EXTERNAL SORTING ALGORITHMS AND IMPLEMENTATIONS

Project Supervisor: Assist. Prof. Mustafa Utku KALAY

Project Group
05011004 Ayhan KARGIN
05011027 Ahmet MERAL

İstanbul, 2012
CONTENT
SYMBOL LIST ........................................................................ iv
ABBREVIATION LIST ................................................................. v
FIGURE LIST ........................................................................ vi
TABLE LIST ........................................................................ vii
PREFACE ........................................................................ viii
ABSTRACT ........................................................................ ix
ÖZET ................................................................................. x
1. INTRODUCTION ................................................................ 1
2. FEASIBILITY ...................................................................... 2
  2.1. Software Feasibility ......................................................... 2
  2.2. Hardware Feasibility ....................................................... 2
  2.3. Legal Feasibility ............................................................. 2
  2.4. Time Feasibility ............................................................. 3
  2.5. Economic Feasibility ...................................................... 5
3. EXTERNAL SORTING ALGORITHMS .................................. 6
  3.1. Merge Sort .................................................................. 6
    3.1.1. Run Generation Phase ........................................... 7
    3.1.2. Merge Phase .......................................................... 7
  3.2. K-Way Merge Sort ........................................................ 7
  3.3. Multi-Step K-Way Merge Sort ....................................... 9
4. THE SIMPLEDB SYSTEM .................................................. 13
  4.1. Overview of SimpleDB .................................................. 13
    4.1.1. The Client Side Code .............................................. 13
    4.1.2. Basic Server .......................................................... 14
    4.1.3. Efficiency Extensions ............................................. 17
  4.2. Detailed Look on SimpleDB for External Sorting ............ 18
    4.2.1. Query .................................................................. 18
    4.2.2. Parser .................................................................. 25
5. EXTERNAL SORTING ON SIMPLEDB ................................ 26
  5.1. Materialize Plan ............................................................ 26
    5.1.1. Implementing Materialize Plan ................................. 27
5.2. External Sorting Approach of SimpleDB’s Author ........................................ 29
5.3. Preparatory Work .................................................................................... 29
5.4. N-Way Merge Sort .................................................................................. 30
5.5. K-Way Merge Sort .................................................................................. 30
5.6. Multi-Step K-Way Merge Sort ................................................................. 31
6. REPLACEMENT SELECTION .................................................................. 32
   6.1. Heap ..................................................................................................... 32
      6.1.1. Operations ..................................................................................... 34
      6.1.2. Implementation ............................................................................. 37
   6.2. Heap Sort ............................................................................................ 38
   6.3. Replacement Selection ......................................................................... 39
   6.4. Run Length .......................................................................................... 40
   6.5. Determining the Stage Area Size .......................................................... 42
7. CONCLUSION ............................................................................................... 44
REFERENCES .................................................................................................. 48
APPENDIX ...................................................................................................... 49
CURRICULUM VITAE ...................................................................................... 49

Hata! Yer işareti tanımlanmamış.
### SYMBOL LIST

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Σ</td>
<td>Total</td>
</tr>
<tr>
<td>√</td>
<td>Square Root</td>
</tr>
<tr>
<td>O</td>
<td>Complexity</td>
</tr>
</tbody>
</table>
### ABBREVIATION LIST

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRP</td>
<td>Current Record Pointer</td>
</tr>
<tr>
<td>ESMS</td>
<td>Edward Sciore’s Merge Sort</td>
</tr>
<tr>
<td>ETL</td>
<td>Extraction, Transform and Load</td>
</tr>
<tr>
<td>GNU</td>
<td>Gnu Not Unix</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JAR</td>
<td>Java Archive</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>KWMS</td>
<td>K-Way Merge Sort</td>
</tr>
<tr>
<td>LSS</td>
<td>Load Sort Store</td>
</tr>
<tr>
<td>MSKWMS</td>
<td>Multi-Step K-Way Merge Sort</td>
</tr>
<tr>
<td>MSRSS</td>
<td>Multi-Step Replacement Selection Sort</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RPB</td>
<td>Records Per Block</td>
</tr>
<tr>
<td>RS</td>
<td>Replacement Selection</td>
</tr>
<tr>
<td>RSS</td>
<td>Replacement Selection Sort</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
**FIGURE LIST**

- Figure 2.1 Time Plan with Sub Unit Explanations ............................................ 3
- Figure 2.2 Time Plan with Visual Expression ..................................................... 4
- Figure 3.1 8,000 MB File Sorting With 800 Way Merge ...................................... 8
- Figure 3.2 Two-Step K-Way Merge of 800 Runs .................................................. 10
- Figure 3.3 Two Step K-Way Merge (25*32 Way and 25 Way) ............................... 11
- Figure 4.1 Printing the Salary of Everyone in the Sales Department .................... 14
- Figure 4.2 Student and Dept Tables .................................................................... 22
- Figure 4.3 Query Tree with Relational Algebra Operators .................................... 22
- Figure 4.4 Scan Interface ..................................................................................... 23
- Figure 4.5 UpdateScan Interface ......................................................................... 24
- Figure 4.6 Plan Interface ..................................................................................... 24
- Figure 5.1 MaterializePlan Class ......................................................................... 28
- Figure 6.1 An Example Graph .............................................................................. 32
- Figure 6.2 A Completely Filled Binary Max Heap .................................................. 34
- Figure 6.3 The Process of Adding the Record 91 to the Heap ............................... 35
- Figure 6.4 The Process of Deleting the Top Record of the Heap ......................... 37
- Figure 6.5 A Binary Complete Tree with 3 Levels and Its Associated Array ......... 38
- Figure 6.6 Circular Road with a Snowplow ......................................................... 41
- Figure 6.7 Stabilized Situation ............................................................................. 41
- Figure 7.1 Algorithm Choice Dependency ............................................................ 47
TABLE LIST
Table 6.1 Generating and Merging Runs with Different Stage Area Sizes ............... 42
Table 6.2 Only Generating Runs Phase Results ................................................... 42
Table 7.1 Comparison of External Sorting Algorithms Part One .......................... 44
Table 7.2 Comparison of External Sorting Algorithms Part Two ......................... 45
PREFACE

We would like to thank our supervisor Assist. Prof. Mustafa Utku KALAY for his support and guidance on our project. Also we would thank to Assoc. Prof. Edward Sciore who has written SimpleDB educational open source database management system. By the way we could implemented our project on SimpleDB. And we thank to open knowledge and open source world that helped us to do our project better.
ABSTRACT

On database management systems or on processing data joining is a big need. Before computer hardware features were low and data which will be processed were small. Nowadays data amount increased a lot, meanwhile hardware features are high. These situation express, before and now (possibly in the future) when all the data which will be sorted can not place in main memory, sorting have to be done by using secondary storage device (hard disc).

To solve this need k-way merge, multi-step k-way merge and replacement selection algorithms are thought. On this project our purpose is understand these algorithms, implement these algorithms on SimpleDB (which is a educational purposed open source database management system) and analyze results (characteristics) of algorithms on various test case scenarios. At the end we will create a table that compare these algorithms’ result set.

Keywords: heapsort, external sorting, k-way merge, multi-step k-way merge, replacement selection
ÖZET

Özellikle veritabanı yönetim sistemlerinde ya da verileri işlerken yapısal (normalize) verilerin eşleştirilmesi (join) gerekir. Eskiden bilgisayarların donanımsal özellikleri düşüktü ve işlenecek veri miktarı az idi. Günümüzde de veri miktarı çok fazla ve bilgisayar donanımlarının özellikleri çok yüksek. Bu durumlar göz önüne alındığında eskiden de günümüzde de sıralanacak tüm verinin bellekte tutulamadan ve ikincil saklama ünitesi (sabit disk) kullanılarak sıralama yapılmaya ihtiyaç duyulmuştur.

Bu ihtiyacı çözüm olarak k-yollu sıralı birleştirme (k-way merge), çok-adımlı k-yollu sıralı birleştirme (multi-step k-way merge) ve değişmeli seçim (replacement selection) algoritmaları geliştirilmiştir. Bu projede amacımız bu algoritmaları anlayıp eğitimsel amaçlı olarak yazılmış açık kaynak olan SimpleDB veritabanı yönetim sisteminde gerçekleştirmek ve çeşitli senaryolarda bu algoritmaların karakteristiklerini (davranışlarını) karşılaştıran tablolar oluşturmaktır.

Anahtar Kelimeler: yığın sıralama (heapsort), harici sıralama (external sorting), k-yollu sıralı birleştirme, çok-adımlı k-yollu sıralı birleştirme, değişmeli seçim
1. INTRODUCTION

External Sorting is needed when big data have to be processed (join, sort, group by, etc.). Mostly these types of processes are used in Database Management Systems, ETL (Extraction Transform Load) on Data Warehouse, Data Mining.

