PERFORMANCE ANALYSIS OF EXTERNAL SORTING ALGORITHMS

BY
DEBATOSH DEBNATH

A THESIS SUBMITTED TO THE DEPARTMENT
OF COMPUTER SCIENCE AND ENGINEERING, BUET, DHAKA,
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE IN ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
OCTOBER, 1993

"R"
001.61
1993
DEB

PERFORMANCE ANALYSIS OF EXTERNAL SORTING ALGORITHMS

A thesis submitted by

DEBATOSH DEBNATH
Roll no 901827F, Registration no 84193,
for the partial fulfillment of the degree of
M. Sc. Engg. in Computer Science and Engineering.
Examination held on October 12, 1993.

Approved as to style and contents by:

[illegible] 12.10.93

DR. MOHAMMAD KAYKOBAD Assistant Professor Department of Computer Science & Engineering BUET, Dhaka—1000, Bangladesh	Chairman (Supervisor)

[illegible] 12/10/93

DR. A. B. M. SIDDIQUE HOSSAIN Professor and Head Department of Computer Science & Engineering BUET, Dhaka—1000, Bangladesh	Member (Ex officio)

M. S. Alam 12/10/93

DR. MD. SHAMSUL ALAM Associate Professor Department of Computer Science & Engineering BUET, Dhaka—1000, Bangladesh	Member

[illegible] 12.10.93

DR. MD. LUTFAR RAHMAN Associate Professor and Chairman Department of Computer Science Dhaka University Dhaka—1000, Bangladesh	Member (External)

CERTIFICATE

This is to certify that the work presented in this thesis is done by me under the supervision of Dr. Mohammad Kaykobad, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka. It is also certified that neither this thesis nor any part thereof has been submitted elsewhere for the award of any degree or diploma.

Countersigned by the supervisor

[illegible]

Dr. Mohammad Kaykobad

Signature of the candidate

[illegible]

Debatosh Debnath

ACKNOWLEDGEMENTS

The author would like to express his sincere gratitude and profound indebtedness to his supervisor Dr. Mohammad Kaykobad, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka, for his constant guidance, helpful advice, invaluable assistance and endless patience throughout the progress of this work, without which the work could not have been completed.

The author gratefully acknowledges the kind support and encouragement extended to him by Dr. A. B. M. Siddique Hossain, Professor and Head, Department of Computer Science and Engineering, BUET, Dhaka, during the course of this work.

The author conveys his sincere thanks to Dr. Md. Shamsul Alam, Associate Professor, Department of Computer Science and Engineering, BUET, Dhaka, for his kind cooperation during the progress of this study. The author also expresses his sincere gratitude to Dr. Md. Lutfar Rahman, Chairman, Department of Computer Science, Dhaka University, Dhaka, for kindly acting as external examiner.

Finally, the author acknowledges with hearty thanks, the all out cooperation of the faculty members and his friends of the Computer Science and Engineering Department, BUET, Dhaka.

ABSTRACT

Sorting is a classical problem in computer science, whereas external sorting is what is needed in almost every application; whether it be commercial or statistical.

In this thesis a set of important algorithms of external sorting have been thoroughly studied theoretically and their relative performances have been evaluated by carefully designing experiments. In these experiments we vary two important parameters namely, number of files (*F*) and output to input buffer ratios (*BR*) to find the best values at which total computational time for sorting is minimum. This study shows that polyphase merging technique is the best among these algorithms, for any given *BR* values and for permissible values of *F*. It has been further revealed that for small number of records, the order of merging has little effect on the total sorting time. However, this is not true as the number of records increases in which case the order of merging becomes a dominant parameter.

Since balanced merge makes complete passes over all the records, computational time is piecewise linear with different slopes in different intervals of number of records with a jump at the end of each interval. Moveover, with the increase in the number of records slope also increases. In case of polyphase and cascade merging techniques computational time grows faster than linear curves.

CONTENTS

INTRODUCTION

General:

Sorting is the rearrangement of items into ascending or descending order. It is hard to imagine using a dictionary if its words were not listed in alphabetical order. In a similar way, the order in which items are stored in a computer memory (primary or secondary) has a profound influence on the speed and simplicity of algorithms which manipulate them.

Let us define sorting a little more clearly, and introduce some terminology commonly used in this area. Suppose we are given N items

$$R_1, R_2, R_3, \ldots\ldots, R_N$$

to be sorted; we shall call them records. A *record* is a collection of information items about a particular entity. For example, a record may consist of information about a passenger on an airplane flight, or an article sold at a retail distribution store. A *field* of a record is a unit of meaningful information about an entity. The different items of a passenger record may be passenger's name, address, seat number, and menu restrictions.

A collection of records involving a set of entities with certain aspects in common and organized for some particular purpose is called a *file*. In the above case, the entire collection of N records will be called a file or the collection of all passenger records for the passengers on a particular flight constitutes a file. In a data file, the field which uniquely identifies a record is called the *key field* or simply *key*, K_j.

In the passengers file, individual passengers records can be uniquely identified by the passenger's name, assuming duplicate names do not occur for a particular flight. If required, the seat number item can also be used as a key, since seat numbers are uniquely assigned for a given flight.

It is common practice to order the records in a file according to key. Therefore, if the passenger's name is selected as the key item, the record for Adams appears before the record for Chowdhury, which appears before the record of Kato in lexical ordering by surname. There may be additional information (different fields), besides the key in a record. This extra information has no effect on the sorting process except that it remains with the same record.

Mathematically, the goal of sorting is to determine a permutation $p(1)\,p(2)\ldots\ldots p(N)$ of the records, which puts the keys in non-decreasing (*i.e.*, increasing or same) order:

$$K_{p(1)} \leq K_{p(2)} \leq \ldots\ldots \leq K_{p(N)}.$$

Types of Sorting:

Sorting can be classified generally into two types: internal sorting and external sorting. In *internal sorting*, the volume of the records to be sorted must be small enough to hold in the computer's high-speed random access memory at once. Internal sorting allows more flexibility in the structuring and accessing of the data and there are many good algorithms for internal sorting. Again, *external sorting* is applicable when there are more records than can be hold in memory at once. In this situation, sorting is done in a piecemeal fashion and frequent accessing of the external storage devices is necessary. But accessing the external devices is very slow, and as a result external sorting is too time consuming, and external sorting algorithms show how to live with this stringent accessing constraints.

Importance of the Study:

It is well known that, sorted computer output are more suitable for human understanding. Sorting is also an aid to searching and it leads to an efficient searching algorithm. If we only need to search a file once, it is faster to do a sequential search than to do a complete sort of the file; but if we need to make repeated searches in the same file, it is wiser to put the records in order.

External sorting is quite different from internal sorting, even though the problem in both cases is to sort the records of a given file into non-decreasing order of their keys. The data structures must be arranged so that comparatively slow peripheral memory devices (disks, drums, tapes etc.) can quickly cope with the requirements of the sorting algorithms. Because there is a great emphasis on minimizing the input/output time, comparatively primitive data structures are used for external sorting. Consequently most of the available internal sorting techniques are virtually useless for external sorting and it is necessary to consider the whole question separately.

Due to its slow response time, the efficient storage accessing on external devices is a subject of concentration. It is usually economical to access a list of information in sequence from the beginning to end, instead of skipping around at random in the list, unless the list is small enough to be held in the high-speed random-access memory of a computer.

As a very simple example, we can consider the formation of a large data file from the collected data. Usually the collected data are stored in small internally sorted files. To form a final data file from these individual files, we have to combine them. Then to store this final data file for convenient processing, it is necessary to organize it in a sorted form. But if the volume of the collected data are too large to hold in the computer's memory (which is usual), then none of the internal sorting algorithm will be applicable in this situation and external sorting is the only way to organize the total collected data in a sorted order.

External sorting has been used mostly for business data processing. As a particular example, let us consider the computerized billing system of a large public utility company, with tens of thousands of subscribers. The master file of this billing system would contain a single record for each subscriber. Again each record would contain several fields, such as subscriber's identification number, name, address, area code, amount due etc. The very practical size of such a master file would be several tens of megabytes. During bill collection, transaction files would be produced to hold transaction records, where each transaction record would contain subscriber's identification number and amount deposited. Periodically it would be necessary to update the master file according to transaction records. If both the files are in random order then it would be very time consuming task to complete the above mission. But if the files are sorted on the identical key field into the same order then it is possible to find all the matching entries in one sequential pass through them, without moving forward and backward. As the master file is large enough to hold in the random-access memory of a computer, external sorting routine is necessary to do this ordering.

For another example, let us consider a file of 8000 records, $R_1 R_2 \ldots . R_{8000}$ to sort, and that each record R_i is 40 words long. We have a computer system which can hold 2000 of these records at once in its high-speed internal memory. Clearly, none of the internal sorting algorithms will be useful in this situation. One fairly obvious solution is to start by sorting each of the four sub-files $R_1 \ldots R_{2000}$, $R_{2001} \ldots R_{4000}$, $R_{4001} \ldots R_{6000}$, $R_{6001} \ldots R_{8000}$, independently, then to merge the resulting sub-files together. Fortunately, the process of merging uses only a very simple data structure, namely linear lists which are traversed in a sequential manner. Hence, merging can be done without difficulty on the least expensive external memory devices.

There are numerous practical applications of external sorting and undoubtedly it deserves to be studied seriously. The process just described, internal sorting followed by "external merging" is very commonly used, and we shall devote most of our study of external sorting to variations of this theme.

Historical Perspective:

Before making any further comment on external sorting algorithms, it may be helpful to put things in historical perspective. A search for the origin of today's sorting techniques takes us back to the nineteenth century (1880), when the first machines for card sorting were invented by Herman Hollerith, a 20 year old employee of the Census Bureau of the United States. The story of Hollerith's early machines and its sorting techniques has been told in interesting details by Truesdell [26].

The idea of merging goes back to card-walloping machine — the *collator*, which was invented in 1938. With its two feeding stations, it could merge two sorted decks of cards into one, in only one pass. The technique for doing this was clearly explained in the first IBM collator manual (April, 1939).

Then computers arrived on the scene and John von Neumann prepared programs for internal merge sorting in 1945. The details of this interesting development have been described in an article by Knuth [19].

To organize large databases using early computers made it natural to think of external sorting. Eckert and Mauchly [6] pointed out that a computer augmented with magnetic tape devices could simulate the operations of card equipment, achieving a faster sorting speed. This progress report described balanced two-way merging (called "collating"), using four magnetic tape units. Mauchly lectured on "*Sorting and Collating*" at the special session on computing presented at the Moore School in 1946, and the notes of his lecture constitute the first published discussion of computer sorting.

Shortly afterwards, Eckert and Mauchly started a company which produced some of the earliest electronic computers, the BINAC and the UNIVAC. At this time it was not at all clear that computers would be economically profitable; computing machines could sort faster than card equipment, but they cost more. Therefore, the UNIVAC programmers, led by F. E. Holberton (see [18]), put considerable effort into the design of first commercial external sorting routines. According to their estimates, 100 million 10-word records could be sorted on UNIVAC in 9000 hours (*i.e.*, 375 days)!

In July 1951, UNIVAC I was designed with features like block read/write, forward or backward read/write capability on tapes, and simultaneous reading/writing computing was also possible.

Seward [23] introduced the idea of replacement selection to produce initial runs longer than high-speed internal memory size. Friend [7] remarked on the expected size of the runs produced by replacement selection. E. F. Moore (see [18]) proved that average expected run length produced by this method is double the memory size.

The comprehensive paper by Friend [7] was a major milestone in the development of both internal and external sorting. Although numerous techniques have been developed since 1956, this paper is still remarkably up-to-date in many respects. Friend gave careful descriptions of different external as well as internal sorting techniques.

The three file case of perfect Fibonacci distribution of runs for polyphase merge had been discovered by Betz [3]. The general pattern for perfect Fibonacci distribution for polyphase merge was developed by Gilstad [9]. Actually the Fibonacci numbers of higher order used in polyphase merge was first studied by Schlegel [22]. Sackman [21] suggested the horizontal method of initial run distribution for polyphase merge to equalize the number of dummy runs on tapes or on files.

The general pattern of cascade merge was actually discovered before polyphase merge by Betz and Carter [4].

Thesis Organization and Objective:

This thesis comprises seven chapters. Chapter one discusses balanced merge techniques. The chapter starts with a discussion on preparation of initial runs — which is a fundamental step in external sorting. Through proper examples, we show the gradual improvements of the balanced merge algorithm.

Chapter two discusses polyphase merge, where initial runs are distributed on files according to perfect Fibonacci numbers of appropriate order. Algorithms are provided for smoother transition of sorting time for consecutive numbers of initial runs. Another type of external sorting algorithm called cascade merge is discussed in detail in chapter three.

