
* Repeat the steps until the end of both index files.
* Stop

1. **Experiment Results:**

The experiment was conducted by changing the input file size, the result of which has been consolidated below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Memory Size = 10mb  Block Size = 40 | | | | | | |
| **Tuple Count** | **20,000** | **50,000** | **1,00,000** | **20,000** | **50,000** | **1,00,000** |
| **T1 File** | | | | **T2 File** | | |
| **Gender Bitmap Index** | | | | | | |
| Time taken to generate sublists | 230 ms | 317 ms | 806 ms | 352 ms | 421 ms | 498 ms |
| Time taken for Merge Data | 22 ms | 36 ms | 96 ms | 40 ms | 36 ms | 70 ms |
| Read Count | 248 | 608 | 1210 | 248 | 608 | 1210 |
| Write Count | 86 | 206 | 408 | 86 | 206 | 408 |
| Total Time for Bitmap index | 252 ms | 407 ms | 902 ms | 392 ms | 457 ms | 568 ms |
| Time taken for compression | 4 ms | 8 ms | 16 ms | 4 ms | 10 ms | 16 ms |
|  |  |  |  |  |  |  |
| **Department Bitmap Index** | | | | | | |
| Time taken to generate sublists | 279 ms | 654 ms | 1062 ms | 254 ms | 349 ms | 568 ms |
| Time taken for Merge Data | 75 ms | 87 ms | 172 ms | 43 ms | 61 ms | 183 ms |
| Read Count | 920 | 2240 | 4450 | 920 | 2240 | 4450 |
| Write Count | 430 | 1030 | 2040 | 430 | 1030 | 2040 |
| Total Time for Bitmap index | 354 ms | 741 ms | 1234 ms | 297 ms | 410 ms | 751 ms |
| Time taken for compression | 9 ms | 19 ms | 46 ms | 9 ms | 19 ms | 50 ms |
|  |  |  |  |  |  |  |
| **Employee Bitmap Index** | | | | | | |
| Time taken to generate sublists | 526 ms | 915 ms | 1617 ms | 310 ms | 986 ms | 1670 ms |
| Time taken for Merge Data | 8677 ms | 11 | 206229 ms | 7911 ms | 46749 ms | 170764 ms |
| Read Count | 119469 | 346709 | 785848 | 119755 | 348511 | 793391 |
| Write Count | 99257 | 296324 | 685118 | 99597 | 298131 | 692592 |
| Total Time for Bitmap index | 9203 ms | 48094 ms | 207846 ms | 8221 ms | 47735 ms | 172434 ms |
| Time taken for compression | 8581 ms | 52762 ms | 215179 ms | 8617 ms | 53650 ms | 217501 ms |
|  |  |  |  |  |  |  |

**Time taken to merge T1 and T2:**

* For 20,000: 7157 ms (~approx. 7.157 sec)
* For 50,000: 30369 ms (~approx. 30.369 sec)
* For 1,00,000: 138799 ms (~approx. 138.799 sec)