We need an environment that we can implement algorithms by modifying some area of complete system code. Commercial open source softwares have the cachet of being "real" code, but is large, complex, and full featured. Not only will it have an easy learning advantage, but it will be difficult to modify because all of the simple improvements will have already been made. PosgreSQL is like mentioned above. An alternative Database Management System is Minibase [1]. Minibase attempts to have the structure and functionality of a commercial database system, and yet be simple to understand and extend. By trying to balance both concerns, it winds up not being very good at either. It has an easy learning benefit, but without the advantages of an open source system. It has no multi-user or transaction capability too. Another alternative database system mentioned in the literature for educational purposes is MinSql [2]. This system is designed to have heavyweight architecture but lightweight code. That is, the components of the system and their functionality are essentially the same as in a commercial system, but the actual code implements only a small fraction of what it could. For example, instead of implementing all of SQL, the system implements just enough to allow for nontrivial queries. The advantage to a system such as MinSql is that it simplifies the learning curve – developers are not overwhelmed by irrelevant features and their consequent programming details. Unfortunately, MinSql was never made public, and so it is not possible to build a course around it.

Another alternative is SimpleDB Database Management System [3]. SimpleDB’s primary goal is to be readable, usable and easily modifiable. As with MinSql, it has the basic architecture of a commercial database system, but stripped of all unnecessary functionality and using only the simplest algorithms. The system is written in Java and takes full advantage of Java libraries. For example, it uses Java RMI to handle the client-server issues and the Java VM to handle thread scheduling. We have chosen SimpleDB to implement and compare External Sorting algorithms on.
2. FEASIBILITY

In this section project’s feasibility will be handled by various points of views.

2.1. Software Feasibility

Project will be implemented on SimpleDB educational database management system. SimpleDB is written in Java Programming Language. Java is Operating System (OS) independent so we need an OS which Java Virtual Machine (Java VM) installed. SimpleDB code uses Java Standard Edition (Java SE) libraries which is free to use. On the other hand we need a Java Compiler to implement our modified SimpleDB code. In addition we need an Integrated Development Environment (IDE) to add External Sorting algorithms to SimpleDB code easier. Eclipse or NetBeans can be good choice. Eventually we must install Java VM and Java SE and Eclipse (or NetBeans) on our OS.

2.2. Hardware Feasibility

Minimum hardware requirements to implement our project place below.

20 GB Hard Disc (External Storage), 50$

512 MB Main Memory, 40$

1 Ghz Processor, 60$

2.3. Legal Feasibility

The code which we will write to implement External Sorting algorithms will be GNU/GPL (free to use and distribute) licensed. By the way we will contribute to Open Source idea. There is nothing illegal to install and use softwares which mentioned about in section 2.1. Moreover SimpleDB is open source (appropriately for its purpose) and permitted to use, modify, redistribute.
2.4. Time Feasibility

Timesheet (Gantt Diagram) which we want to follow, place in Figure 2.1 and 2.2.

<table>
<thead>
<tr>
<th>No.</th>
<th>Görev Adı</th>
<th>Süre</th>
<th>Bağlangıç</th>
<th>Bitiş</th>
<th>Öncüller</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Preliminary Investigation and Feasibility Analysis</td>
<td>8 gün</td>
<td>Pzt 26.09.11</td>
<td>Per 06.10.11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Requirements Analysis</td>
<td>1 gün</td>
<td>Pzt 26.09.11</td>
<td>Pzt 26.09.11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Draft System Analysis</td>
<td>1 gün</td>
<td>Sal 27.09.11</td>
<td>Sal 27.09.11, 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Investigation of algorithms</td>
<td>4 gün</td>
<td>Çar 28.09.11</td>
<td>Sal 04.10.11, 3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Feasibility Study</td>
<td>2 gün</td>
<td>Çar 05.10.11</td>
<td>Per 06.10.11, 4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>System Analysis</td>
<td>7 gün</td>
<td>Pzt 10.10.11</td>
<td>Çar 19.10.11, 1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Requirement (time, workforce, etc.) Planning</td>
<td>2 gün</td>
<td>Pzt 10.10.11</td>
<td>Sal 11.10.11</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Detailed System Analysis</td>
<td>3 gün</td>
<td>Çar 12.10.11</td>
<td>Pzt 17.10.11, 7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>System Analysis Report Preparation</td>
<td>2 gün</td>
<td>Sal 18.10.11</td>
<td>Çar 19.10.11, 8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>System Design</td>
<td>19 gün</td>
<td>Per 20.10.11</td>
<td>Sal 21.11.11, 6</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Algorithm and Flow Design</td>
<td>8 gün</td>
<td>Per 20.10.11</td>
<td>Çar 02.11.11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Database (SimpleDB) Investigation</td>
<td>8 gün</td>
<td>Per 03.11.11</td>
<td>Çar 16.11.11, 11</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Preparation of System Design Report</td>
<td>3 gün</td>
<td>Per 17.11.11</td>
<td>Sal 22.11.11, 12</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Implementation</td>
<td>23 gün</td>
<td>Çar 23.11.11</td>
<td>Pzt 02.01.12, 10</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Creating the Test Data (Sample Data)</td>
<td>4 gün</td>
<td>Çar 23.11.11</td>
<td>Sal 29.11.11</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Program Coding (Java in SimpleDB)</td>
<td>13 gün</td>
<td>Çar 30.11.11</td>
<td>Çar 21.12.11, 15</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Program Code Test</td>
<td>13 gün</td>
<td>Çar 30.11.11</td>
<td>Çar 21.12.11, 15</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Documentation Preparation</td>
<td>6 gün</td>
<td>Per 22.12.11</td>
<td>Pzt 02.01.12, 16</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Test</td>
<td>3 gün</td>
<td>Sal 03.01.12</td>
<td>Per 05.01.12, 14</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Engineering Test</td>
<td>1 gün</td>
<td>Sal 03.01.12</td>
<td>Sal 03.01.12</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>User Test</td>
<td>2 gün</td>
<td>Çar 04.01.12</td>
<td>Per 05.01.12, 20</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2.1 Time Plan with Sub Unit Explanations*
2.5. Economic Feasibility

Softwares which mentioned about in section 2.1 are free. Hardwares’ price which mentioned about section 2.2 are total 150$. Cost for the work force in this project is about 144 week * 4 day per week * 2 people * 30$ per day = 3360$. Consequently our project worth 3510$ total.
3. EXTERNAL SORTING ALGORITHMS

External Sorting is a term to refer to a class of sorting algorithms that can handle large amounts of data. The elements that are ordered by a sorting algorithm are referred to as records. When there are more records than those that fit in the main memory of the computing device used to sort the records, external sorting is required. Since more memory is needed, these algorithms rely on the use of slow external memory. Accessing and modifying external memory is much slower than doing the same with internal memory, so the execution time of external sorting algorithms is heavily affected by the number of accesses to external memory. So, when analyzing the performance of an external sorting algorithm, one must consider the amount of input/output operations in addition to the algorithmic complexity of the algorithm.

The problem of external sorting starts to exist typically in databases, because they are specialized applications for handling huge amounts of data. When databases perform operations with data, it is almost every time necessary to sort part of the data for more complex operation. For example, when joining data from two different tables or grouping rows having common values, it is necessary to perform sort operations on data, although the sort is not clearly requested. In most cases, the amount of memory available to the processes performing operations on the database is much less than the amount of data stored in it. In these cases, an external sorting algorithm is needed.

One of the most commonly used generic approaches to external sorting is the merge sort. The merge sort consists of sorting records as they are read from the input, generating several independent sorted lists of records that are merged into the final sorted list in a second phase.

3.1. Merge Sort

One of the most commonly approaches to external sorting is external merge sort, which consists of two phases, the run generation phase and the merge phase. The first phase generates several sorted lists of records, called runs, and the second phase merges the runs into the final sorted list of records. Replacement selection is a sorting algorithm
that performs the run generation phase of external merge sort. As such, it can be combined with any algorithm for the merge phase.

3.1.1. Run Generation Phase

In the run generation phase, data is read from the input to generate subsets of ordered records. These subsets are called runs. Runs are generated using main (internal) memory, and written to external memory (disk). After all input records are distributed in runs, the run generation phase ends and the merge phase starts.

There are several methods used to generate the runs, most of them being based on internal sorting algorithms. For example, the main memory can be filled with records from the input and then sorted using any internal sorting algorithm (merge sort, quicksort, etc.) Using this method, called Load-Sort-Store, the run length is always equal to the size of the main memory, except for maybe the last run. Another more advanced algorithm is replacement selection. Using replacement selection, the run length is nearly equal to twice the size of the main memory (internal) when the input data is randomly distributed.