Replacement selection algorithm is discussed in chapter four. Through examples and step by step discussions, the algorithm is presented to produce long initial runs, which would be longer than the computer's high-speed internal memory capacity. In chapter five, different practical aspects of external sorting are discussed. Among different aspects, the factors affecting external sorting time and the techniques to reduce them are mainly emphasized.

Chapter six is intended to present the experimental results based on the algorithms discussed so far in the previous chapters. This chapter compares different external sorting strategies based on the experimental results. In chapter seven we make conclusion of our findings and recommend some issues for further research in this direction.

This thesis is an attempt to analyze the performance of external sorting techniques. Mainly we have concentrated our study on time and space complexity of these techniques. In external sorting initial runs are distributed on a number of files. The total sorting time of different algorithms depend on the exact number of files used in the operation. Thus, number of files can be used as an external sorting parameter and our purpose is to find out the response of these algorithms under the variation of this parameter.

External sorting is done in a piecemeal fashion and frequent disk access is necessary. But disk access time is the most significant part of the total time required for disk input/output and it is necessary to reduce the total disk access time. The total disk access time can be reduced by allocating input and output buffers in the computer's high-speed memory. In this case a large amount of data can be read from disk to buffer or written from buffer to disk at once. Therefore, the total number of disk accesses can be reduced. Also, it is possible to allocate buffers in a variety of ways and our aim is to ascertain the behavior of different algorithms under these variations.

We have designed simulation programs to foresee the response of all these external sorting techniques with the variations of different sorting parameters and it is hoped that these programs would be useful to sort any large database. All the simulation experiments have been carried out on an IBM PC compatible machine having 80486 processor of 32 MHz speed with a 240 megabytes disk space.

CHAPTER 1

BALANCED MERGE

1.1 Initial Runs:

The general strategy in external sorting is to begin by sorting small batches of records from a file in internal memory. These small batches or the non-decreasing sequences of records, which are produced by internal sorting phase, are often called *initial runs* or simply *runs* (also called *initial strings* or *run lists*). The size of these initial runs depends directly on how much internal memory is set aside to perform the internal sorting. Actually, the initial runs are produced in a piecemeal fashion. They are stored in a target file from which they are later retrieved and merged together again to form fewer but larger runs. This process of merging runs to form fewer but larger runs continues and eventually terminates with the formation of a single run, which is the required sorted file. Since the number of initial runs ultimately determines the cost of merging, we would like to find some method of creating longer and hence fewer initial runs, which would be discussed in detail in chapter four.

1.2 Sorting by Merging:

Merging means combining two or more ordered files into a single ordered file. For example, we can merge the two ordered files with elements 337, 532, 569, 658, and

051, 356, 573 to obtain a single sorted file containing 051, 356, 337, 532, 569, 573, 658. A simple way to accomplish this is to compare the two smallest items from the two ordered files, output the smallest one, and then repeat the same process. Starting with

$$\left\{\begin{array}{l} 337\ 532\ 569\ 658\ \infty \\ 051\ 356\ 573\ \infty \end{array}\right.$$

we obtain

$$051 \left\{\begin{array}{l} 337\ 532\ 569\ 658\ \infty \\ 356\ 573\ \infty \end{array}\right.$$

then

$$051\ 337 \left\{\begin{array}{l} 532\ 569\ 658\ \infty \\ 356\ 573\ \infty \end{array}\right.$$

and so on. An additional key "∞" has been placed at the end of each input file in this example, so that the merging terminates gracefully.

1.3 Balanced 2-Way Merging:

Perhaps the simplest way to merge initial runs is the balanced 2-way merge. Let us use four files in this process. During the first phase, non-descending runs produced by internal sorting phase are placed alternately on Files 1 and 2, until the input is exhausted. This phase is called the initial distribution phase. Then we merge the runs from these two files, obtaining new runs which are twice as long as the original ones; the new runs are written alternately on Files 3 and 4 as they are being formed. If File 1 contains one more run than File 2, an extra "dummy" run of length 0 is assumed to be present on File 2. Then the contents of Files 3 and 4 are merged into quadruple-length runs recorded alternately on Files 1 and 2. The process continues, doubling the length of the runs each time, until only one run is left (namely the entire sorted file). If S runs were produced during the internal sorting phase, and if $2^{k-1} < S \le 2^k$, this balanced 2-way merge procedure makes exactly $k = \lceil \log_2 S \rceil$ merging passes over all the data.

Let us consider the following sorting problem: we have a file of 5000 records, $R_1 R_2 \ldots . R_{5000}$ to be sorted, and that each record R_i is 20 words long (although the keys K_i are not necessarily this long). We have a computer system which can hold 1000 of these records at once in its high-speed internal memory.

To cope with this situation where 5000 records are to be sorted with an internal memory capacity of 1000 records, we have to prepare five initial runs $R_1 \ldots R_{1000}, R_{1001} \ldots R_{2000}, \ldots\ldots, R_{4001} \ldots R_{5000}$ by independent internal sorting. If we consider 2-way merging, the initial distribution phase of the sorting process places five runs on files as follows:

File 1	:	$R_1 \ldots . R_{1000}$; $R_{2001} \ldots . R_{3000}$; $R_{4001} \ldots . R_{5000}$
File 2	:	$R_{1001} \ldots . R_{2000}$; $R_{3001} \ldots . R_{4000}$
File 3	:	(empty)
File 4	:	(empty)

The first pass of merging then produces longer runs on Files 3 and 4, as it reads Files 1 and 2, as follows:

File 3 : $R_1 \dots R_{2000}$; $R_{4001} \dots R_{5000}$
File 4 : $R_{2001} \dots R_{4000}$

A dummy run has implicitly been added at the end of File 2, so that the last run $R_{4001} \dots R_{5000}$ on File 1 is merely copied on to File 3. The next pass over the data produces:

File 1 : $R_1 \dots R_{4000}$
File 2 : $R_{4001} \dots R_{5000}$

Again that run $R_{4001} \dots R_{5000}$ was simply copied; but if we had started with 8000 records, File 2 would have contained $R_{4001} \dots R_{8000}$ at this point. Finally, after another phase of merging, $R_1 \dots R_{5000}$ is produced on File 3, and the sorting is complete. The total number of records handled during this balanced 2-way merging (except initial distribution) is 5000+5000+5000 = 15,000.

1.4 Balanced Multi-Way Merging:

Balanced merging can easily be generalized to the case of F files, for any $F \geq 3$. We can choose any number P with $1 \leq P < F$, and divide the F files into two "banks," with P files on the left bank and $F - P$ files on the right bank. We can distribute the initial runs as evenly as possible on to the P files in the left bank; then do a P-way merge from the left to the right, followed by $(F - P)$-way merge from right to the left, and so on, until sorting is complete.

In general, if $F = 2m$, we can divide the files into two banks, with m files on each bank. In this situation, if S is the total number of runs produced by internal sorting phase and if $m^{k-1} < S \leq m^k$, then this m-way balanced merge procedure makes exactly $k = \lceil \log_m S \rceil$ merging passes over all the data. Therefore, by considering the initial distribution pass, the total number of passes over the data is $k+1$.

Balanced two-way merging is the special case with $F = 4$ and $P = 2$. Let us reconsider the previous example using more files, taking $F = 6$ and $P = 3$. The initial distribution now gives:

File 1	:	$R_1 \ldots . R_{1000}$; $R_{3001} \ldots . R_{4000}$
File 2	:	$R_{1001} \ldots . R_{2000}$; $R_{4001} \ldots . R_{5000}$
File 3	:	$R_{2001} \ldots . R_{3000}$

The first merging pass produces:

File 4	:	$R_1 \ldots . R_{3000}$
File 5	:	$R_{3001} \ldots . R_{5000}$
File 6	:	(empty)

A dummy run has been assumed on File 3 to do the above merging. The second merging pass completes the job, placing $R_1 \ldots . R_{5000}$ on File 1. In this special case with $F = 6$ is essentially the same as $F = 5$, since the sixth file is used only when $S \geq 7$.

Three-way merging actually requires somewhat more computer processing time than two-way merging, but this is essentially negligible compared to the time needed to read and write. We can get a fairly good estimate of the running time by considering only the total number of records handled during the merging process.

In this three-way merge the given problem required only two passes over the data, compared to three passes when $F = 4$, so the merging takes only about two-thirds as long when $F = 6$. The total member of records handled during this balance three-way merge is 5000+5000 = 10,000.

Balanced merging is quite simple, but if we look more closely, we find immediately that it is not the best way to handle the particular case treated above. Let us consider the problem in a little different way by using four working files and doing two and three-way merging. After initial distribution the file contents would be:

File 1	:	$R_1 \ldots . R_{1000};\ R_{2001} \ldots . R_{3000};\ R_{4001} \ldots . R_{5000}$
File 2	:	$R_{1001} \ldots . R_{2000};\ R_{3001} \ldots . R_{4000}$
File 3	:	(empty)
File 4	:	(empty)

Now doing a two-way merge and handling 4000 records we get:

File 1	:	$R_{4001} \ldots . R_{5000}$
File 2	:	(empty)
File 3	:	$R_1 \ldots . R_{2000}$
File 4	:	$R_{2001} \ldots . R_{4000}$

Now we can do a three-way marge to get the sorted output on File 2. The total number of records handled during this merging process is 4000+5000 = 9000.

Again we can do the distribution and merging in another way using four files. Let after initial distribution the contents of the files be:

File 1	:	$R_1 \ldots R_{1000}$; $R_{3001} \ldots R_{4000}$
File 2	:	$R_{1001} \ldots R_{2000}$; $R_{4001} \ldots R_{5000}$
File 3	:	$R_{2001} \ldots R_{3000}$
File 4	:	(empty)

Doing a three-way merge and handling 3000 records we get:

File 1	:	$R_{3001} \ldots R_{4000}$
File 2	:	$R_{4001} \ldots R_{5000}$
File 3	:	(empty)
File 4	:	$R_1 \ldots R_{3000}$

Now we can do a three-way merge to get the sorted output on File 3. The total number of records handled in this case is $3000+5000 = 8000$.

Again, if we wish to do a five-way merging then it is possible to complete the sorting by handling only 5000 records. In this case we have to distribute the initial runs on five files and merge them on a sixth file.

These considerations indicate that the balanced merging is not the best merging technique and it is interesting to look for improved merging patterns.

Though in the above example, balanced five-way merging handles only 5000 records whereas balanced two-way merging handles 15,000 records, there is no guarantee that the five-way merging will require less time than two-way merging.

In the merging process, we have to allocate input and output buffers in the computer's high-speed internal memory to minimize the number of disk accesses. Larger buffer size implies smaller number of disk accesses, and hence smaller merging time. In five-way merging we have to allocate at least six buffers (five for input and one for output), whereas in two-way merging we need three buffers only. As the memory is in fixed size, buffers in five-way merging would be smaller than that of the two-way merging. As a result five-way merging requires more disk accesses than two-way merging. Again we have seen that five-way merging handles smaller number of records than that of the two-way merging. Therefore, it is very difficult to conclude from this information which merging scheme would require less time. There are more improved merging patterns and buffer allocation techniques, as well as the techniques to produce long initial runs, which we shall explore throughout this thesis.

1.4.1 Pseudo Code of Balanced Merging:

```
Initialize $F$ ⟦ $F$ is the total number of files⟧
$P \leftarrow \lceil F/2 \rceil$ or $P \leftarrow \lfloor F/2 \rfloor$
Initialize TM ⟦ TM is total memory⟧
OFN $\leftarrow$ 0 ⟦ OFN is output file no⟧
Calculate IBS ⟦ IBS is input buffer size⟧
Calculate OBS ⟦ OBS is output buffer size⟧
Calculate RBS ⟦ RBS is runs read buffer size⟧

Open runs file
Set buffer to runs file
Create and open $F$ merge files
```

```
    For $j = 1$ to $P$
            Set input buffer to merge file($P$)
            ⟦ For initial distribution⟧

    Distribute initial runs on file(1) to file($P$) as evenly as possible

    CONTROL ← 1 ⟦ If CONTROL = 1, left bank of files is the source of the
    merging; otherwise right bank⟧

Merge:
    If(CONTROL = 1)
            SFSN ← 1 ⟦ SFSN is the source file start no⟧
            SFEN ← $P$ ⟦ SFEN is the source file end no⟧
            TFSN ← $P + 1$ ⟦ TFSN is the target file start no⟧
            TFEN ← $F$ ⟦ TFEN is the target file end no⟧
            CONTROL ← 0
    else
            SFSN ← $P + 1$
            SFEN ← $F$
            TFSN ← 1
            TFEN ← $P$
            CONTROL ← 1

    For $i$ = SFSN to SFEN
            Set buffer to file($i$)
```

```
Repeat
        For OFN = TFSN to TFEN
                Set buffer to file(OFN)
                Merge one run from each file(SFSN) to file(SFEN) to
                file(OFN)
                If there is only one run on the file(OFN) and all source files
                reach their end
                        Print("Sorting complete and output is on file(OFN)");
                        Exit from the program
until end of all source files is reached
goto Merge ■
```

CHAPTER 2

POLYPHASE MERGE

2.1 Fibonacci Numbers:

The sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

in which each number is the sum of the preceding two, plays an important role in many algorithms of computer science. The numbers of this sequence are generally denoted by F_n and are defined as follows:

$$F_0 = 0, \quad F_1 = 1, \quad F_{n+2} = F_{n+1} + F_n; \quad n \geq 0.$$

Actually the sequence represented by this definition is the *2nd order Fibonacci numbers*.