**A close up of text on a white background

Description automatically generated**

Figure 1: 10MB – 20,000 records Bitmap Index

**A close up of text on a white background

Description automatically generated**

Figure 2: 10MB – 50,000 records Bitmap Index

**A close up of text on a white background

Description automatically generated**

Figure 3: 10MB – 1,00,000 records Bitmap Index

1. **Results TPMMS**

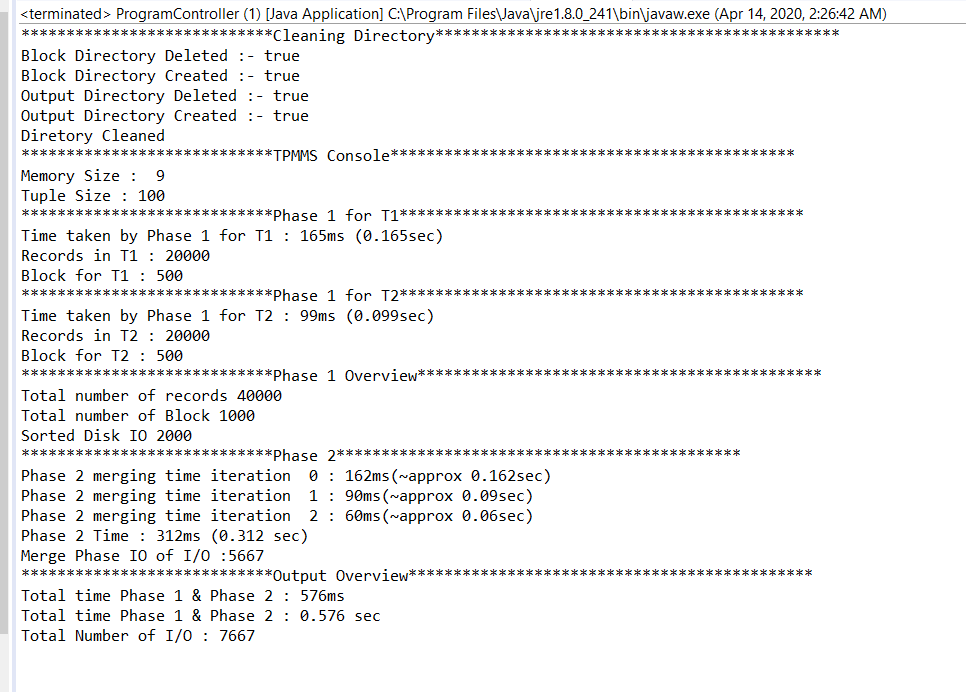


Figure 4: 10MB – 20,000 records TPMMS

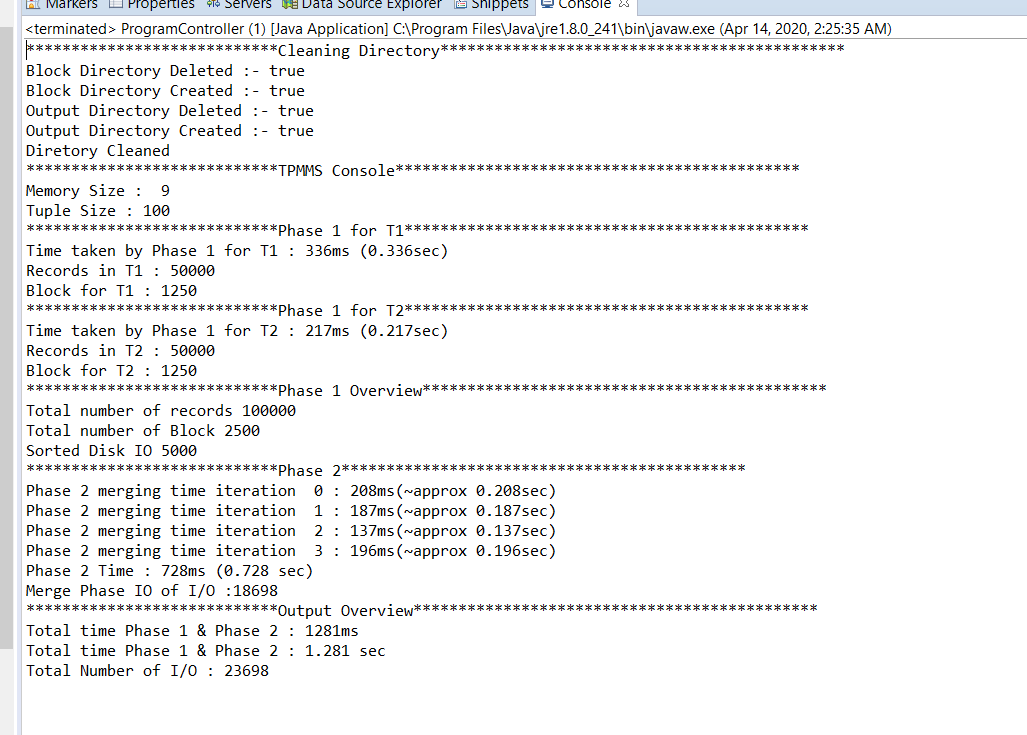


Figure 5: 10MB – 50,000 Records TPMMS

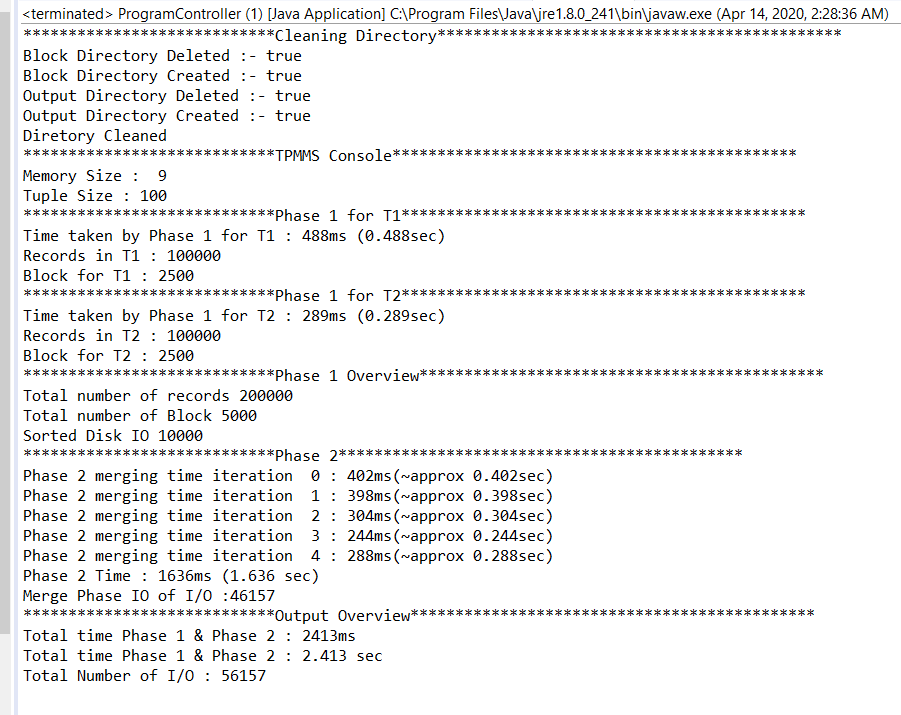


Figure 6: 10MB – 1,00,000 Records TPMMS

1. **Comparison to TPMMS**

Compare to TPMMS, Bitmap index takes more time to generate output file containing recently updated records and no duplicates. One of the major problems occurs while using Bitmap index approach, size of the file is continuously increasing with increase in number of Tuple. For example, when file with 5,00,000 Tuples uses more than 137 GB space for only Bitmap index and output file as well as it might takes days to generate Bitmap Index only. While in TPMMS, it just takes few mb to generate output in few minutes. While Comparing Disk I\O, both Bitmap Index and TPMMS have approximately same number of disk I/O. Since, both approaches have advantages and disadvantages. It was also observed that if file contain

1. **Coding Standards**

The most general coding conventions were followed while the codes were developed as follows,

* The class name begins with an uppercase word.
* E.g.: ProgramController.java
* Constants are called with characters in the upper case
* The variable name is descriptive and is rendered in lower case including a capital letter to separate words.
* The procedure name begins with a lowercase character and uses the uppercase characters to separate words.