Generating longer runs means having fewer runs to merge in the next phase, which in turn allows for shorter execution times. Replacement selection is more complicated than a Load-Sort-Store solution, but overall less time is needed to sort records.

3.1.2. Merge Phase

The purpose of the merge phase is to merge the runs generated in the previous phase into a unique run containing all input records. We are going to talk about two different ways of performing this phase are k-way merge and balanced (multi-step) k-way merge.

3.2. K-Way Merge Sort

A k-way merge combines k runs into one sorted run. This process reduces the number of runs to merge by k-1, and is repeated until only one run is left. The simplest example
is the 2-way merge, where two sorted runs are merged into one. In a two way merge, we only need to know the first two records of each run. The smaller of these two records is the smallest record overall. This record is put in the output run and removed from the related run. This process is repeated until the two initial runs are empty. The k-way merge behaves identically, but at each step the smallest of k records is selected and removed from its run.

As an example, we want to sort a file that has 80,000,000 records and each record 100 byte long (~8000 MB). Also we have a 10 MB main memory and 200,000 byte length of output buffers for writing to final output. We can have only 10 MB / 100 Byte = 100,000 records in main memory at the same time. So, 80,000,000 records / 100,000 = 800 and it means k=800. We are going to do 800-way merge.

(Disk specification: seek time (ts)=8, rotation time(tr)=3, Transfer Rate (tt)=14.500bytes/msec)

![Diagram of 8,000 MB File Sorting With 800 Way Merge](image.png)

Figure 3.1 8,000 MB File Sorting With 800 Way Merge
**Step 1**: Fill the main memory with records from the relation to be sorted and do this 800 times. When we access the disk, it’s for either reading or writing, it takes up time as (the number of access)\((ts+tr)\) +(amount of the data)\(\cdot tt\).

In the Step 1 from the Figure 3.1, we have accessed 800 times and we transferred the entire data in the main memory. So;

Expected time: \[ 800 \cdot (8 + 3) + 8,000MB / 14,500B / 10^3 = 600 \text{ sec.} \]

**Step 2**: Main memory sorts the taken records. Write records in sorted order into new blocks, forming one sorted sublist (or run). Now we have 800 sorted runs.

Again, we have accessed 800 times and we transferred the entire data in the external disk. So;

Expected time: \[ 800 \cdot (8 + 3) + 8,000MB / 14,500B / 10^3 = 600 \text{ sec.} \]

**Step 3**: We take 100,000/800=125 records from each run into memory and do this 100,000/125=800 times for each run. It means we take 1/800 of the every run on each turn. So, we will access the disk 800*800 times. We transfer the entire data in the main memory. Using a heap, find the smallest key among the first remaining record in each access, and then move the corresponding record to the first available position of the output buffer. So;

Expected time: \[ 800 \cdot 800 \cdot (8 + 3) + 8,000MB / 14,500B / 10^3 = 7640 \text{ sec.} \]

**Step 4**: If the output buffer is full, write it to disk and empty the output buffer. Total amount of data 8,000MB and output buffer 200KB. So, we empty the output buffer 8,000MB/200KB=40,000 times. We write the entire data in the disk.

Expected time= 40,000 \(\cdot (8 + 3)\) + 8,000MB / 14,500B / 10^3 = 1040 sec.

**Total Time** = 600 + 600 + 7640 + 1040 = 9880 sec.

3.3. **Multi-Step K-Way Merge Sort**

As can be seen in the k-way merge sort, merging the runs take almost the entire time. This is related to the number of disk accesses. Therefore, we use multi-step k-way merge to decrease the number of disk accesses. Instead of merging all runs at once, we
break the original set of runs into small groups and merge the runs in these groups separately. When all of the smaller merges are completed, a second pass merges the new set of merged runs.

We will use the same example above to perform multi-step k-way merge sort. We want to sort a file that has 80,000,000 records and each record 100 byte long (~8000 MB). Also we have a 10 MB main memory and 200,000 byte length of output buffers for writing to final output. We can have only 10 MB / 100 Byte = 100,000 records in main memory at the same time. So, 80,000,000 records / 100,000 = 800 runs.

(Disk specification: seek time (ts)=8, rotation time(tr)=3, Transfer Rate (tt)= 14.500bytes/msec)

Multi-step k-way merge sort is different from k-way merge sort only in merge phase. So, these two algorithms have the same run generation phase for the example above. At the end of the run generation phase, we have 800 runs. We split the 800 runs into 25 pieces and each piece has 32 runs. Firstly we merge every piece separately and then merge all the 25 pieces.
Step 1: We take $100,000/32 = 3125$ records from each run into memory and do this 32 times for each pieces. We have 25 pieces. So, we will access the disk $(32 \times 32) \times 25$ times. We transfer the entire data in the main memory. Using a heap, find the smallest key among the first remaining record in each access, and then move the corresponding record to the first available position of the output buffer.

Expected time $= (32 \times 32) \times 25 \times (8 + 3) + 8,000 \text{MB} / 14,500 \text{B} / 10^3 = 882 \text{ sec.}$

Step 2: If the output buffer is full, write it to disk and empty the output buffer. Total amount of data 8,000MB and output buffer 200KB. So, we empty the output buffer 8,000MB/200KB=40,000 times. We write the entire data in the disk.

Expected time $= 40,000 \times (8 + 3) + 8,000 \text{MB} / 14,500 \text{B} / 10^3 = 1040 \text{ sec.}$

Step 3: After the step 2, we have 25 runs in the disk and length of the each run 320MB ($32 \times 100,000 \text{ records} \times 100B$). We have to do 25-way merge. So, we take $100,000/25 = 4,000$ records from each run into memory and do this $3,200,000/4,000 = 800$
times for each run. We will access the disk 800*25 times. Using a heap, find the smallest key among the first remaining record in each access, and then move the corresponding record to the first available position of the output buffer.

Expected time = 800 * 25 * (8 + 3) + 8,000MB / 14,500B / 10³ = 820 sec.

**Step 4:** If the output buffer is full, write it to disk and empty the output buffer. Total amount of data 8,000MB and output buffer 200KB. So, we empty the output buffer 8,000MB/200KB=40,000 times. We write the entire data in the disk again.

Expected time = 40,000 * (8 + 3) + 8,000MB / 14,500B / 10³ = 1040 sec.

Total Time = 600 + 600 + 882 + 1040 + 820 + 1040 = 4982 sec.

When we use multi-step k-way merge, we can use larger buffers and avoid a large number of disk seeks. But, we have to read same record more than once. We create longer runs with adding one more merge step. Creating longer runs means, we decreased the number of disk accesses. For instance, when we use 800-way merge we access the disk 800*800=640,000 times for merging the data. But; when we use 32*25-way and 25-way, we access 25,600+20,000=45,600 times.
4. THE SIMPLEDB SYSTEM

In this chapter we are going to overview the SimpleDB system and examine relevant layer with external sorting algorithms that we will try to perform.

4.1. Overview of SimpleDB

SimpleDB’s primary goal is to be readable, usable, and easily modifiable. It has the basic architecture of a commercial database system, but stripped of all unnecessary functionality and using only the simplest algorithms. The system is written in Java, and takes full advantage of Java libraries. For example, it uses Java RMI to handle the client-server issues, and the Java VM to handle thread scheduling.

The SimpleDB code comes in three parts:

- The client-side code that contains the JDBC interfaces and implements the JDBC driver.
- The basic server, which provides complete (albeit barebones) functionality but ignores efficiency issues.
- Extensions to the basic server that support efficient query processing.

The following subsections address each part.

4.1.1. The Client Side Code

A SimpleDB client is a Java program that communicates with the server via JDBC. For example, the code fragment of Figure 4.1 prints the salary of each employee in the sales department.
The JDBC package java.sql defines the interfaces Driver, Connection, Statement and ResultSet. The database system is responsible for providing classes that implement these interfaces; in SimpleDB, these classes are named SimpleDriver, SimpleConnection, etc. The client only needs to know about SimpleDriver, but all classes need to be available to it. In most commercial systems, these classes are packaged in a jar file that is added to the client’s classpath. SimpleDB does not come with a client-side jar file, but it is an easy (and useful) exercise for the students to create one.

The standard JDBC interfaces have a large number of methods, most of which are peripheral to the understanding of database internals. Therefore, SimpleDB comes with its own version of these interfaces, which contain a small subset of the methods. The advantages are that the SimpleDB code can be smaller and more focused, and that the omitted methods can be implemented as class exercises, if desired.

4.1.2. Basic Server

The basic server comprises most of the SimpleDB code. It consists of ten layered components, where each component uses the services of the components below it and provides services to the components above it. These components are displayed in Figure 4.2. The remainder of this section discusses these components briefly, from the bottom up.