Again the sequence

$$0, 0, 1, 1, 2, 4, 7, 13, 24, 44, \ldots,$$

is called *3rd order Fibonacci numbers*. Here the sequence starts with two 0's, and then a 1, and then each number is the sum of the preceding three numbers. In a similar way we can generate Fibonacci numbers of any order as we shall see in subsequent sections.

2.2 Polyphase Merge on Three Files:

Let us consider a balanced merge example using only three files F1, F2 and F3, where two files is on the left bank and rest on the right bank, *i.e.*, with $P = 2$ and $F = 3$. The merging steps would take the following form:

Step 1: Distribute initial runs alternately on files F1 and F2.

Step 2: Merge runs from F1 and F2 onto F3; then stop if F3 contains only one run.

Step 3: Copy the runs of F3 alternately onto F1 and F2, then return to Step 2.■

If the initial distribution pass produces S runs, the first merge pass will produce $\lceil S/2 \rceil$ runs on F3, the second will produce $\lceil S/4 \rceil$, etc. Thus if, say, $17 \le S \le 32$, we will have 1 distribution pass, 5 merge passes and 4 copy passes. In general, if $S > 1$, the number of passes over all the data is $2\lceil \log_2 S \rceil$.

The copying passes in this procedure are undesirable, since they do not reduce the number of runs. Half of the copying can be avoided if we use a two-phase procedure:

Step 1: Distribute initial runs alternately on files F1 and F2.

Step 2: Merge runs from F1 and F2 onto F3; then stop if F3 contains only one run.

Step 3: Copy half of the runs from F3 onto F1.

Step 4: Merge runs from F1 and F3 onto F2; then stop if F2 contains only one run.

Step 5: Copy half of the runs from F2 onto F1. Return to Step 1.■

The number of passes over the data has been reduced to $\frac{3}{2}\lceil \log_2 S \rceil + \frac{1}{2}$, since Step 3 and Step 5 do only "half a pass"; about 25 percent of the time has, therefore, been saved.

The copying can actually be eliminated entirely, if we start with F_n runs on F1 and F_{n-1} runs on F2, where F_n and F_{n-1} are consecutive 2nd order Fibonacci numbers. Let us consider, for example, the case $n = 7$, $S = F_n + F_{n-1} = 13 + 8 = 21$:

	Contents of F1	**Contents of F2**	**Contents of F3**	**Remarks**
Phase 1:	1,1,1,1,1,1,1,1,1,1,1,1,1	1,1,1,1,1,1,1,1	—	Initial distribution
Phase 2:	1,1,1,1,1	—	2,2,2,2,2,2,2,2	Merge 8 runs to F3
Phase 3:	—	3,3,3,3,3	2,2,2	Merge 5 runs to F2
Phase 4:	5,5,5	3,3	—	Merge 3 runs to F1
Phase 5:	5	—	8,8	Merge 2 runs to F3
Phase 6:	—	13	8	Merge 1 run to F2
Phase 7:	21	—	—	Merge 1 run to F1

Here, for example, "3,3,3,3,3" denotes five runs of relative length 3, considering each initial run to be of relative length 1. Second order Fibonacci numbers are present everywhere in this chart.

Only phases 1 and 7 are complete passes over the data; phase 2 processes only 16/21 of the initial runs, phase 3 only 15/21, etc., and so the total number of "passes" comes to $(21+16+15+15+16+13+21)/21 = 5\frac{4}{7}$ if we assume that the initial runs have approximately equal length. This three-file case of perfect distribution of runs had been discovered earlier by Betz [3]. But if we use balanced merging using three files would require 10 passes over the data for these 21 initial runs. Again, the two-phase procedure described would require 8 passes to sort these 21 initial runs.

2.3 Generalized Polyphase Merge:

The idea of polyphase merge can be generalized to F files, for any $F \geq 3$, using $(F - 1)$-way merging. The generalized pattern involves generalized Fibonacci numbers. Let us consider the following seven-file example by distributing 321 initial runs:

	F1	F2	F3	F4	F5	F6	F7	Initial runs processed
Phase 1:	1^{63}	1^{62}	1^{60}	1^{56}	1^{48}	1^{32}	—	321
Phase 2:	1^{31}	1^{30}	1^{28}	1^{24}	1^{16}	—	6^{32}	$6 \times 32 = 192$
Phase 3:	1^{15}	1^{14}	1^{12}	1^{8}	—	11^{16}	6^{16}	$11 \times 16 = 176$
Phase 4:	1^{7}	1^{6}	1^{4}	—	21^{8}	11^{8}	6^{8}	$21 \times 8 = 168$
Phase 5:	1^{3}	1^{2}	—	41^{4}	21^{4}	11^{4}	6^{4}	$41 \times 4 = 164$
Phase 6:	1^{1}	—	81^{2}	41^{2}	21^{2}	11^{2}	6^{2}	$81 \times 2 = 162$
Phase 7:	—	161^{1}	81^{1}	41^{1}	21^{1}	11^{1}	6^{1}	$161 \times 1 = 161$
Phase 8:	321^{1}	—	—	—	—	—	—	$321 \times 1 = 321$

Here phase 1 is for initial distribution and 1^{63} represents 63 runs of relative length 1, etc.; six-way merges have been used throughout. This general pattern was developed by Gilstad [9], who called it the polyphase merge.

In order to make polyphase merging work as in the above examples, we need to have a "perfect Fibonacci distribution" of runs on the files after each phase.

2.4 Rules for Perfect Fibonacci Distribution:

When $F = 6$, we have the following perfect distribution of runs, from where we can get the idea about perfect Fibonacci distribution:

Level	F1	F2	F3	F4	F5	F6	Total	Final output will be on	
0	1	0	0	0	0	0	1	F1	
1	1	1	1	1	1	1	6	F7	
2	2	2	2	2	2	1	11	F6	
3	4	4	4	4	3	2	21	F5	
4	8	8	8	7	6	4	41	F4	
5	16	16	15	14	12	8	81	F3	
6	32	31	30	28	24	16	161	F2	
7	63	62	60	56	48	32	321	F1	
8	125	123	119	111	95	63	636	F7	
9	248	244	236	220	188	125	1261	F6	
....	....	....	....	....	....	....	....	.....	
n	a_n	b_n	c_n	d_n	e_n	f_n	t_n	F(k)	
$n+1$	a_n+b_n	a_n+c_n	a_n+d_n	a_n+e_n	a_n+f_n	a_n	t_n+5a_n	F($k-1$)	(1)
....	....	....	....	....	....	....	....	.....	

The file F7 will always be empty after initial distribution. The rule for going from level n to level $n+1$ shows that the condition

$$a_n \geq b_n \geq c_n \geq d_n \geq e_n \geq f_n \qquad (2)$$

will hold in every level.

In fact, it can be easily shown from the perfect distribution (1) that

$$\begin{aligned}
&f_n = a_{n-1},\\
&e_n = a_{n-1} + f_{n-1} = a_{n-1} + a_{n-2},\\
&d_n = a_{n-1} + e_{n-1} = a_{n-1} + a_{n-2} + a_{n-3},\\
&c_n = a_{n-1} + d_{n-1} = a_{n-1} + a_{n-2} + a_{n-3} + a_{n-4},\\
&b_n = a_{n-1} + c_{n-1} = a_{n-1} + a_{n-2} + a_{n-3} + a_{n-4} + a_{n-5},\\
&a_n = a_{n-1} + b_{n-1} = a_{n-1} + a_{n-2} + a_{n-3} + a_{n-4} + a_{n-5} + a_{n-6},
\end{aligned} \tag{3}$$

where $a_0 = 1$ and where we let $a_n = 0$ for $n = -1, -2, -3, -4, -5$.

The *pth order Fibonacci numbers* $F_n^{(p)}$ are defined by the rules

$$\begin{aligned}
&F_n^{(p)} = F_{n-1}^{(p)} + F_{n-2}^{(p)} + \cdots + F_{n-p}^{(p)}, \qquad \text{for} \qquad n \geq p;\\
&F_n^{(p)} = 0 \qquad \text{for} \qquad 0 \leq n \leq p-2;\\
&F_{p-1}^{(p)} = 1.
\end{aligned}$$

In other words, we start with p - 1 0's, then a 1, and then each number is the sum of the preceding p values. When $p = 2$, this is the usual Fibonacci sequence; for large values of p the sequence was first studied by Schlegel [22].

Equation (3) shows that the number of runs on F1 during a seven-file polyphase merge is a sixth-order Fibonacci number: $a_n = F_{n+5}^{(6)}$

In general, if we set $P = F - 1$, the polyphase merge distributions for F files will correspond to Pth order Fibonacci numbers in the same way. The kth file gets

$$F^{(P)}_{n+P-2} + F^{(P)}_{n+P-3} + \cdot\cdot\cdot\cdot + F^{(P)}_{n+k-2}$$

initial runs in the perfect nth level distribution, for $1 \leq k \leq P$, and the total number of initial runs on all files is therefore

$$t_n = PF^{(P)}_{n+P-2} + (P-1)F^{(P)}_{n+P-3} + \cdot\cdot\cdot\cdot + F^{(P)}_{n-1}.$$

2.5 Polyphase Merge Sorting Algorithm:

This algorithm takes initial runs and disperses them to files one run at a time, until the supply of initial runs is exhausted. Then it specifies how the files are to be merged, assuming that there are $F = P + 1 \geq 3$ files, using P-way merging. File F may be used to hold the input, since it does not receive any initial runs. In this algorithm we would deal with "logical file numbers" whose assignment to physical file varies as the algorithm proceeds. The following tables are maintained:

A[j], $1 \leq j \leq F$	:	The perfect Fibonacci distribution we are trying for.
D[j], $1 \leq j \leq F$	:	Number of dummy runs assumed to be present at the beginning of logical file number j.
FILE[j], $1 \leq j \leq F$	:	Number of physical file corresponding to logical file number j.

2.5.1 Pseudo Code of the Algorithm:

```
Step 1:  ⟦ Initialize ⟧
         for $j = 1$ to $F$ do
                 $A[j] \leftarrow 1$; $D[j] \leftarrow 1$; FILE$[j] \leftarrow j$
         $A[F] \leftarrow 0$; $D[F] \leftarrow 0$; FILE$[F] \leftarrow F$
         $l \leftarrow 1$; $j \leftarrow 1$

Step 2:  ⟦ Input to file $j$ ⟧
         Write one run on file number $j$
         $D[j] \leftarrow D[j] - 1$
         If end of the input file is reached
                 rewind all files
                 goto Step 5

Step 3:  ⟦ Advance $j$ ⟧
         If $D[j] < D[j + 1]$
                 $j \leftarrow j + 1$; goto Step 2
         else if $D[j] = 0$
                 goto Step 4
         else
                 $j \leftarrow 1$; goto Step 2

Step 4:  ⟦ Up a level ⟧
         $l \leftarrow l + 1$; $a \leftarrow A[1]$
         for $j = 1$ to $P$ do
                 $D[j] \leftarrow a + A[j + 1] - A[j]$
                 $A[j] \leftarrow a + A[j + 1]$
         $j \leftarrow 1$; goto Step 2
```

```
Step 5: ⟦ Merge ⟧
        If $l = 0$
            print("sorting is complete and the output is on FILE[1]");
            Exit from the program
        else
            ⟦ merge runs from FILE[1], . . ,FILE[$P$] onto FILE[$F$] until
            FILE[$P$] is empty. Merging process should operate as
            follows, for each run merged: ⟧
            If $D[j] > 0$ for all $j = 1$ to $P$
                $D[F] \leftarrow D[F] + 1$
                for $j = 1$ to $P$ do
                    $D[j] \leftarrow D[j] - 1$
            else
                merge one run from each FILE[$j$] for all $j = 1$ to $P$
                provided $D[j] = 0$
                for $j = 1$ to $P$ do
                    If $D[j] \neq 0$
                        $D[j] \leftarrow D[j] - 1$
                ⟦ Thus dummy runs are imagined to be at the
                beginning of the file, instead of at the ending ⟧

Step 6: ⟦ Down a level ⟧
        $l \leftarrow l - 1$
        rewind FILE[$P$] and FILE[$F$]
        FILE[1] $\leftarrow$ FILE[$F$]; $D[1] \leftarrow D[F]$
        for $j = 1$ to $P$ do
            FILE[$j + 1$] $\leftarrow$ FILE[$j$]
            $D[j + 1] \leftarrow D[j]$
        goto Step 5 ■
```

The distribution rule which is stated in Step 3 of this algorithm is intended to equalize the number of dummies on each file as well as possible. Fig. 2.1. illustrates the order of distribution when we go from level 4 (41 runs) to level 5 (81 runs) in a seven-file polyphase merging. If for example, there were only 61 initial runs, all runs numbered 62 and higher would be treated as dummies. The runs are actually being written at the end of the file, but it is best to imagine them being written at the beginning, since dummies are assumed to be at the beginning.