1. **Class Description:**

**Constants.java:**  This class stores the constant values required for program execution like file IO paths, block size etc.

**ProgramController.java:**  This class has the methods for reading tuples into main memory, sorting them and creating the sublists. It also handles reading the indices and producing final merged output.

**BuildIndex.java:** This class creates bit vectors and bitmap index subsequently from the input files.

**CompressedBitmap.java:** This creates the compressed bitmaps.

**QuickSort.java:**  This class performs the quicksort algorithm on the bitmap index for blocks read into memory.

**MergeData.java:**  This class merges the bitmap for blocks into a single bitmap index.

1. **Group Member Contribution:**

Member participation was uniform across all stages of the project development. We had meetings once a week to discuss on design changes and individual progress. This ensured that everyone is on the same page. We also adopted pair programming strategy which let us help each other with our areas of expertise, thereby developing efficient code. The documentation part was split into sections and assigned to each teammate as a part of even work distribution.

**References**

1. https://www.geeksforgeeks.org/java-program-for-quicksort/
2. https://oracle-base.com/articles/9i/bitmap-join-indexes
3. https://logicalread.com/2013/06/03/oracle-11g-bitmap-join-indexes-mc02/
4. https://www.youtube.com/watch?v=sMbQW7XNUZs
5. http://www.dba-oracle.com/art\_builder\_bitmap\_join\_idx.htm
6. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7030998/
7. https://www.geeksforgeeks.org/bitmap-indexing-in-dbms/
8. https://www.geeksforgeeks.org/indexing-in-databases-set-1/
9. https://en.wikipedia.org/wiki/Bitmap\_index
10. https://roaringbitmap.org/about/
11. https://sdm.lbl.gov/~kewu/ps/LBNL-49627.pdf