- **Remote**: Perform JDBC requests received from clients.
- **Planner**: Create an execution strategy for an SQL statement, and translate it to a relational algebra plan.
- **Parse**: Extract the tables, fields, and predicate mentioned in an SQL statement.
- **Query**: Implement queries expressed in relational algebra.
- **Metadata**: Maintain metadata about the tables in the database, so that its records and fields are accessible.
- **Record**: Provide methods for storing data records in pages.
- **Transaction**: Support concurrency by restricting page access. Enable recovery by logging changes to pages.
- **Buffer**: Maintain a cache of pages in memory to hold recently-accessed user data.
- **Log**: Append log records to the log file, and scan the records in the log file.
- **File**: Read and write between file blocks and memory pages.

The file manager supports access to the various data files used by SimpleDB: a file for each table, the index files, some catalog files, and a log file. The file manager API contains methods for random-access reading and writing of blocks. Higher-level components see the database as a collection of blocks on disk, where a block contains a fixed number of bytes.

The log manager is responsible for maintaining the log file. Its API contains methods to write a log record to the file, and to iterate backwards through the records in the log file.

The buffer manager is responsible for the in-memory storage of pages, where a page holds the contents of a block. Its API contains methods to pin a buffer to a block, to flush a buffer to disk, and to get/set a value at an arbitrary location inside of a block. Higher-level components see the database as a collection of in-memory pages of values.

The transaction manager is a wrapper around the buffer manager. It has essentially the same API as the buffer manager, with some additional methods to commit and rollback transactions. The job of the transaction manager is to intercept calls to the buffer manager in order to handle concurrency control and recovery. It treats blocks as the unit
of lock granularity, obtaining an slock (or xlock) on the appropriate block whenever a method to get (or set) a value is called. The transaction manager also supports recovery by using write-ahead logging of values; when a method to set a value is called, the transaction manager writes the old value into the log before telling the buffer manager to write the new value to the page. Higher-level components still see the database as a collection of pages of values, but with methods that ensure safety and serializability.

The record manager is responsible for formatting a block into fixed-length, unspanned records. Its API contains methods to iterate through all of the records in a file. The record manager hides the block structure of the database. Higher-level components see the database as a collection of files, each containing a sequence of records.

The metadata manager stores schema information in catalog files. Its API contains methods to create a new table given a schema, and to retrieve the schema of an existing table. The metadata manager hides the physical characteristics of the database. Higher-level components see the database as a collection of tables and indexes, each containing a sequence of records.

The query processor implements query trees that can be composed from the relational algebra operators select, project, and product. Its API contains methods to create a query tree and to iterate through it.

The parser recognizes a stripped-down subset of SQL, using recursive descent. The language corresponds to select-project-join queries having very simple predicates. There are no Boolean operators except “and”, no comparisons except “=” , no arithmetic or built-in functions, no grouping, no renaming, etc.

The planner builds a query plan from the parsed representation of the query. The plan is the simplest possible: It takes the product of the mentioned tables (in the order mentioned), followed by a select operation using the where-clause predicate, and followed by a projection on the output fields.
Finally, the remote interface implements a small subset of the JDBC API. The key method is `Statement.executeQuery`, which calls the parser and planner to construct the query tree and passes it to the `ResultSet` object for traversal. All of the network communication is taken care of by Java RMI.

### 4.1.3. Efficiency Extensions

The basic query processor only knows about three relational operators. It doesn’t know how to use indexing, nor can it handle sorting or grouping. Moreover, the iterator implementations are as simple as possible – most notably, the implementation of `product` uses nested loops.

These algorithms are, of course, remarkably inefficient. But they also have a simplicity that allows students to focus on the flow of control in the execution of a query tree. Students tend to have difficulty grasping how a query tree of iterators’ works and so clarity is more important than efficiency at this point.

The basic planner is equally simple. It does not try to perform joins, or push selections, or optimize join order. The advantage is again clarity over efficiency. A trivial planner allows students to focus exclusively on how the translation from SQL to relational algebra works.

But efficiency, of course, is critical for a database system. Once students understand the basic server, it can be extended with four components to improve efficiency:

- Support for indexing.
- Sorting and operators that rely on sorting (such as aggregation, duplicate removal, and mergejoin).
- Sophisticated buffer allocation.
- Query optimization.
The indexing component implements both B-tree and static hash indexes, and provides implementations of the `indexselect` and `indexjoin` operators.

The sorting component provides a sort operator, implemented using a simple mergesort algorithm. It also uses the sort operator to implement `groupby` and `mergejoin` operators.

The buffer allocation component modifies the `sort` and `product` operators to take maximum advantage of available buffers.

The query optimization component implements an intelligent planner. The planner uses a greedy optimization algorithm, and can be configured to use the various efficient operators (e.g. `mergejoin` or `indexjoin` instead of `product`) when possible.

4.2. Detailed Look on SimpleDB for External Sorting

We have to do a few changes on layered structure of SimpleDB for implementing external sorting. So, we have a deeply look on Query, Parser and Planner layers. In appendix we can see class diagram of all SimpleDB layers.

4.2.1. Query

In this part, we focus on queries, which are commands that extract information from a database. We introduce the two query languages relational algebra and SQL. Both languages have similar power, but very different concerns. A query in relational algebra is task centered: it is composed of several operators, each of which performs a small, focused task. On the other hand, a query in SQL is result oriented: it specifies the desired information, but is vague on how to obtain it. Both languages are important, because they have different uses in a database system. In particular, users write queries in SQL, but the database system needs to translate the SQL query into relational algebra in order to execute it.
4.2.1.1. Relational Algebra

Relational algebra consists of operators. Each operator performs one specialized task, taking one or more tables as input and producing one output table. Complex queries can be constructed by composing these operators in various ways.

We shall consider twelve operators that are useful for understanding and translating SQL. The first six take a single table as input, and the remaining six take two tables as input. The following subsections discuss each operator in detail. For reference, the operators are summarized below.

The single-table operators are:

- select, whose output table has the same columns as its input table, but with some rows removed.
- project, whose output table has the same rows as its input table, but with some columns removed.
- sort, whose output table is the same as the input table, except that the rows are in a different order.
- rename, whose output table is the same as the input table, except that one column has a different name.
- extend, whose output table is the same as the input table, except that there is an additional column containing a computed value.
- groupby, whose output table has one row for every group of input records.

The two-table operators are:

- product, whose output table consists of all possible combinations of records from its two input tables.
- join, which is used to connect two tables together meaningfully. It is equivalent to a selection of a product.
- semijoin, whose output table consists of the records from the first input table that match some record in the second input table.
- antijoin, whose output table consists of the records from the first input table that do not match records in the second input table.
- union, whose output table consists of the records from each of its two input tables.
- outer join, whose output table contains all the records of the join, together with the non-matching records padded with nulls.

SimpleDB supports only select, project and product operators from the mentioned relational algebra operators. We will add “sort” operator to these supported operators later.

**Select**
The select operator takes two arguments: an input table and a predicate. The output table consists of those input records that satisfy the predicate. A select query always returns a table having the same schema as the input table, but with a subset of the records.

For example, query Q1 returns a table listing those students who graduated in 2004:

\[ Q1 = \text{select(STUDENT, GradYear=2004)} \]

Students who graduated in 2004

**Project**
The project operator takes two arguments: an input table, and a set of field names. The output table has the same records as the input table, but its schema contains only those specified fields.

For example, query Q2 returns the name and graduation year of all students:

\[ Q2 = \text{project(STUDENT, \{SName, GradYear\})} \]

The name and graduation year of all students
**Product**
This operator takes two input tables as arguments. Its output table consists of all combinations of records from each input table, and its schema consists of the union of the fields in the input schemas. The input tables must have disjoint field names, so that the output table will not have two fields with the same name.

For example query Q3 returns the product of the STUDENT and DEPT tables:

\[ Q3 = \text{product}(\text{STUDENT}, \text{DEPT}) \]

All combinations of records from STUDENT and DEPT

**Sort**
The sort operator takes two arguments: an input table and a list of field names. The output table has the same records as the input table, but sorted according to the fields. For example, query Q4 sorts the STUDENT table by GradYear; students having the same graduation year will be sorted by name. If two students have the same name and graduation year, then their records may appear in any order.

\[ Q4 = \text{sort}(\text{STUDENT}, \{\text{GradYear}, \text{SName}\}) \]

Student records sorted by graduation year and name

We will give an example to explain how to convert SQL expressions to relational algebra operators using Figure 4.3 tables.
An example SQL expression:

Select SId, SName, DName
from STUDENT, DEPT
where MajorId=DId and DName='math'

Figure 4.2 Student and Dept Tables

Figure 4.3 Query Tree with Relational Algebra Operators
Figure 4.4 depicts query tree of example SQL expression and another presentation of SQL expression is:

Q5: Product (STUDENT, DEPT)  
Q6: Select (Q5, MajorId=DId )  
Q7: Select (Q6, DName='math')  
Q8: Project (Q7, {SId, SName, DName})

4.2.1.2. Scans and Plans

Supported three relational algebra operators (select, product, project) works with “scans” and “plans” interfaces on query layer. We have a two different scan type.