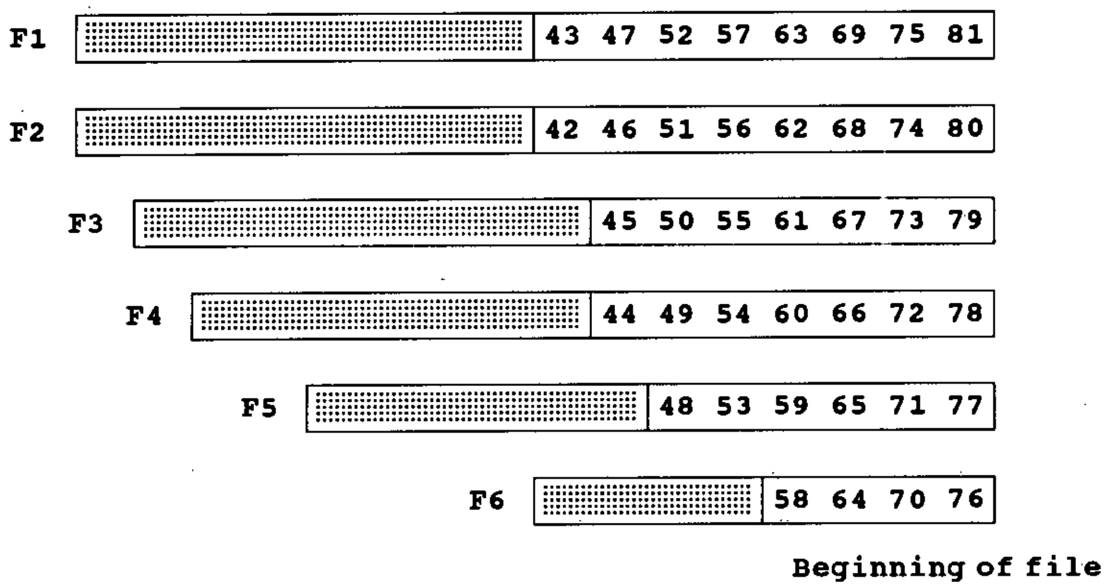


Fig. 2.1 The order in which runs 42 through 81 are distributed on files, when advancing from level 4 to level 5 of a seven-file case of polyphase merge. Shaded areas represent the first 41 runs which were distributed when level 4 was reached.

CHAPTER 3

CASCADE MERGE

3.1 Development of Cascade Merge:

Another basic pattern, called the "Cascade merge," was actually discovered before polyphase. A cascade merge, like polyphase, starts with a "perfect distribution" of initial runs on files, although the rule for perfect distribution is somewhat different from those of polyphase merge. The general pattern of cascade merge was developed by Betz and Carter [4].

Let us consider an example of cascade merge with five files and 707 initial runs. The file contents after initial distribution and different phases of merge are as follows:

	F1	**F2**	**F3**	**F4**	**F5**
Initially:	1^{246}	1^{216}	1^{160}	1^{85}	—
Phase 1:	1^{161}	1^{131}	1^{75}	—	4^{85}
Phase 2:	1^{86}	1^{56}	—	3^{75}	4^{85}
Phase 3:	1^{30}	—	2^{56}	3^{75}	4^{85}
Phase 4:	—	1^{30}	2^{56}	3^{75}	4^{85}

Here 1^{246} stands for 246 runs of relative length 1, etc. Each line in the above table represents a partial pass over the data. Phase 1, for example, is obtained by doing a four-way merge from F1, F2, F3, F4 to F5, until F4 is empty (this puts 85 runs of relative length 4 on F5). Phase 2 is obtained by doing a three-way merge from F1, F2, F3 to F4. Similarly a two-way merge to F3, and finally a one-way merge (copying operation) from F1 to F2, to obtain subsequent phases.

Now if we do a four-way merge to F1, then three-way merge to F2, and so on, we will obtain the following file contents at different phases:

	F1	F2	F3	F4	F5
Initially:	—	1^{30}	2^{56}	3^{75}	4^{85}
Phase 1:	10^{30}	—	2^{26}	3^{45}	4^{55}
Phase 2:	10^{30}	9^{26}	—	3^{19}	4^{29}
Phase 3:	10^{30}	9^{26}	7^{19}	—	4^{10}
Phase 4:	10^{30}	9^{26}	7^{19}	4^{10}	—

To complete the merging, we have to repeat the same process, until a single run is produced. We can show the whole process in a more compact form in the following way:

	F1	F2	F3	F4	F5	Initial runs processed
Phase 1:	1^{246}	1^{216}	1^{160}	1^{85}	—	707
Phase 2:	—	1^{30}*	2^{56}	3^{75}	4^{85}	707
Phase 3:	10^{30}	9^{26}	7^{19}	4^{10}*	—	707
Phase 4:	—	10^{4}*	19^{7}	26^{9}	30^{10}	707
Phase 5:	85^{4}	75^{3}	56^{2}	30^{1}*	—	707
Phase 6:	—	85^{1}*	160^{1}	216^{1}	216^{1}	707
Phase 7:	707^{1}	—	—	—	—	707

It is clear that the copying operations are unnecessary, and they could be omitted. Actually, however, in five-file case this copying takes only a small percentage of the total time. The items marked with an asterisk in the above table are those which were simply copied. Only 225 of the 4949 (=707×7) runs processed are of this type. Most of the time is devoted to four-way and three-way merging.

It is not hard to derive the "perfect distribution" for a cascade merge, with five files they are:

Level	**F1**	**F2**	**F3**	**F4**	**Total**
0	1	0	0	0	0
1	1	1	1	1	4
2	4	3	2	1	10
3	10	9	7	4	30
4	30	26	19	10	85
5	85	75	56	30	246
6	246	216	160	85	707
7	707	622	462	246	2037
....	....	....	....	....	....
n	a_n	b_n	c_n	d_n	$a_n+b_n+c_n+d_n$
$n+1$	$a_n+b_n+c_n+d_n$	$a_n+b_n+c_n$	a_n+b_n	a_n	$4a_n+3b_n+2c_n+d_n$
....	....	....	....	....	....

3.2 Initial Distribution of Runs:

When the actual number of initial runs is not perfect, we can insert dummy runs as usual. A superficial analysis of this situation indicates that the method of dummy run assignment is immaterial, since cascade merging operates by complete passes.

If we have 707 initial runs, each record is processed seven times as in the above example, but if there are 708 we must apparently go up a level so that every record is processed eight times. Fortunately, this abrupt change is not actually necessary. David E. Ferguson (see[18]) has found a way to distribute initial runs so that many of the operations during the first merge pass reduce to copying the contents of file. When such copying relations are by passed by simply changing "logical" file numbers relative to the "physical" file numbers, we obtain a relatively smooth transition from level to level.

Ferguson's method of distributing runs to files can be illustrated by considering the process of going from level 4 to level 5 in above perfect distribution. Assume that "logical" files (F1, . . ,F4) contain respectively (10, 19, 26, 30) runs, and that we want eventually to bring this up to (85, 75, 56, 30). The procedure may be summarized as follows:

	Add to F1	Add to F2	Add to F3	Add to F4
Step (1, 1)	19	0	0	0
Step (2, 2)	7	26	0	0
Step (2, 1)	19	0	0	0
Step (3, 3)	4	4	30	0
Step (3, 2)	7	26	0	0
Step (3, 1)	19	0	0	0

Here we first put 19 runs on F1, then (7, 26) on F1 and F2, etc.

3.3 Cascade Merge Sorting Algorithm:

The algorithm takes initial runs and disperses them to files, one run at a time, until the supply of initial runs is exhausted. Then it specifies how the files are to be merged, assuming that there are $F \geq 3$ files, using at most $(F - 1)$-way merging and avoiding unnecessary one-way merging. The following table are maintained:

A[j], $1 \leq j \leq F$	: The perfect cascade distribution we have most recently reached.
AA[j], $1 \leq j \leq F$	: The perfect cascade distribution we are striving for.
D[j], $1 \leq j \leq F$	: Number of dummy runs assumed to be present on logical file number j.
M[j], $1 \leq j \leq F$	: Maximum number of dummy runs desired on logical file number j.
FILE[j], $1 \leq j \leq F$	: Number of the physical file corresponding to logical file number j.

3.3.1 Pseudo Code of the Algorithm:

```
Step 1:  ⟦ Initialize⟧
         for $k$ = 2 to $F$ do
                 A[$k$] $\leftarrow$ 0; AA[$k$] $\leftarrow$ 0; D[$k$] $\leftarrow$ 0
         A[1] $\leftarrow$ 0; AA[1] $\leftarrow$ 1; D[1] $\leftarrow$ 1
         for $k$ = 1 to $F$ do
                 FILE[$k$] $\leftarrow$ $k$
         $i \leftarrow F - 2$; $j \leftarrow 1$; $k \leftarrow 1$
         $l \leftarrow 0$; $m \leftarrow 1$; goto Step 5
```

```
        ⟦This maneuvering is one way to get everything started, by
        jumping right onto the inner loop with appropriate settings of the
        control variables⟧

Step 2: ⟦Begin new level⟧
        ⟦We have just reached a perfect distribution, and since there is
        more input we must get ready for the next level⟧
        $l \leftarrow l + 1$
        for $k = 1$ to $F$ do
            A[$k$] ← AA[$k$]
        for $k = 1$ to $F - 1$ do
            AA[$F - k$] ← AA[$F - k + 1$] + A[$k$]
        for $k = 1$ to $F - 1$ do
            temp[$k$] ← FILE[$k$]
        for $k = 1$ to $F - 1$ do
            FILE[$F - k$] ← temp[$k$]
        for $k = 1$ to $F$ do
            D[$k$] ← AA[$k + 1$]
        $i \leftarrow 1$

Step 3: ⟦Begin $i$th sub-level⟧
        $j \leftarrow i$
        ⟦The variable $i$ and $j$ represents "Step ($i, j$)" in the example shown
        previously to go from level 4 to level 5 of perfect distribution⟧

Step 4: ⟦Begin Step($i, j$)⟧
        $k \leftarrow j$; $m \leftarrow$ A[$F - j - 1$]
        If $m = 0$ and $i = j$
```

```
                $i \leftarrow F - 2$; goto Step 3
          If $m = 0$ and $i \neq j$
                goto Step 2
          ⟦ Variable $m$ represents the number of runs to be written onto
          FILE[$k$]; $m = 0$ occurs only when $l = 1$⟧

Step 5:   ⟦ Input to FILE[$k$]⟧
          If $m \leq 0$
                goto Step 6
          Write one run on FILE[$k$]
          D[$k$] $\leftarrow$ D[$k$] - 1
          If input finished
                rewind all the files
                goto Step 7

Step 6:   ⟦ Advance⟧
          $m \leftarrow m - 1$
          If $m > 0$
                goto Step 5
          else
                $k \leftarrow k - 1$
          If $k > 0$
                $m \leftarrow$ A[$F - j - 1$] - A[$F - j$]
                goto Step 5
          else
                $j \leftarrow j - 1$
          If $j > 0$
                goto Step 4
```

```
else
        $i \leftarrow i + 1$
If $i < F - 1$
        goto Step 3
else
        goto Step 2

Step 7: ⟦ Prepare to merge⟧
        ⟦ At this point the initial distribution is complete, and the A, AA,
        D, and FILE tables describes the present states of the files⟧
        for $k = 1$ to $F - 1$ do
                M[$k$] $\leftarrow$ AA[$k + 1$]
        FIRST $\leftarrow$ 1
        ⟦ Variable FIRST is non-zero only during the first merge pass⟧

Step 8: ⟦ Cascade⟧
        If $l = 0$
                print("sorting is complete and the output is on FILE[1]");
                Exit from the program
        else
                for $p = F - 1$ to 1 by -1 do
                        a $p$-way merge from FILE[1], . . . ,FILE[$p$] to
                        FILE[$p + 1$] in the following way:
                        If $p = 1$
                                ⟦ simulation of a one-way merge by simple
                                interchange⟧
                                rewind FILE[2]
                                FILE[1] $\leftrightarrow$ FILE[2]
        else if FIRST = 1 and D[$p$ - 1] = M[$p$ - 1]
            ⟦simulation of a $p$-way merge by simple
            interchange⟧
            FILE[$p$] ↔ FILE[$p$ + 1]
            for $k$ = 1 to $p$ - 1 do
                D[$k$] ← D[$k$] - M[$p$ - 1]
                M[$k$] ← M[$k$] - M[$p$ - 1]
        else
            for $k$ = 1 to $p$ - 1 do
                M[$k$] ← M[$k$] - M[$p$ - 1]
            for $j$ = 1 to $p$ do
                if D[$j$] ≤ M[$j$]
                    merge one run from each FILE[$j$] and
                    put the output on FILE[$p$+1]
            for $j$ = 1 to $p$ do
                if D[$j$] > M[$j$]
                    D[$j$] ← D[$j$] - 1
        continue merging in this way until FILE[$p$] is empty

Step 9: ⟦Down a level⟧
    $l$ ← $l$ - 1
    FIRST ← 0
    for $k$ = 1 to $F$ do
        temp[$k$] ← FILE[$k$]
    for $k$ = 1 to $F$ do
        FILE[$F$ - $k$ + 1] ← temp[$k$]
    ⟦At this point all D's and M's are zero and will remain so⟧
    goto Step 8 ■
```

CHAPTER 4

REPLACEMENT SELECTION

4.1 Tree Selection Sorting:

This is an important internal sorting technique, based on the idea of repeated selection. The principle of tree selection sorting are easy to understand in terms of matches in a typical "knock-out tournament." Consider, for example, the result of the ping-pong contest shown in Fig. 4.1; where Q, R, S, T, U, V, W and X are the names of the eight players. Here Q beats R and S beats T, then in the next round S beats Q, etc.