Scan: Objective of scan is access to records on database for reading. We can see interface of Scan in Figure 4.5.

```java
class Scan {
    public void beforeFirst();
    public boolean next();
    public void close();
    public Constant getValue(String fieldName);
    public int getInt(String fieldName);
    public String getString(String fieldName);
    public boolean hasField(String fieldName);
}
```

**Figure 4.4 Scan Interface**

UpdateScan: This interface provides us insert, update and delete operations on database tables. We can see UpdateScan interface in Figure 4.6.
For managing scans and making effective query plans, SimpleDB uses plans. Plan interface provides us accessing statistical data and an open method which returns a result Scan of previous relational algebra. We can see interface of Plan in Figure 4.7.

SelectScan, ProjectScan and ProductScan classes perform in order of select, project and product operators. There is a TableScan under these three scans. Table scan provides relation to database table. TableScan derive from UpdateScan to insert, delete and update records on database tables. And also SelectScan derive from UpdateScan to update specified records on database tables. Furthermore, ProjectScan and ProductScan derive from Scan, not from UpdateScan. Because, when we use product (join) or project, update operation cannot be done on resultset of these relational algebra operators.

We have to add new plans and scans to implement sort relational algebra operator. So, we can design external sort algorithms.
4.2.2. Parser

The parser is responsible for ensuring that its SQL expression is syntactically and semantically correct. If it is correct, then parser converts the expression to QueryData class. Tables, columns and conditions are held structural form in this QueryData class.
5. EXTERNAL SORTING ON SIMPLEDB

Sort scan must be top of the relational algebra tree. Which means sort operator have to start after the other relational algebra operators. Select, project and product operators turn a result set and sort operator sorts this result set.

5.1. Materialize Plan

Every operator implementation that we have seen so far has had the following characteristics:

- Records are computed one at a time, as needed, and are not saved.
- The only way to access previously-seen records is to recompute the entire operation from the beginning.

In this chapter we will consider implementations which materialize their input. Such implementations compute their input records when they are first opened, and save the records in one or more temporary tables. We say that these implementations preprocess their input, because they look at their entire input before any output records have been requested. The purpose of this materialization is to improve the efficiency of the ensuring scan.

Materialization is a two-edged sword. On one hand, using a temporary table can significantly improve the efficiency of a scan. On the other hand, creating the temporary table can be expensive:

- The implementation incurs block accesses when it writes to the temporary table.
- The implementation incurs block accesses when it reads from the temporary table.
- The implementation must preprocess its entire input, even if the JDBC client is interested in only a few of the records.
A materialized implementation is useful only when these costs are offset by the increased efficiency of the scan.

5.1.1. Implementing Materialize Plan

The materialize operation is implemented in SimpleDB by the class MaterializePlan, whose code appears in Figure 5.1.
package simpledb.materialize;

import static simpledb.file.Page.BLOCK_SIZE;
import simpledb.tx.Transaction;
import simpledb.record.*;
import simpledb.query.*;

public class MaterializePlan implements Plan {
    private Plan srcplan;
    private Transaction tx;

    public MaterializePlan(Plan srcplan, Transaction tx) {
        this.srcplan = srcplan;
        this.tx = tx;
    }

    public Scan open() {
        Schema sch = srcplan.schema();
        TempTable temp = new TempTable(sch, tx);
        Scan src = srcplan.open();
        UpdateScan dest = temp.open();
        while (src.next()) {
            tx.incrementSortRecordCount();
            dest.insert();
            for (String fldname : sch.fields())
                dest.setVal(fldname, src.getVal(fldname));
        }
        src.close();
        dest.beforeFirst();
        return dest;
    }

    public int blocksAccessed() {
        // create a dummy TableInfo object to calculate record length
        TableInfo ti = new TableInfo("", srcplan.schema());
        double rpb = (double) (BLOCK_SIZE / ti.recordLength());
        return (int) Math.ceil(srcplan.recordsOutput() / rpb);
    }

    public int recordsOutput() {
        return srcplan.recordsOutput();
    }

    public int distinctValues(String fldname) {
        return srcplan.distinctValues(fldname);
    }

    public Schema schema() {
        return srcplan.schema();
    }
}
The open method preprocesses its input, creating a new temporary table and copying the underlying records into it. The values for methods recordsOutput and distinctValues are the same as in the underlying plan. The method blocks accessed returns the estimated size of the materialized table. This size is computed by calculating the records per block (RPB) of the new records and dividing the number of output records by this RPB. Note that blocks accessed does not include the preprocessing cost. The reason is that the temporary table is built once, but may be scanned multiple times. Also note that there is no MaterializeScan class. Instead, the method open returns a tablesan for the temporary table. The open method also closes the scan for the input records, because they are no longer needed.

5.2. External Sorting Approach of SimpleDB’s Author

In this chapter we will examine SortScan and SortPlan which written by Edward Sciore. SortScan and SortPlan’s logic is as follows:

Firstly, current record pointer (CRP) goes to the first record of resultset which relational algebra operators like select, product and project created. Algorithm proceeds on records one by one. If current record is greater than previous record, this record is written in the temporary table of previous record. Otherwise, the temporary table of previous record is closed and a new temporary table is opened. And this record is written in new temporary table.

After all records are preceded, temporary tables are merged in pairs. In every iteration two temporary tables are merged to one temporary table. Iteration ends when one or two temporary tables are remained. This is for avoiding last merge iteration. Sort scan handles last merge iteration. Thus some disc accesses are avoided.

Code for this algorithm place in SortPlan and SortScan classes.

5.3. Preparatory Work

In this project, we want to implement external sorting algorithms on SimpleDB. So, we have to add some codes.
• We have to introduce order by operator (in SQL) to parser layer of simpleDB.
• The columns, which will be sort, must be presenced in QueryData class in structural form.
• If sorting will be happened, sorting plans must be called by BasicQueryPlanner class.
• Calculating number of accesses per file on FileMgr layer.
• Calculating duration of transaction with adding start time and end time to Transaction class.
• Calculating number of unsorted records on each transaction.
• We may have to compare, copy and exchange congeneric records on different scans. So, RecordExchange class is written for these purposes.
• StepCalculator class is written. It splits any number to two close integer multipliers. For example; from 29, 6 and 5; from 49, 7 and 7; from 50, 8 and 7; etc.

5.4. N-Way Merge Sort
We improved sorting approach of SimpleDB’s author. In this approach, we don’t close temporary tables. Every temporary table is controlled, if record is fixed for temporary table, for each record. If the record is not written in any open temporary tables, we have to open new one and write the record in it. At the end, every temporary tables are merged in one step. Not like the previous one, which two temporary tables merge in every step. This approach’s class name is NWayMergePlan in code.

5.5. K-Way Merge Sort
While implementing k-way merge algorithm on SimpleDB, we should know memory size and number of unsorted records. With this information, we know how many runs will appear. We sort each run separately first. In this step, there is a condition: run count must be less than buffer size for k-way merge theory. Order of result set determines number of runs in n-way merge plan. But in this algorithm number of the records determines run count. So, complexity of algorithm is reduced.
In this algorithm record’s places, which block number and slot number, are calculated with details. By the way only these blocks are taken to memory. This method provides us determining accessed blocks in absolute way. This way avoid any buffer management business by calculating absolute block numbers.

Also in this algorithm, we need two temporary tables. First table is used to materialize records and sort these records in it. Second table is used to merge parted and sorted records in destination table.

5.6. Multi-Step K-Way Merge Sort
We decrease disc access and increase transfer time in multi-step k way merge algorithm.

\[ k = \sqrt[3]{N} \]

This formula is used to determine optimum values. N refers to number of runs, x refers to number of step and k refers to how many ways we will use. Moreover, this algorithm decreases time of finding next records.

In this algorithm, three temporary tables are used. First table is used to materialize records and sort these records in small runs. Second table is used to sort these small runs in bigger runs. Third table is used to convert bigger runs into one destination table.

Multi-step k-way merge is different from k-way merge. In multi-step k-way merge, merging takes two steps and creates bigger runs. Multi-step k-way merge algorithm reduces complexity. So, run time of SQL expressions takes much less time than k-way merge algorithm.
6. REPLACEMENT SELECTION

Replacement Selection (RS) is an external sorting algorithm, based on the merge sort. The objective of RS is to sort a stream of records as they come (usually from secondary storage), producing another stream of released data records called run, which is sorted. Generated runs are always at least as large as the available memory, so it is at least as good as Load-Sort-Store (LSS) in terms of generated run length.

6.1. Heap

Before introducing RS, we describe some previous concepts. The RS algorithm uses a tree-based data structure called heap. A tree is a subtype of a more general entity called graph. A graph G is an ordered pair of sets G = (V, E). The elements of the set V are called vertices or nodes and the elements of the set E are called edges, and are subsets of V of size 2. Two edges u, v ∈ V are called adjacent if the pair (u, v) is in the set E. We will only consider simple graphs, which are those that do not have nodes adjacent to them and each edge appears at most once. Figure 6.1 shows an example of a graph, where nodes have been labeled using integers from 1 to 7.

![Figure 6.1 An Example Graph](image)

We will use a subtype of graph called tree. A tree is a connected graph without cycles. There are several other definitions of tree, however, all of them are equivalent. For instance, a tree is a connected graph where there is a unique path between every pair of
nodes. For the interested reader, more definitions and properties of trees can be found in [4].

A convenient form of describing a tree is classifying the nodes following the symmetric relation parent of and child of. In order to define the relation, we select an arbitrary node of the tree and designate it as the root node. When then have what is called a rooted tree. The parent node of a node \( u \) is the node connected to it in the unique path between the root node and \( u \). A child node of a node \( u \) is a node whose parent is \( u \). Note that in a rooted tree, the root has no parent and all other nodes have a single parent. If a node does not have any children, it is said to be a leaf node.

In a rooted tree, the depth of a node \( u \) is the length of the path between \( u \) and the root. The height of the tree is the maximum of the depth of each node.

An important set of trees are binary trees, which are trees with the property that each node has at most two children. If all nodes have exactly two children except possibly one, then the tree is called complete binary tree.

A data structure is a way to store and organize data so it can be accessed. A tree-based data structure maps a set of records to the set of nodes of a tree assigns a record to each node of a tree. A heap is a tree-based data structure that stores a set of records having a total order, denoted by \( \leq \). There are several variations of the heap data structures. The most common, the binary heap, uses a complete binary tree. A heap stores records in the nodes of the tree satisfying the heap property, namely, that if a node \( v \) is a child node of \( u \), then these records are such that \( u \leq v \). This means that the record stored at the root node, called top record, is always the smallest record according to the total order defined on the records. If the order relation is changed from \( \leq \) to \( \geq \), the heap is called max heap because the top record is the greatest record stored in the heap. If the order relation is the usual \( \leq \), the heap is sometimes called min heap to distinguish it from max heaps. Figure 6.2 shows an example of a completely filled max heap.

Replacement selection stores the records in a binary min heap in memory.
6.1.1. Operations

Heaps implement two operations, adding a record to it and popping the top record.

Adding a record

In order to add a record to a binary heap, a procedure named upheap is used. The upheap procedure is the following: the record is added at the bottom level of the heap, keeping the heap binary and complete. However, it is possible that the new record violates the heap property. A sequence of swaps is needed to restore this property. The process starts in the new node, which is compared to its parent. If they are in the wrong order, that is, if the heap condition is not satisfied, they are swapped. This comparison with the parent node goes on until the node is in the correct order with respect to its parent or the root node of the tree is reached.

It is possible to prove by induction the correctness of the upheap procedure: when the upheap procedure ends, the resulting tree is a heap, that is, the heap condition is satisfied everywhere in the tree. The induction hypothesis is that after the k-th step, the heap condition is satisfied in for every record in the last k+1 levels of the tree. When the k+1-th step begins, the heap condition may only be violated for a record in the level k+2 from the end, per the induction hypothesis. Only the record in this level that belongs to the path between the root and the new record can violate the heap condition, because the rest of the heap is identical to the heap before the new inserted node. The k +1 step swaps this record with one of its children if necessary, so after completion of this step, the heap condition is met everywhere in the last k + 2 levels of the heap. Before the first
step, the last level consists of records without children, so the heap condition is vacuously met, completing the proof.

As an example, consider the heap shown in Figure 6.2. This heap stores integers with the total order \((N, \geq)\), which means that it is a max heap. The process of adding a new record, 91, is depicted in Figure 6.3. When the new record is inserted, the 91 is first added at the end of the heap. Since the last level of the heap is already full, a new one is created with the new record, as shown in Figure 6.3(a). Now, the new record has to be compared with its parent. Since it is a max heap, the heap condition says that parent records need to be larger than child records, and in this case this condition is not met, since \(66 < 91\), so they are swapped. The current state of the heap is shown in Figure 6.3(b). The heap condition must be checked with the new parent of 91, which is 88, and the condition is still not met. 91 is now swapped with 88 and the result is shown in Figure 6.3(c). The new parent of 91 is 93, which is larger, so the heap condition is satisfied and the process of adding the new record to the heap ends. As it can be checked in Figure 6.3(c), the heap condition is satisfied for all records in the heap, as it is assured by the correctness of the upheap procedure.

![Figure 6.3 The Process of Adding the Record 91 to the Heap](image)

The complexity of adding a record depends on the number of swaps. This number is bounded by the height of the tree. If there are \(n\) records stored on the heap, the depth size is \(O(\log n)\). So, assuming that two records can be swapped in constant time, which can be done by swapping values when the record size is small or by swapping pointers, and that the time needed to compare two records is negligible in front of the time
needed to swap two records, the process of adding a record to the heap has a time complexity of $O(\log n)$.

**Popping the Top Record**

In order to delete the root record of the heap, a procedure named downheap is used. The downheap procedure is the following: the root record is replaced with the last record on the last level of the tree. The record at the root node is then compared to its children, and if the heap condition is not met, it is swapped with the larger of the two children (smallest in a min heap). This is repeated until the heap condition is met or the last level of the tree is reached.

The proof of correctness of the downheap procedure is very similar to the proof of correctness of the upheap procedure, but changing the induction hypothesis by “After the $k$-th step, the heap condition is satisfied everywhere in the first $k + 1$ levels of the tree”.

As an example, consider removing the top record of the heap in Figure 6.3(c). The process is depicted in Figure 6.4. The first step is to remove the top record 93, and put in its place the last record of the last level, 66 in this case. The result of this step is shown in Figure 6.4(b). Now, the heap condition is not satisfied, because 66 is not greater than both its children, so it is swapped with the largest of the two child records, 91. The heap is now as shown in Figure 6.4(c). The second step is repeated, and since the heap condition is again not met, 66 is swapped with the largest of its two child records, 88. The heap is now as shown in Figure 6.4(d). Since 66 has no child the process ends here. It can be checked that in the end the heap condition is satisfied for all records.

The number of swaps to be done is at most the height of the tree, so deleting the top record has a time complexity of $O(\log n)$, the same as adding a record.

The action of deleting the top record of a heap is referred to as popping the top record. Note that this operation always retrieves the maximum element of the record set stored in the max heap (minimum in a min heap).
6.1.2. Implementation

Heaps are stored contiguously in memory as arrays because of efficiency. The array is one of the simplest data structures: the records are stored as a sequence and they are accessed through a set of integer indexes. An n-dimensional array uses n integers to index each record. The simplest case is the one-dimensional array, in which records are indexed using only one integer. Usually if the array has n records stored, the index is between 0 and n−1. Vectors and matrices are typically implemented as one and two-dimensional arrays, and hence arrays are also referred to as vectors and matrices.

In order to use a one-dimensional array to store a heap, each node is labeled with an integer, starting with 0 for the root node and assigning the numbers in order level by level, left to right. If a node has the label i, its parent node has the label [(i−1)/2], and its children nodes have labels 2i + 1 and 2i + 2. This is true for every complete binary tree with any number of nodes. Thus, a heap is stored in memory using an array of records,
having each record indexed by the integer indicated by its label in the heap. In Figure 4.5, we depict a complete binary tree, representing a heap, and the array storing the heap.

Since the operations of access and modification of records in an array can be executed in constant time, this implementation allows for $O(\log n)$ time complexity addition and deletion of records in the heap.

### 6.2. Heap Sort

Heap sort is an internal sorting algorithm that uses a heap to sort records initially stored in an array. Heap sort is implemented with the aid of a heap in addition to the input array. The heap is used to sort the records while the array stores them already sorted.