Fig. 4.1 shows that S is the champion of the eight players, and 8 - 1 = 7 matches (*i.e.*, comparisons) were required to determine this fact. U is not necessarily be the second-best player; any of the people defeated by S, including the first-round loser T, might possibly be the second best. We can determine the second-best player by having T play Q, and the winner of that match plays U; only two additional matches are required to find the second-best player, because of the structure we have remembered from the earlier games.

In general, we can "output" the player at the root of the tree, and replace him/her by an extremely weak player. Substituting this weak player means that the original second-best player will now be the best, so he/she will appear at the root if we

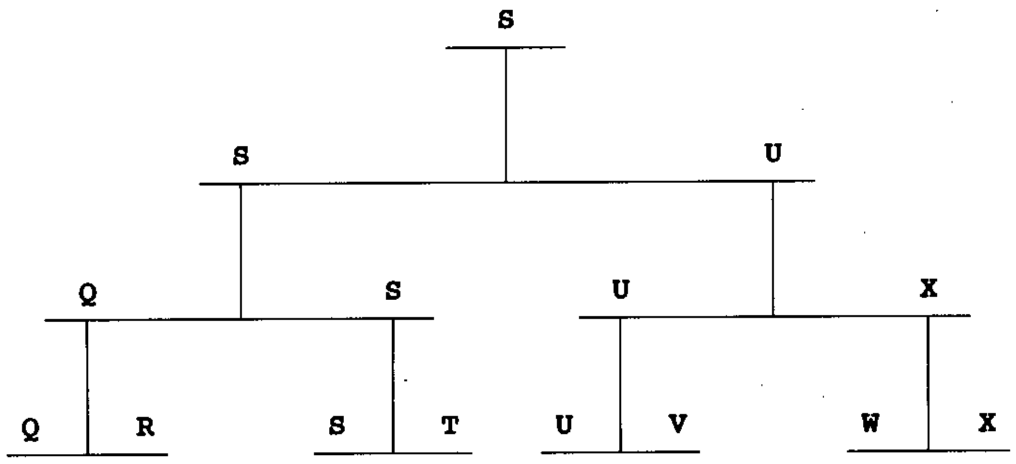


Fig. 4.1 A ping-pong tournament in tree form.

recalculate the winner in the upper levels of the tree. For this purpose only one path in the tree must be changed. If N is the number of players in the tournament, less than $\lceil \log_2 N \rceil$ further comparisons are needed to select the next-best player.

4.2 Tree form of Multi-Way Merging:

Let us assume that we have been given P ascending runs, *i.e.*, sequences of records whose keys are in non-decreasing order. The obvious way to merge them is to look at the first record of each run and to select the record whose key is the smallest. This record is transferred to the output and removed from the input, and the process is repeated. At any given time we need to look at only P keys (one from each input run) and select the smallest. If two or more keys are smallest, an arbitrary one is selected.

When P is not too large, it is convenient to make this selection by simply doing P - 1 comparisons to find the smallest of the current keys. But when P is, say, 8 or more, we can save work by using selection tree. For the case of selection tree only about $\log_2 P$ comparisons are needed each time, once the tree has been setup.

Consider, for example, the case of four-way merging, with a two-level selection tree:

$$\text{Step 1:}\qquad 237\left\{\begin{array}{l} 237\left\{\begin{array}{l}237\ \ 653\ \ \infty\\ 320\ \ 987\ \ \infty\end{array}\right.\\ 304\left\{\begin{array}{l}304\ \ 576\ \ \infty\\ 762\ \ \infty\end{array}\right.\end{array}\right.$$

Step 2:

$$237\ 304 \begin{cases} 320 \begin{cases} 653\ \infty \\ 320\ 987\ \infty \end{cases} \\ 304 \begin{cases} 304\ 576\ \infty \\ 762\ \infty \end{cases} \end{cases}$$

Step 3:

$$237\ 304\ 320 \begin{cases} 320 \begin{cases} 653\ \infty \\ 320\ 987\ \infty \end{cases} \\ 576 \begin{cases} 576\ \infty \\ 762\ \infty \end{cases} \end{cases}$$

⋮

⋮

Step 8:

$$237\ 304\ 320\ 576\ 653\ 762\ 987\ \infty \begin{cases} \infty \begin{cases} \infty \\ \infty \end{cases} \\ \infty \begin{cases} \infty \\ \infty \end{cases} \end{cases}$$

An additional key "∞", which is assumed to be higher than any actual key, has been placed at the end of each run in this example, so that merging terminates gracefully. Since external merging generally deals with very long runs, the addition of records with ∞ keys does not add substantially to the length of the data or to the amount of work involved in merging and such extra records frequently serve as a useful way to delimit the runs on a file.

Each step after the first in the above process consists of replacing the smallest element by the succeeding element in its run, and changing the corresponding path in the selection tree. Thus the three positions of the tree which contain 237 in Step 1 are changed in Step 2; the three positions containing 304 in Step 2 are changed in Step 3, and so on. The process of replacing one key by another in the selection tree is called *replacement selection.* From one standpoint, we can look at this four-way merge as equivalent to three two-way merges performed concurrently.

4.3 A Tree of Losers:

Fig. 4.2 shows the complete binary tree with 8 external (square) nodes and 7 internal (circular) nodes; the external nodes have been filled with keys and the internal nodes have been filled with the "winners" if the tree is regarded as a tournament to select the smallest key. The smaller numbers above each node show the traditional way to allocate consecutive storage positions for complete binary tree.

When the smallest key, 07, is to be replaced by another key in the selection tree of Fig. 4.2, we will have to look at the keys 24, 17 and 44, and no other existing keys, in order to determine the new state of the selection tree. Considering the tree as a tournament, these three keys are the losers in the matches played by 07. This suggests that what we really ought to store in the internal nodes of the tree is the loser of each match, instead of the winners; then the information required for updating the tree is readily available.

Fig. 4.3 shows the same tree as Fig. 4.2, but with the losers represented instead of the winners. An extra node number has been appended at the top of the tree, to indicate the champion of the tournament. Actually, each key except the champion is a loser exactly once, so each key appears once in an external node and once in an internal node. In practice, the external nodes at the bottom of Fig. 4.3 will represent fairly long records stored in computer memory, and the internal nodes will represent pointers to those records.

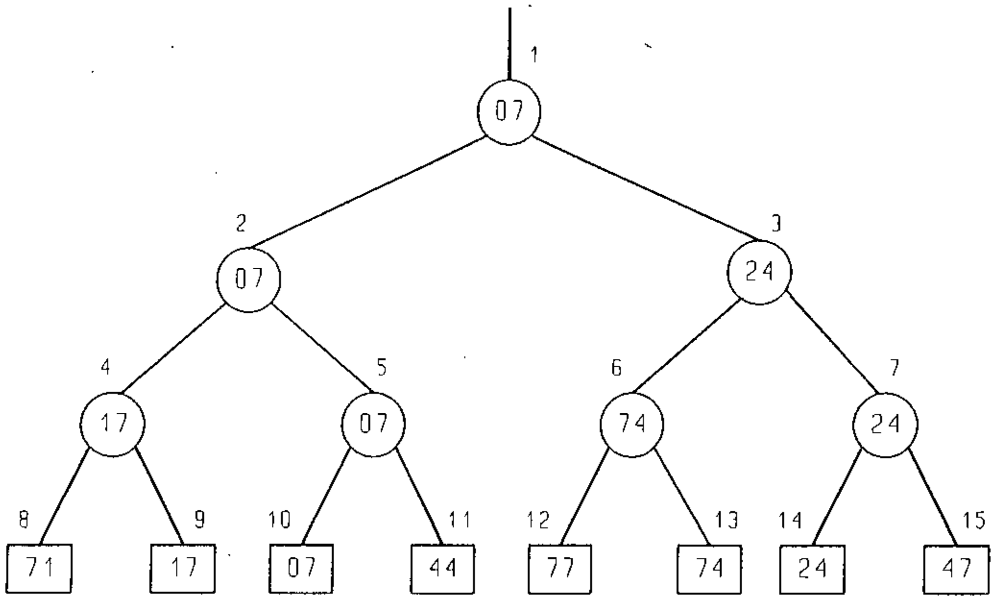


Fig. 4.2 Eight keys in binary tree form to select the smallest one.

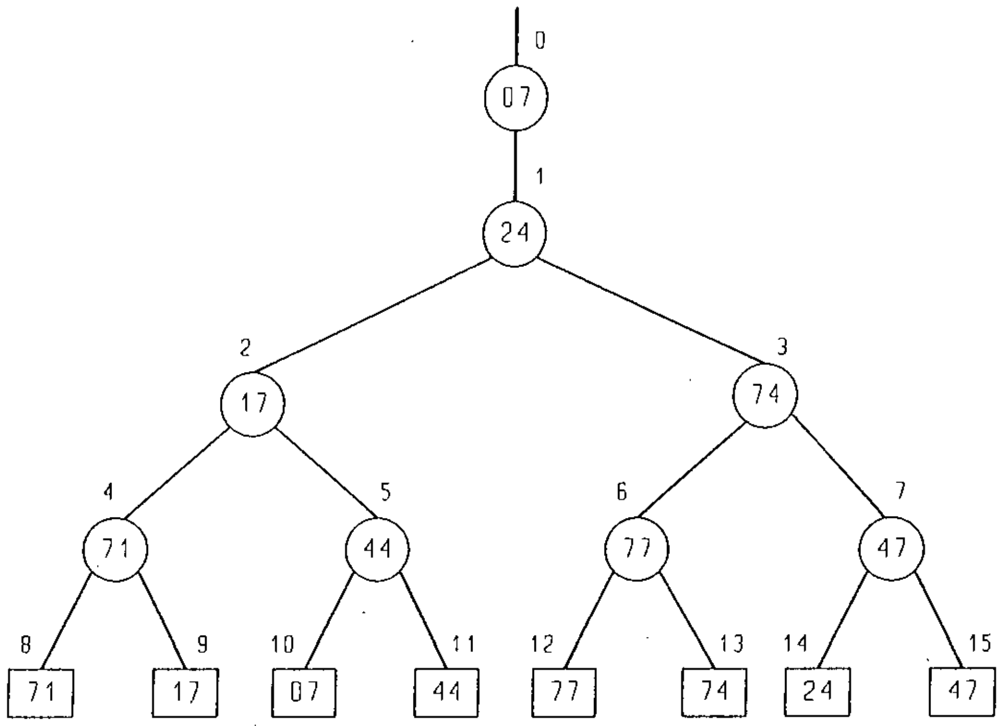


Fig. 4.3 Same keys as Fig. 4.2, but the losers are shown instead of the winners; the champion appears at the very top of the tree.

4.4 Initial Runs by Replacement Selection:

The technique of replacement selection can be used also in the first phase of external sorting, if we essentially do a *P*-way merge of the input data with itself. In this case we take *P* to be fairly large, so that the internal memory is essentially filled. When a record is output, it is replaced by the next record from the input. If the new record has a smaller key than the one just output, we cannot include it in the current run; but otherwise we can enter it into the selection tree in the usual way, and it will form part of the run currently being produced. Thus the runs can contain more than *P* records each, even though we never have more than *P* records in the selection tree at any time. Following table illustrates this process for $P = 5$; numbers in the parenthesis are waiting for inclusion in the following run.

EXAMPLE OF A FIVE-WAY REPLACEMENT SELECTION

In memory					At output
514	631	212	647	186	186
514	631	212	647	978	212
514	631	334	647	978	334
514	631	925	647	978	514
992	631	925	647	978	631
992	(626)	925	647	978	647
992	(626)	925	739	978	739
992	(626)	925	(046)	978	925
992	(626)	(582)	(046)	978	978
992	(626)	(582)	(046)	(845)	992
(767)	(626)	(582)	(046)	(845)	(end of run)
767	626	582	046	845	046
767	626	582	590	845	582
767	626	(312)	590	845	590

and so on.

This important method of forming initial runs was first described by Harold H. Seward (see [18]), who argued that the runs would contain more than $1.5P$ records when applied to random data. E. H. Friend (see [18]) remarked that "the expected length of the sequences produced eludes formulation but experiment suggests that $2P$ is a reasonable expectation."

A clever way to show that $2P$ is indeed the expected run length was discovered by E. F. Moore (see [18]), who compared the situation to a snowplow on a circular track. Consider the situation shown in Fig. 4.4; flakes of snow are falling uniformly on a circular road, and a lone snowplow is continually clearing the snow. Once the show has been plowed off the road, it disappears from the system. Points on the road may be designated by real numbers x, $0 \leq x < 1$; a flake of snow falling at position x represents an input record whose key is x, and the snowplow represents the output of replacement selection. The ground speed of the snowplow is inversely proportional to the height of snow it encounters, and the situation is perfectly balanced so that the total amount of snow on the road at all times is exactly P. A new run is formed in the output whenever the plow passes point 0.

After this system has been in operation for a while, it is immediately clear that it will approach a stable situation in which the snowplow runs at constant speed (because of the circular symmetry of the track). This means that the snow is at constant height when it meets the plow, and the height drops off linearly in front of the plow as shown in Fig. 4.5. It follows that the volume of snow removed in one revolution (*viz* the run length) is twice the amount present at any one time (*viz* P).

In many commercial applications the data is not completely random, it already has a certain amount of existing order. Therefore, the runs produced by replacement

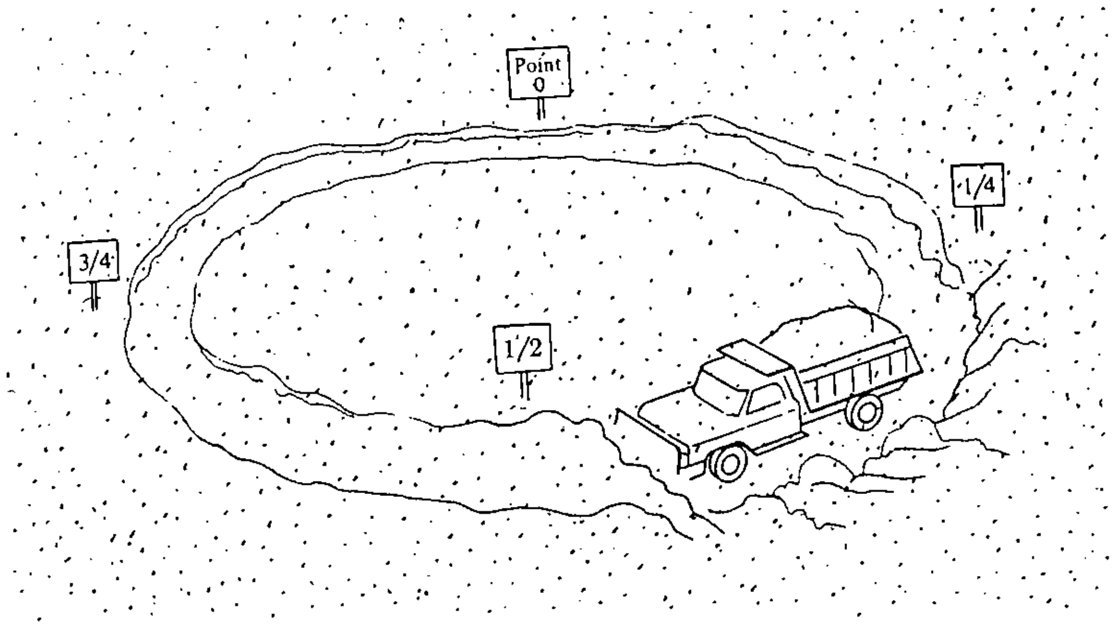


Fig. 4.4 A snowplow clearing the snow on a circular road.

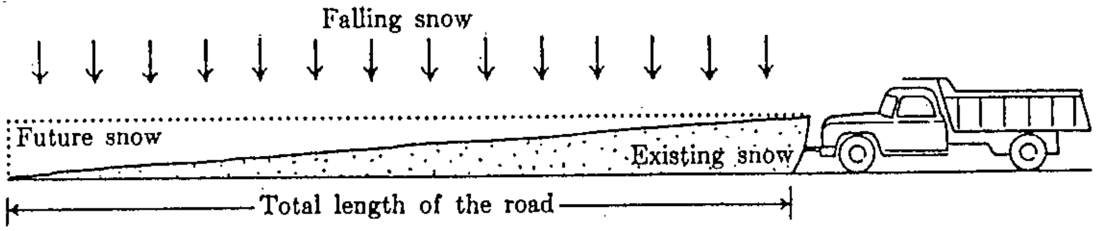


Fig. 4.5 Schematic diagram of steady state snow in front of the snowplow.

selection will tend to contain even more than $2P$ records. As the time required for external merge sorting is largely governed by the number of runs produced by the initial distribution phase, so that replacement selection becomes specially desirable; other types of internal sorting would produce about twice as many initial runs because of the limitations of memory size.

4.5 Replacement Selection Algorithm:

Let as now consider the process of creating initial runs by replacement selection in detail. This algorithm incorporates a nice way to initialize the selection tree. The principal idea is to consider each key as a pair (S, K), where K is the original key and S is the run number to which this record belongs. When such extended keys are lexicographically ordered, with S as major key and K as minor key, we obtain the output sequence produced by replacement selection.

This algorithm uses a data structure containing P nodes to represent the selection tree; the jth node X[j] is assumed to contain c words beginning in LOC(X[j]) = $L_0 + cj$, for $0 \leq j < P$, and it represents both internal node number j and external node number $P + j$ in Fig. 4.3. There are several named fields in each node:

KEY	=	the key stored in this external node;
RECORD	=	the record stored in this external node (including KEY as a sub-field);
LOSER	=	pointer to the "loser" stored in this internal node;
RN	=	run number of the record pointed to by LOSER;
FE	=	pointer to internal node above this external node in the tree;
FI	=	pointer to internal node above this internal node in the tree.

For example, when $P = 8$, internal node number 3 and external node number 11 of Fig. 4.3 would both be represented in X[3], by the fields KEY = 44, LOSER = $L_0 + 10c$ (the address of external node number 13), FE = $L_0 + 5c$, FI = $L_0 + 1c$.

The algorithm reads records sequentially from an input file and writes them sequentially onto an output file, producing RMAX runs whose length is P or more (except for the final run). There are $P \geq 2$ nodes, $X[0], \ldots, X[P-1]$ having fields as described above.

4.5.1 Pseudo Code of the Algorithm:

```
Step 1: ⟦ Initialize⟧
        RMAX ← 0; RC ← 0; RQ ← 0
        LASTKEY ← ∞; Q ← LOC(X[0])
        ⟦ RC is the number of the current run and LASTKEY is the key of
        the last record output. The initial setting of LASTKEY should be
        larger than any possible key.⟧
        For $j$ = 0 to $P$ do
                J ← LOC(X[$j$]); LOSER(J) ← J; RN(J) ← 0
                FE(J) ← LOC(X[$\lfloor (P + j)/2 \rfloor$])
                FI(J) ← LOC(X[$\lfloor j/2 \rfloor$])
        ⟦ The settings of LOSER(J) and RN(J) are artificial ways to get the
        tree initialized by considering a fictitious run number 0 which is
        never output.⟧
```

```
Step 2: ⟦ End of run? ⟧
        If RQ = RC
            goto Step 3.
        ⟦ Otherwise RQ = RC + 1 and we have just completed run
        number RC; any special actions required by merging pattern for
        subsequent passes of the sort would be done at this point. ⟧
        If RQ > RMAX
            print("initial run preparation complete")
            Exit from the program
        else RC ← RQ.

Step 3: ⟦ Output top of tree ⟧
        ⟦ Now Q points to the "champion" and RQ is its run number ⟧
        If RQ ≠ 0
            output RECORD(Q); LASTKEY ← KEY(Q)

Step 4: ⟦ Input new record ⟧
        If end of the input file is reached
            RQ ← RMAX + 1; goto Step 5
        else
            RECORD(Q) ← next record from the input file
        If KEY(Q) < LASTKEY ⟦ so that this new record does not belong
        to the current run ⟧
            RQ ← RQ + 1
            if RQ > RMAX
                RMAX ← RQ
```

```
Step 5:  ⟦ Prepare to update ⟧
         ⟦ Now Q points to a new record, whose run number is RQ ⟧
         T ← FE(Q)
         ⟦ T is a pointer variable which will move up the tree ⟧

Step 6:  ⟦ Set new loser ⟧
         If (RN(T) < RQ) or ( (RN(T)=RQ) and
                 (KEY(LOSER(T)) < KEY(Q)) )
                 LOSER(T) ↔ Q; RN(T) ↔ RQ
                 ⟦ variables Q and RQ keep track of the current winner and
                 its run number ⟧

Step 7:  ⟦ Move up ⟧
         If T = LOC(X[1])
                 goto Step 2
         else
                 T ← FI(T)
                 goto Step 6 ■
```

CHAPTER 5

PRACTICAL CONSIDERATIONS

5.1 External Storage Devices:

All the information that is processed by a computer should not reside in computer's costly high-speed internal or main memory. There are large volumes of data and computer usable information, that are commonly stored in computer's external or secondary memory as special entities called *files*.

Any location in main memory can be accessed very quickly. A typical access time of any location in main memory is less than one microsecond. Main memory provides for the immediate storage requirements of the central processor for the execution of programs, including users programs, assemblers, compilers and supervisory routines of the operating system.

The storage capacity of main memory is limited by two major factors — the cost of main memory and the technical problems in developing a large-capacity main memory. The storage requirements for programs and the data on which they operate exceed the capacity of main memory in virtually all computer systems. Therefore, it is necessary to extend the storage capabilities of a computer by using devices external to the main memory.

An *external storage device* may be loosely defined as a device other than main memory on which information or data can be stored and from which the information can be retrieved for processing at some subsequent point in time. External storage devices have a larger capacity and are less expensive per bit of information stored than main memory. The time required to access information, however, is much greater for these devices.

The most common external storage devices are magnetic tapes, drums and disk units. The first compact external storage medium to be widely used was magnetic tape. In early days external sorting was done on magnetic tapes. Though they were cheapest form of external bulk storage of data, they were not fast enough for on-line input output operations. Therefore, soon they were replaced by magnetic disk devices. Again, the maximum storage capacity for magnetic drum devices is limited and drums have been declining in popularity primarily due to the success of magnetic disk devices. Disk devices provide relatively low access time and high-speed data transfer.

5.2 Magnetic Disk Packs:

The magnetic disk packs consists of a number of metal platters which were stacked on top of each other on a spindle. The upper and lower surfaces of each platter are coated with ferromagnetic particles that provide an information storage medium. The surfaces of each platter are divided into concentric bands called *tracks*. Again, track is further subdivided into *sectors*, which are the addressable storage units.

Information is transferred to or from a disk through read/write heads. Each read/write head floats just above or below the surface of a disk, while the disk is rotating constantly at a high speed.

The storage area on the disk to or from which data can be transferred without movement of the read/write heads is termed as *cylinder* or *seek area*. Hence, cylinder is a set of vertically aligned tracks which reside on the platters of a disk.

Disk storage devices are the most versatile storage devices available. They can provide large capacity and relatively fast access time and satisfy the requirements of most computer systems.

5.3 Disk I/O Time:

The amount of time required to read or write a disk file is essentially the sum of three quantities:

Seek time: Disk storage can be viewed as consisting of consecutively numbered cylinders. A seek is a movement of the read/write head to locate the cylinder in which a particular track resides and the *seek time* is the time to move the access arm to the proper cylinder. The time for a seek is the most significant delay when accessing data on a disk.

Latency time: In disk units there is a rotational delay or latency in waiting for the disk surface to rotate to a sector, where a data transfer can commence. In other wards, *latency time* is the rotational delay until the read/write head reaches the right spot to read or write data.

Therefore, the *disk access time* associated with a particular I/O operation can be expressed as the sum of the seek time and the latency time.