The algorithm performs two steps. The first step consists of inserting all $n$ records into the heap, one by one. Once this process is finished, the top record of the heap is the smallest of the $n$ records. This record is removed from the heap and put in the first position of the array, because it is the smallest record due to the heap property. The new top record of the heap is the second smallest record. This record is removed from the heap and inserted into the second position of the array. This process is repeated $n$ times: at each step, the top record of the heap is removed and inserted into the next empty position of the heap. Once the heap is empty, the array contains the $n$ records already sorted.
In a heap with \( n \) records, the operations of adding and deleting a record require \( O(\log n) \) time. When sorting \( n \) records using heap sort, each record is inserted and deleted one time, and written to the output array. When the \( i \)-th record is inserted, the heap has \( i \) records, so the insertion cost is \( O(\log i) \). When this same record is removed, the heap has also size \( i \), so the removal cost is also \( O(\log i) \). Writing a record to the array has a constant cost, that is, \( O(1) \). Thus, heap sort has a worst-case running time of

\[
\sum_{i=1}^{n} (2 \cdot O(\log i) + O(1)) = \sum_{i=1}^{n} O(\log i) \leq \sum_{i=0}^{n} O(\log n) = O(n \log n)
\]

6.3. Replacement Selection

Replacement Selection (RS) is a run generation algorithm based on heap sort that can be applied if the set of records to be sorted does not fit into memory. It uses a heap to store records. At each step, the top record of the heap is removed and put in the output, like heap sort does. Once a record has been removed, a new record is read from the input and inserted into the heap.

The main difference between heap sort and RS is the fact that, at each step, a new record is inserted into the heap. Another difference is that the output is written directly into a run, which is stored on disk, in order to have more internal memory available for the heap. This is possible because each time a record is removed from the heap, it has to be appended to the output and it is not needed again during the run generation, so there is no need to store all the sequence of output records in internal memory. The runs are written sequentially to disk, and there is no random access. This is a limitation of magnetic tapes, as they only allow sequential reading or writing. Hard disks allow random access, but sequential reading and writing is much faster.

If a record read from the input is smaller than the last record used as output in the current run, it is not be possible to use it as part of the current run, because records being output are already larger. This situation arises with RS and not with heap sort.
because RS adds new records to the heap. In this case, the record is marked as belonging to the next run. When a record marked is inserted into the heap, it is put at the bottom of the heap. In order to do this, it is considered that records belonging to the next run are larger than all records belonging to the current run. Therefore, when the top record of the heap belongs to the next run, all records stored in the heap also belong to the next run. The reasoning is the following: suppose that we had another record belonging to the current run in the heap. This record is smaller than the top record. Thus, the heap condition would be violated somewhere along the path that joins the top record with the record belonging to the current run. Since this is an impossible situation, if the top record belongs to the next run, so do the rest of the records stored in the heap.

The generation of the current run ends when the top record belongs to the next run, and then a new run is started. The algorithm proceeds using the same strategy until there are no more records to read from the input.

6.4. Run Length

The length of the runs generated by a run generation algorithm like RS has an impact on the performance of the merge phase. If we consider a k-way merge as the merging algorithm, older machines stored runs using tapes, so the value of k is bounded by the number of tapes used to store runs. Newer computer are able to perform a k-way merge for small and large values of k, but there is an optimum value of k in terms of performance. Thus, if a run generation algorithm creates larger runs, the total number of runs to be merged decreases and the merge phase has a shorter execution time.

When the input records follow a random uniform distribution, RS generates runs that have an average length equal to the size of the available memory.

There is a proof of this fact in [6]. This proof is intuitive because it builds a physical model for the problem, and it is reproduced here.

The proof considers a circular road on which snow flakes drop at a constant rate. A snowplow is continually clearing the snow, as shown in Figure 6.6. Once the snow is
plowed, it disappears from the system. Points on the road may be designated with a real number $0 \leq x < 1$. A snowflake falling on $x$ represents an input value of $x$. That is, we consider input records to be between 0 and 1, without loss of generality.

![Figure 6.6 Circular Road with a Snowplow](image)

The snowplow represents the output of RS, and has a speed inversely proportional to the height of the snow it encounters. The situation is balanced so the quantity of snow in the road is $P$ at all times (so $P$ corresponds to the total memory available). The generation of a run finishes when the snowplow passes through $x=1$.

After operating for a while, this system will approach a stable situation where the snowplow has a constant speed, because of the circular symmetry the road. This means that the snow is at a constant height in front of the snowplow and decreases linearly in front of it. It follows that the amount of snow removed at each revolution is $2P$ (see Figure 6.7). The first triangle represents the snow which is already on the track and has size $P$, and the second one represents snow that will fall while the snow plough is running, and it also has size $P$. 

![Figure 6.7 Stabilized Situation](image)
6.5. Determining the Stage Area Size

When using RS algorithm a piece of main memory is used for stage area. As stage area size changes run lengths are changes too. Problem is that: Does it worth to use biggest stage area in main memory? What is the relationship between stage area size and run lengths? To determine best stage area size we tried different stage area sizes to generate runs in normally distributed records. Results of test place in Table 6.1 and Table 6.2. Results are for 8x main memory and 32x file size.

Table 6.1 Generating and Merging Runs with Different Stage Area Sizes

<table>
<thead>
<tr>
<th>Stage Area</th>
<th>Disc Access Count</th>
<th>Run Time</th>
<th>K (Run Count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>3817</td>
<td>0.690</td>
<td>17</td>
</tr>
<tr>
<td>2x</td>
<td>420</td>
<td>0.742</td>
<td>8</td>
</tr>
<tr>
<td>3x</td>
<td>252</td>
<td>1.049</td>
<td>6</td>
</tr>
<tr>
<td>4x</td>
<td>258</td>
<td>1.420</td>
<td>4</td>
</tr>
<tr>
<td>5x</td>
<td>266</td>
<td>1.952</td>
<td>4</td>
</tr>
<tr>
<td>6x</td>
<td>279</td>
<td>2.333</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.2 Only Generating Runs Phase Results

<table>
<thead>
<tr>
<th>Stage Area</th>
<th>Disc Access Count</th>
<th>Run Time</th>
<th>K (Run Count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>136</td>
<td>0.482</td>
<td>17</td>
</tr>
<tr>
<td>2x</td>
<td>139</td>
<td>0.687</td>
<td>8</td>
</tr>
<tr>
<td>3x</td>
<td>146</td>
<td>0.975</td>
<td>6</td>
</tr>
<tr>
<td>4x</td>
<td>158</td>
<td>1.646</td>
<td>4</td>
</tr>
<tr>
<td>5x</td>
<td>169</td>
<td>2.048</td>
<td>4</td>
</tr>
<tr>
<td>6x</td>
<td>180</td>
<td>2.387</td>
<td>3</td>
</tr>
</tbody>
</table>

In Table 6.1 we see when stage area increases run count and disc access count decrease (later increases little), meanwhile run time increases too. Here we must think that: When we move from 1x stage area to 2x stage area, does it worth to decrease 17 runs to 8 runs in 0.152 seconds? And at next step same question is asked ourselves. In which step we say no, it doesn’t worth. Secondly by comparing Table 6.1 and Table 6.2 we can see
both how long does merging takes and how long generating runs phase takes. Seeing Table 6.1 and Table 6.2 together can help us to determine stage area percentage better.

Additionally, to handle worst case scenario, that records are in reverse order, RS algorithm must keep run lengths as long as possible. So RS should increase stage area size, note that: RS algorithm generates \((\text{file size} \div \text{stage area size})\) pcs runs.

On mostly normalized distributed test results with better and worse case scenarios nearly 30 percent of main memory usage as stage area best worth. So in our example we must choose step 2 which have 2x stage area size.
7. CONCLUSION

Result of our project is a comparison table of different algorithms. Abreviation list for the result table is:

ESMS: Edward Sciore’s Merge Sort
KWMS: K-Way Merge Sort
MSKWMS: Multi-Step K-Way Merge Sort
RSS: Replacement Selection Sort
MSRSS: Multi Step Replacement Selection Sort