Table 5.1 Characteristics of Magnetic Disk Packs:

System	IBM		DEC	Commodore
Model	3330	3380	RM80	D9090
Usable surfaces/unit	19	15	7	6
Tracks/surface	404	885	1,122	153
Sector size (chars)	variable	variable	512	256
Chars/track	13,030	47,476	15,360	8,192
Chars/unit (megabytes)	100	630	124	9.5
Avg. seek time (ms)	30	16	25	153
Avg. latency time (ms)	8.3	8.3	8.3	8.3
Transmission rate (chars/ms)	806	3,000	1,212	500

Transmission time: The *transmission time*, also called flow time, is the time to read or write the record or series of records (as dictated by the I/O operation), given that the heads are positioned over the disk location of the first record to be read or written. The transmission time depends directly on how fast the disk rotates.

Therefore, the *total time* to complete a disk operation is the sum of three components *i.e.* the seek time, the latency time and the transmission time.

The characteristics of a few representative magnetic disk units are summarized in Table 5.1.

5.4 Factors Affecting External Sorting Time:

External sorting time depends on many factors, such as seek time, latency time and transmission time of magnetic disks, and computer's main memory capacity, characteristics of the sorting algorithm, number of files used for merging or sorting operation, number of initial runs to be processed, number of passes required over the data, speed of the computer etc.

5.4.1 Effect of Disk I/O Time:

We know that external sorting is done in a piecemeal fashion and frequent disk access is necessary. But it is apparent from Table 5.1 that disk access time (sum of the seek time and the latency time) is the most significant part of the total time required for disk input/output. Calculation shows that one disk access time is almost equivalent to 30 ~ 50 kilobytes of information transmission time.

The influence of disk access time can be clearly explained with the help of an example. Let it is required to write 100 kilobytes of information to a disk file. For this purpose, we consider an IBM 3380 magnetic disk pack, the characteristics of which are shown in Table 5.1. We can write this 100 kilobytes of information in a single disk access or in multiple disk accesses. Thus, the total disk access time would be different, depending on the number of disk accesses required. But the total transmission time would be remain constant, irrespective of the number of disk accesses. For different writing strategies, the total time required to write this 100 kilobytes of information can be calculated as follows.

Total transmission time (which is always constant)

$$= 100 \times 1024 \times \frac{1}{3000} \text{ milliseconds}$$
$$= 34 \text{ milliseconds}$$

Total writing time (if 1 byte of information is written at once)

$$= 100 \times 1024(16+8.3)+34 \text{ milliseconds}$$
$$= 2488320+34 \text{ milliseconds}$$
$$= 41.5 \text{ minutes}$$

Total writing time (if 1 kilobyte of information is written at once)

$$= 100(16+8.3)+34 \text{ milliseconds}$$
$$= 2430+34 \text{ milliseconds}$$
$$= 2.464 \text{ seconds}$$

Total writing time (if 100 kilobytes of information are written at once)

$$= 16+8.3+34 \text{ milliseconds}$$
$$= 58.3 \text{ milliseconds}$$

In this example, 99.99, 98.62 and 41.68 percent of total writing time is spent for disk accesses respectively, for the three strategies of disk writing shown above. If we consider the case of reading in place of writing then a similar timing values would also appear. Thus disk access time is an important factor for disk input/output operations. Again, most of the external sorting time is spent for disk operations. Therefore, we can conclude that disk access time is one of the major factor which affect the external sorting time.

Data transmission rate of the magnetic disk have also influence on the external sorting time. Faster the data transmission rate, smaller would be the external sorting time.

5.4.2 Effect of Number of Initial Runs:

External sorting time also depends on the number of initial runs produced by internal sorting phase. We have seen previously that if *S* runs were produced during the internal sorting phase, and if $2^{k-1} < S \leq 2^k$, then balanced 2-way merge procedure makes exactly $k = \lceil \log_2 S \rceil$ merging passes over all the data. Also, if $m^{k-1} < S \leq m^k$, then *m*-way balanced merge procedure makes exactly $k = \lceil \log_m S \rceil$ merging passes over all the data. Again, as the number of merging passes increases, the amount of data handling would also increase. Similarly, for the case of polyphase and cascade merging also, the amount of data handling would also increase with the increase in the number of initial runs.

We can more easily express the fact with the help of an example. Suppose we want to sort a large data file. If 50 initial runs were produced from this data file by internal sorting phase, then using a 4-way balanced merge procedure, it would require three merging passes over all the data. But if 75 initial runs were produced by internal sorting phase then, the same 4-way balanced merge procedure would require four merging passes over all the data. Therefore, for the same data file, the number of passes over the data may vary due to the change in the number of initial runs produced. Thus, we can establish that the total amount of data to be handled by external sorting routine would increase with the increase in the number of initial runs produced by internal sorting phase, resulting an increase in the external sorting time.

5.4.3 Effect of Algorithms:

Characteristics of the algorithms have also influence on the external sorting time. Balanced merge algorithm makes complete merging passes over all the data, while polyphase and cascade merge algorithms make partial merging passes over the data. As balanced merge algorithm make always complete passes over the data, therefore, it is not always possible to get smooth transition for sorting time for consecutive number of runs. But, it is almost always possible for the case of polyphase and cascade merge algorithms, because these algorithms sometimes make partial merging passes over the data depending on the number of initial runs distributed.

Let us consider a 3-way balanced merge example to show that for the case of balanced merging, it is not always passible to get smooth transition for sorting time for consecutive number of runs. Suppose, we have 27 initial runs, which would require three merging passes over all the data, for this 3-way balanced merging. The number of runs handled during this merging process is 81 (=27×3). But, if we have 28 initial runs, then the number of merging passes over all the data would be four and the number of runs handled during this merging passes is 112 (=28×4). For the case of 29 initial runs, the number of runs handled is 116 (=29×4) and for the case of 26 initial runs, the number of runs handled is 78 (=26×3).

Summarizing, we can write that for the case of 26, 27, 28 and 29 initial runs, the number of runs handled by 3-way balanced merge routine is 78, 81, 112 and 116 respectively. Here it is observed that, as the number of initial runs changes from 27 (a power of 3) to 28 the number of runs handled have abruptly changed from 81 to 112. But for the other two cases (26 to 27 and 28 to 29) the changes in the number of runs handled is relatively small. But for the case of polyphase and cascade merge algorithms, the above type of abrupt change in the number of initial runs handling can be avoided because these algorithms use partial passes over the data.

Actually, for an m-way balanced merge algorithm, there would be an abrupt change in the number of initial runs handled by sorting routine, when the number of initial runs to be processed by the sorting routine cross the number m^x, where x is any positive integer.

Again, merging time is proportional to the number of initial runs handled by the sorting routine. Therefore, we can conclude that for polyphase and cascade merge algorithms the external sorting time changes smoothly for consecutive number of initial runs, but for balanced merge algorithm external sorting time do not change smoothly. This conclusion indicates that external sorting time also depends on the sorting algorithm.

5.4.4 Effect of Order of Merging:

External sorting time depends on the number of files used in merging operation. For an easy explanation of this fact, let us define a new term called order of merging. The *order of merging* is the number of data items that are compared and then merged in a merging operation.

To explain the dependence of external sorting time on the order of merging, let us consider a simple example. Suppose 100 initial runs are given and it is required to merge them. We can easily merge this runs by simply doing a 100-way merging. But to perform a 100-way merging, we have to allocate 101 buffers (100 for input and 1 for output) in computer's memory. As computer's allocable memory is always limited is size, therefore, larger the number of buffers, smaller would be the individual buffer size, results larger number of disk accesses, which increase the total external sorting time.

But if we perform a 2-way balanced merging, the only three buffers (two for input and one for output) are required and the buffers would be relatively many times larger than those of the 100-way merging. Therefore, the number of disk accesses would be reduced drastically and at the first glance, it may be natural to think that external sorting time would also be accordingly diminished.

But actually the fact is different. In 100-way merging, only one merging pass is required over all the data, while in 2-way balanced merging, seven complete passes over all the data is required. Therefore, in 100-way merging, smaller would be the number of initial runs handling but larger would be the number of disk accesses; while in balanced 2-way merging, larger would be the number of initial runs handling but smaller would be the number of disk accesses. And it is difficult to establish, which merging scheme would require smaller amount of time. But it may conclude that the order of merging is a factor which affect the external sorting time.

5.4.5 Effect of Computer's Speed:

The computational speed of the computer have also influence on the external sorting time. But practically, most of the external sorting time is spent for disk input and output operations and an insignificant percentage of total sorting time is spent for internal computation. Therefore, computational speed of the computer is not a seriously considerable factor for external sorting time estimation. But for a speedy computer, the sorting time would be diminished.

5.5 Techniques to Reduce External Sorting Time:

The major techniques to reduce external sorting time is to maintain an appropriate buffering strategy for read/write operations and to prepare longest initial runs by replacement selection algorithm. Also, it is essential to use a good sorting algorithm with an optimum order of merging to complete the external sorting within shortest possible time.

5.5.1 Buffering:

The main problem with external sorting is that it require frequent disk access, which cost more from the point of time requirement. Larger the number of disk accesses, more the sorting time. Therefore, it is necessary to reduce the number of disk accesses to diminish the external sorting time. To reduce the number of disk accesses or seek, we can allocate input and output buffers in the computer's high-speed internal memory. In this case, the information to be written to a disk file accumulated initially in the output buffer and when the buffer is full, it is physically written (flushed) to the disk file. Thus, for a large buffer, a sizeable amount of information can be written to the disk file, with the expense of only one seek. Again, in the case of read operation from a disk file, a full buffer of information is read at once and then they are used from the buffer when necessary. In this situation also, only one seek is required to read a full buffer of information.

Thus it is apparent from the above discussion that for the read/write operations, input/output buffers can be used to reduce the number of disk accesses. Larger the buffer size, smaller would be the number of disk accesses. But we cannot increase the buffer size as we wish, due to the limitations of total allocable memory for this purpose.

5.5.2 Unequal Buffering:

To reduce the number of disk accesses, we have mentioned about the input/output buffers. But up to this point, nothing is specified about their relative sizes. Actually, it is quite straightforward to allocate all the buffers of equal size. But practically, in external sorting, data from several source files are merged into a target file. Thus one output and several (one for each source file) input buffers are essential for efficient disk input/output operation. Careful examination shows that it is better to make the output buffer larger than input buffers to reduce the number of disk accesses. The following simple example can reveal the fact.

Let a small computer system for external sorting. The memory space available for buffer allocation for this system during the merging operation is equivalent to the 45 records of a database at once. At a particular step of the external sorting, source file F1 and F2 contain 1500 records each. It is required to merge these two files into the target file F3. Straightforwardly, we can equally divide the allocable memory into input/output buffers. The following table calculates the total number of disk accesses necessary for this situation.

Table 5.2 Equal Buffer Allocation

	File F1	File F2	File F3	Total
Number of records to be read from or written to the file	1,500	1,500	3,000	
Buffer size (in terms of the number of records)	15	15	15	
Number of disk accesses required	100	100	200	400

?

where, the number of disk accesses required

$$= \left\lceil \frac{\text{Number of records to be read from or written to the file}}{\text{Buffer size (in terms of number of records)}} \right\rceil$$

Again, if we properly make the output buffer larger than input buffers, then the total number of disk accesses may be reduced, which is shown in the following table.

Table 5.3 Efficient Buffer Allocation

	File F1	File F2	File F3	Total
Number of records to be read from or written to the file	1,500	1,500	3,000	
Buffer size (in terms of the number of records)	13	13	19	
Number of disk accesses required	116	116	158	390

But for the case of unequal buffers, if the buffer size is not properly set, then the number of disk accesses may be larger than those of the equal buffers. The following table illustrates the fact for the same example.

Table 5.4 Inefficient Buffer Allocation

	File F1	File F2	File F3	Total
Number of records to be read from or written to the file	1,500	1,500	3,000	
Buffer size (in terms of the number of records)	10	10	25	
Number of disk accesses required	150	150	120	420

Among the above three illustrations, Table 5.3 represents an efficient buffer allocation, where number of disk accesses are minimum. Therefore, to optimize the number of disk accesses, it is not sufficient to maintain the input and output buffers only, but it requires appropriate consideration to their relative sizes.

5.5.3 Long Initial Runs:

We have discussed in the previous section that there is another factor which has a profound influence on the total external sorting time. We know, sorting time increases with the increase in the number of initial runs produced by internal sorting phase. Therefore, by any means, if it is possible to reduce the number of initial runs formed from a given data file, then the total sorting time may shorten. The reduction of number of initial runs produced from a data file means individual runs would be longer in size. But the size of the initial runs produced by conventional internal sorting methods is limited to the maximum size of the computer's high-speed internal memory capacity. By adopting special techniques of internal sorting, called replacement selection algorithm, it is possible to produce initial runs larger than the computer's memory capacity, which have been discussed in detail in chapter four.

CHAPTER 6

EXPERIMENTS AND RESULTS

6.1 Design of Experiments:

This thesis is an endeavor to analyze the performance of external sorting techniques. Here our main purpose is to examine the time complexity of these techniques. Due to the unavailability of any standard formula for external sorting time calculation, we have taken an elaborate process of sorting a large data file under various sorting conditions by varying different external sorting parameters to find the time complexity.