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>File Size (Byte)</th>
<th>Memory Size (Byte)</th>
<th>Record Count</th>
<th>K (Run Count)</th>
<th>Temp Table (File Count)</th>
<th>Disc Access Count</th>
<th>Run Time (Second)</th>
<th>Step Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESMS</td>
<td>3200*2</td>
<td>3200</td>
<td>128</td>
<td>60</td>
<td>118</td>
<td>513</td>
<td>0.618</td>
<td>5</td>
</tr>
<tr>
<td>KWMS</td>
<td>3200*2</td>
<td>3200</td>
<td>128</td>
<td>2</td>
<td>2</td>
<td>149</td>
<td>0.372</td>
<td>1</td>
</tr>
<tr>
<td>MSKWMS</td>
<td>3200*2</td>
<td>3200</td>
<td>128</td>
<td>2</td>
<td>2</td>
<td>149</td>
<td>0.372</td>
<td>1</td>
</tr>
<tr>
<td>RSS</td>
<td>3200*2</td>
<td>3200</td>
<td>128</td>
<td>4</td>
<td>3</td>
<td>118</td>
<td>0.406</td>
<td>1</td>
</tr>
<tr>
<td>ESM</td>
<td>3200*8</td>
<td>3200</td>
<td>512</td>
<td>246</td>
<td>490</td>
<td>2703</td>
<td>2.596</td>
<td>7</td>
</tr>
<tr>
<td>KWMS</td>
<td>3200*8</td>
<td>3200</td>
<td>512</td>
<td>8</td>
<td>2</td>
<td>939</td>
<td>1.311</td>
<td>1</td>
</tr>
<tr>
<td>MSKWMS</td>
<td>3200*8</td>
<td>3200</td>
<td>512</td>
<td>3*3</td>
<td>3</td>
<td>786</td>
<td>1.338</td>
<td>2</td>
</tr>
<tr>
<td>RSS</td>
<td>3200*8</td>
<td>3200</td>
<td>512</td>
<td>17</td>
<td>3</td>
<td>7708</td>
<td>1.659</td>
<td>1</td>
</tr>
<tr>
<td>MSRSS</td>
<td>3200*8</td>
<td>3200</td>
<td>512</td>
<td>5*4</td>
<td>4</td>
<td>676</td>
<td>1.517</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.1 Comparison of External Sorting Algorithms Part One
When we need to sort data, which have size more than main memory, we need to do external sorting. To solve this problem some algorithms are thought. In our project we researched these algorithms and implemented these algorithms on SimpleDB, which is educational open source database management system. At the end of our project we created a table to compare external sorting algorithms, place in Table 7.1 and 7.2. Note that, we used normalized distributed records in our tests.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>File Size (Byte)</th>
<th>Memory Size (Byte)</th>
<th>Record Count</th>
<th>K (Run Count)</th>
<th>Temp Table (File Count)</th>
<th>Disc Access Count</th>
<th>Run Time (Second)</th>
<th>Step Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESMS</td>
<td>3200*36</td>
<td>3200</td>
<td>2304</td>
<td>1161</td>
<td>2320</td>
<td>13716</td>
<td>12.546</td>
<td>13</td>
</tr>
<tr>
<td>KWMS</td>
<td></td>
<td></td>
<td></td>
<td>36</td>
<td>2</td>
<td>112631</td>
<td>8.342</td>
<td>1</td>
</tr>
<tr>
<td>MSKWMS</td>
<td></td>
<td></td>
<td></td>
<td>6*6</td>
<td>3</td>
<td>4858</td>
<td>6.240</td>
<td>2</td>
</tr>
<tr>
<td>MSKWMS</td>
<td></td>
<td></td>
<td></td>
<td>4<em>3</em>3</td>
<td>4</td>
<td>4722</td>
<td>6.415</td>
<td>3</td>
</tr>
<tr>
<td>MSKWMS</td>
<td></td>
<td></td>
<td></td>
<td>3<em>3</em>2*2</td>
<td>5</td>
<td>5508</td>
<td>6.878</td>
<td>4</td>
</tr>
<tr>
<td>RSS</td>
<td></td>
<td></td>
<td></td>
<td>72</td>
<td>3</td>
<td>218703</td>
<td>12.226</td>
<td>1</td>
</tr>
<tr>
<td>MSRSS</td>
<td></td>
<td></td>
<td></td>
<td>9*8</td>
<td>4</td>
<td>18022</td>
<td>7.172</td>
<td>2</td>
</tr>
<tr>
<td>MSRS</td>
<td></td>
<td></td>
<td></td>
<td>5<em>4</em>4*4</td>
<td>5</td>
<td>4551</td>
<td>7.219</td>
<td>3</td>
</tr>
<tr>
<td>MSRSS</td>
<td></td>
<td></td>
<td></td>
<td>3<em>3</em>3*3</td>
<td>6</td>
<td>4967</td>
<td>7.317</td>
<td>4</td>
</tr>
<tr>
<td>ESMS</td>
<td>3200*64</td>
<td>3200</td>
<td>4096</td>
<td>2023</td>
<td>4044</td>
<td>24649</td>
<td>23.344</td>
<td>13</td>
</tr>
<tr>
<td>KWMS</td>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>2</td>
<td>353574</td>
<td>18.909</td>
<td>1</td>
</tr>
<tr>
<td>MSKWMS</td>
<td></td>
<td></td>
<td></td>
<td>8*8</td>
<td>3</td>
<td>28219</td>
<td>11.124</td>
<td>2</td>
</tr>
<tr>
<td>MSKWMS</td>
<td></td>
<td></td>
<td></td>
<td>4<em>4</em>4*4</td>
<td>4</td>
<td>9095</td>
<td>11.428</td>
<td>3</td>
</tr>
<tr>
<td>MSKWMS</td>
<td></td>
<td></td>
<td></td>
<td>3<em>3</em>3*3</td>
<td>5</td>
<td>10803</td>
<td>12.485</td>
<td>4</td>
</tr>
<tr>
<td>RSS</td>
<td></td>
<td></td>
<td></td>
<td>131</td>
<td>3</td>
<td>672584</td>
<td>28.132</td>
<td>1</td>
</tr>
<tr>
<td>MSRSS</td>
<td></td>
<td></td>
<td></td>
<td>12*11</td>
<td>4</td>
<td>66701</td>
<td>13.547</td>
<td>2</td>
</tr>
<tr>
<td>MSRS</td>
<td></td>
<td></td>
<td></td>
<td>6<em>5</em>5</td>
<td>5</td>
<td>11725</td>
<td>12.752</td>
<td>3</td>
</tr>
<tr>
<td>MSRSS</td>
<td></td>
<td></td>
<td></td>
<td>4<em>4</em>3<em>3</em>3</td>
<td>6</td>
<td>10427</td>
<td>13.271</td>
<td>4</td>
</tr>
</tbody>
</table>
Firstly ESMS and RSS algorithms’ run count depends on order of the input records. KWMS and MSKWMS algorithms are independent from input order. ESMS algorithm is very simple but mostly creates very small runs and have long run time. If we think KWMS and MSKWMS, for small file sizes (for Table 7.1) disc access counts and run times are near. When file sizes increase (for Table 7.2) KWMS cannot use main memory efficiently, so disc access count and run time increase enormously. If we compare MSKWMS step counts (from 2 to 4), when step count increases disc access count decrease (later it increases too, its cause is inefficient main memory usage), but run time increases too. This is because data transfer time increases more than disc access count decrement. If we compare MSKWMS and MSRSS, when input records are normalize distributed and reverse ordered MSKWMS’s run count is less then MSRSS. When input records are ordered MSRSS’s run count is greater than MSKWMS. Additionally when we look at part of Table 7.2 which have 4096 record count. MSRSS is fastest with 3 steps, because run count is enormously big, thus RSS with 3 steps can use main memory efficiently.

So in RSS algorithm, if input records are exactly ordered then run count will be one and fastest algorithm will be RSS. If input records are exactly reverse ordered then (file size / stage area size) pcs run is generated and slowest algorithm will be RSS. If we think to sort records in certain way MSKWMS must be our choice. So all kinds of external sorting algorithms have different advantages.

As result of our table and tests algorithm choice depends on efficient main memory usage and input order. You can see this dependency in Figure 7.1.
Figure 7.1 Algorithm Choice Dependency
REFERENCES
APPENDIX

These are classes of SimpleDB. Diagrams are shown package by package (layer by layer).

Part 1:
Part 2:

```plaintext
simplifiedb.query
Constant, IntConstant, StringConstant,
Expression, ConstantExpression, FieldNameExpression
Term, Predicate
Scan, TableScan, UpdateScan, SelectScan, ProjectScan, ProductScan
Plan, TablePlan, SelectPlan, ProjectPlan, ProductPlan

simplifiedb.metadata
StatInfo, IndexInfo,
StatMgr, IndexMgr,
TableMgr, ViewMgr,
MetaDataMgr

simplifiedb.record
RID, Schema, TableInfo,
RecordFormatter, RecordPage, RecordFile

simplifiedb.tx
BufferList, Transaction

simplifiedb.tx.concurrency
LockTable, ConcurrencyMgr

simplifiedb.tx.recovery
LogRecord, SetIntRecord, SetStringRecord
StartRecord, CheckPointRecord,
RecoveryRecord, CommitRecord

simplifiedb.buffer
PageFormatter, Buffer,
BasicBufferMgr, BufferMgr

simplifiedb.log
LogIterator, BasicLogRecord, LogMgr

simplifiedb.file
Block, Page, FileMgr
```