As the first step of external sorting, replacement selection routine has been prepared to produce long initial runs larger than the computer's high-speed internal memory capacity. Then to merge these initial runs, merging routines have been prepared using balanced, polyphase and cascade merging algorithms.

In external sorting, data is read from several hard disk files and after processing it in the computer's memory, it is written back to the hard disk in a different file. Thus, necessary hard disk capacity for external sorting must be at least twice as that of the size of the data file to be sorted.

For the purpose of our experiment and by considering the availability of disk space for this work, we have chosen a normalized data file with 1.05 million records, where the size of each record is 80 bytes, *i.e.*, the size of our data file is 84 megabytes. Keys of these records — based on which the records are to be sorted, have been randomly generated using a random number generator. The remaining places of the records — which have no influence on the sorting process, except it remains with the records, have been filled with spaces.

6.1.1 Replacement Selection Routine:

We know, as the number of initial runs decreases the total external sorting time also decreases. Therefore, to produce longer and smaller number of initial runs, replacement selection routine has been prepared. The method of producing long initial runs larger than the computer's high-speed internal memory capacity has been discussed in detail in chapter four.

The size of the initial runs produced by replacement selection algorithm depends on the amount of memory allocated for the selection tree. We have adapted the replacement selection routine in such a way, so that during run time of the program, it can obtain the information about the available allocable memory size. From this total available memory, two 10 kilobyte buffers have been allocated, one for input and the other for output of disk file for this replacement selections routine. Also to avoid any run time hang up of the computer, another 10 kilobytes of memory have been reserved for program's run time use. Then the remaining available memory have been allocated for the selection tree.

After all the above consideration, 448 kilobytes of memory is allocable for the selection tree for our computer system. We know, the average length of the runs produced by replacement selection is twice as that of the size of the available memory for the selection tree. Therefore, the average size of the runs produced by replacement selection is 896 kilobytes, which is equal to the size of 11,200 records. In other words, on the average, each run contains 11,200 records. Since, the data file size is equivalent to 1.05 million records, the number of runs produced by replacement selection routine is 94.

During the development of this replacement selection routine, great care has been taken to allocate and free the far heap memory by using far heap management routines available in C. As far heap memory is used by the program, it is compiled in huge model and register variables are used throughout the program to speed up the internal computations.

6.1.2 Merging Routines:

To merge the long initial runs produced by replacement selection procedure, merging routines have been developed using balanced, polyphase and cascade merging algorithms. The theoretical aspects of these algorithms have been discussed in detail in chapter one, two and three respectively. Theoretically, the algorithms are quite different from each other and each one merge runs on its own way. We have taken all the measures to develop these external sorting routines, so that they can perfectly follow their own rules to merge the initial runs.

From chapter five we know that to reduce the total external sorting time it is necessary to minimize the number of disk accesses. Again number of disk accesses can be minimized by maintaining appropriate buffers for input and output disk files. Also it has been shown that unequal buffers can reduce the number of disk accesses, making it minimum at certain value of output to input buffer ratio (BR) > 1.

The merging routines in this thesis have been developed in such a way, so that during run time of the program they can get information about the available allocable memory of the computer system. From this available memory 10 kilobytes have been reserved for programs run time use, as was done in case of replacement selection, to avoid any run time hang up of the computer. The remaining memory have been allocated for buffers of input and output disk files. We have maintained output buffer larger than input buffers, by using *BR* to be equal to 4, 8, 10, 12, and 16.

To design the merging routines, far heap management procedures available in C have been properly used to manage all the available memory of the computer system. The programs have been compiled in huge model so that it can use 32 bit address to access the memory beyond its current data segment and register variables have been used to speed up the internal computations.

In order to compare performance of various external sorting algorithms, we have used total computational time as a summarized criterion.

6.2 Balanced Merge:

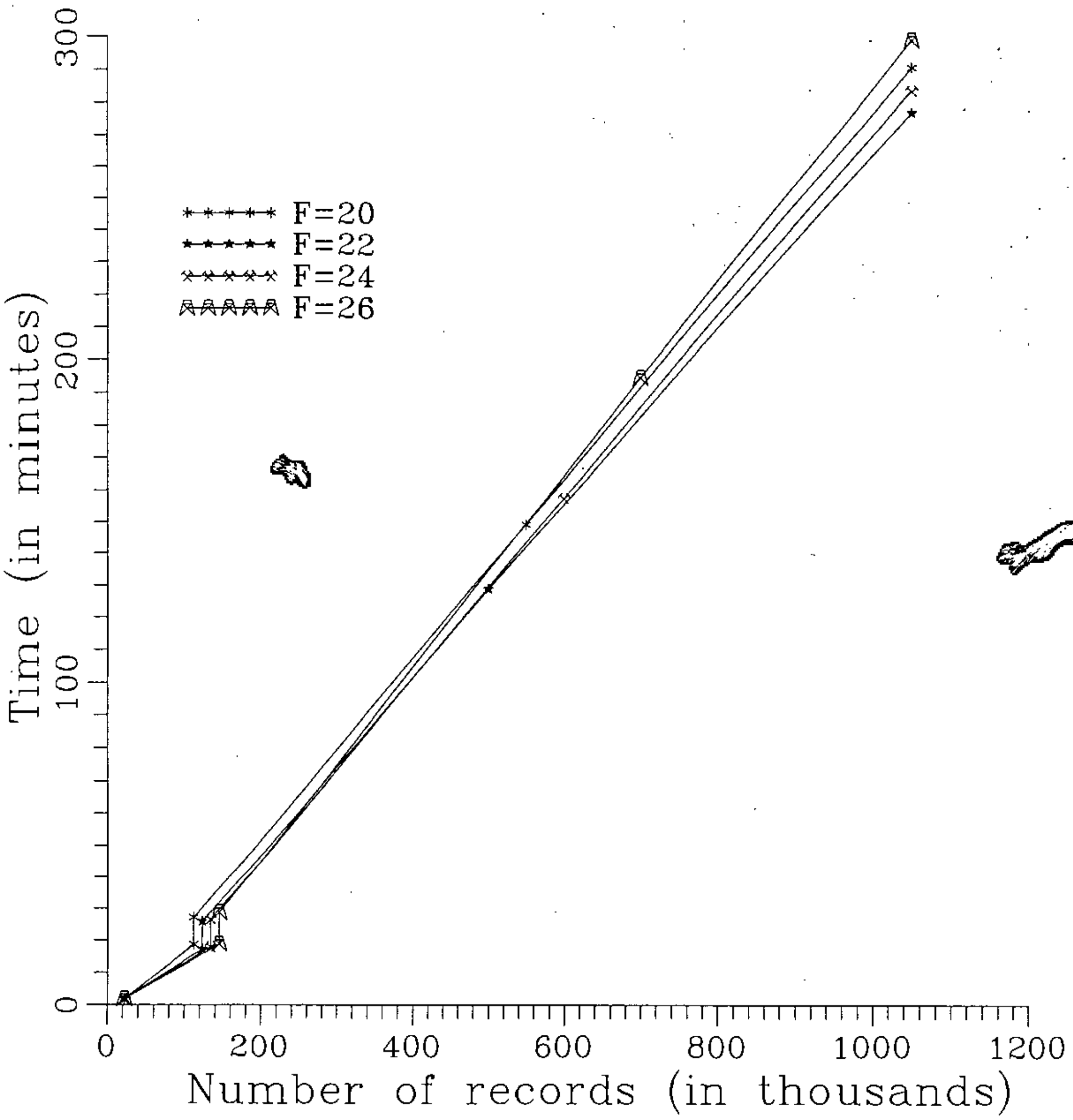
To find the performance of balanced merge techniques with the variation of different external sorting parameters, the balanced merge routine has been developed in C. We have taken *F* and *BR* as parameters for this sorting technique. Initially, considering smaller values of *F* (6, 10 etc.), we have sorted different amount of data. But this showed that the sorting time is too large. Then by several trials we have found that for $F=22$, the sorting time is minimum for large number ($>$400,000) of records. When value of *F* exceeds 22 the sorting time again increases. Therefore, we have decided to take the experimental data around $F=22$, namely for *F*=20, 22, 24 and 26. We have discarded any odd values of *F* because, the sorting time for these cases are similar to the corresponding even values.

For different values of *BR* (4, 8, 10, 12 and 16), with different values of *F*, experimental data have been gathered and it is observed from graphs 6.1-6.5 that for all values of *BR*, minimum sorting time occurs when *F*=22, for large number of records. Among these different minimum sorting times for different *BR* values, the minimum occurs when *BR* is 10. Therefore, for our experimental environment with large number of records minimum sorting time occurs when *BR*=10 and *F*=22 (see graph 6.3).

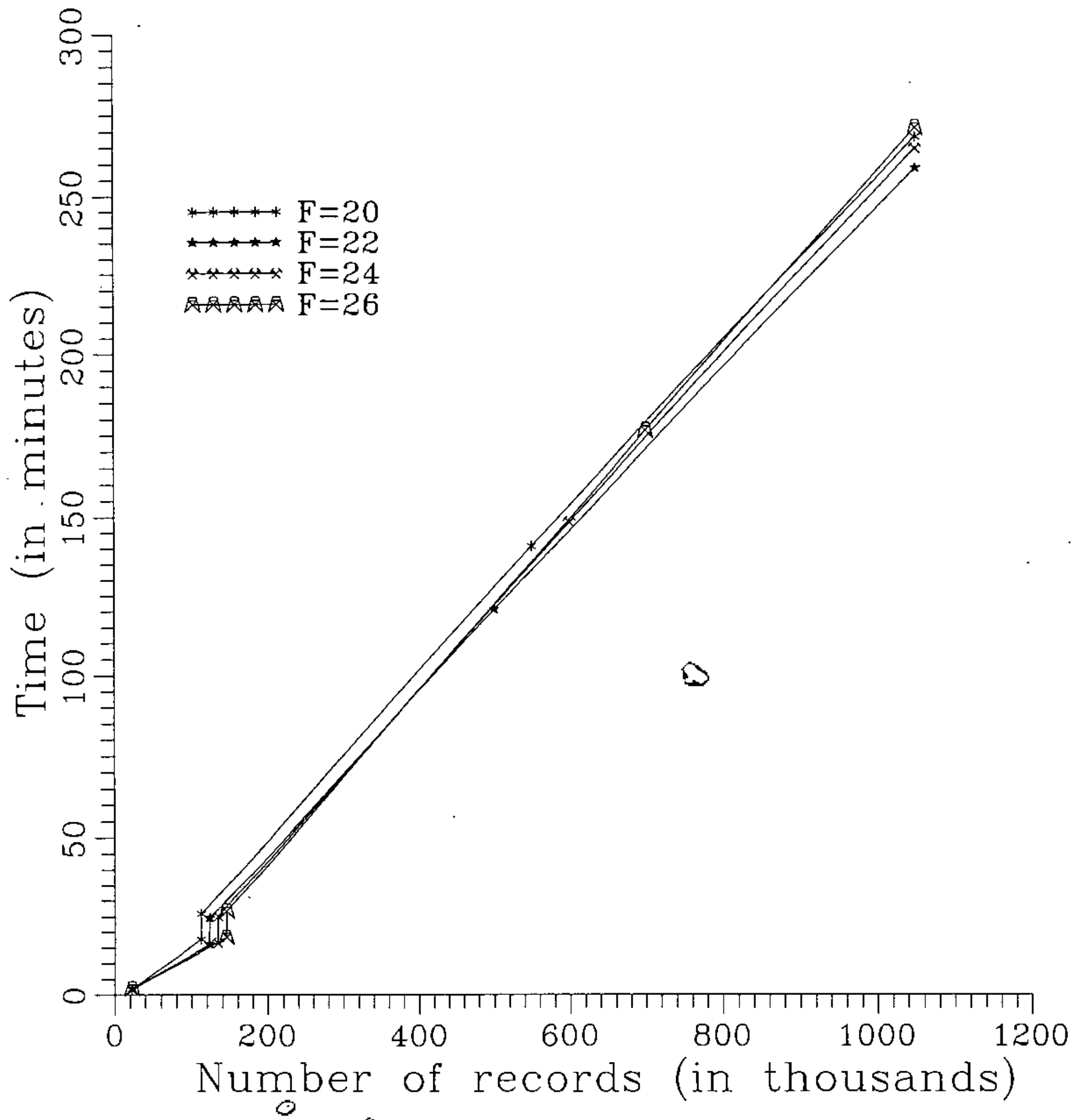
Again, sorting time is not minimum with *F*=22 when number of records to be sorted is small (<200,000). For this case *F*=26 is better than other values of *F* as shown in graphs 6.1-6.5.

For this algorithm there is an abrupt change in sorting time when the number of records to be handled cross the numbers 112, 123, 134 and 146 thousand for the case of *F*=20, 22, 24 and 26 respectively. This is because, at these values of number of records the required number of passes over the data increases from two to three.

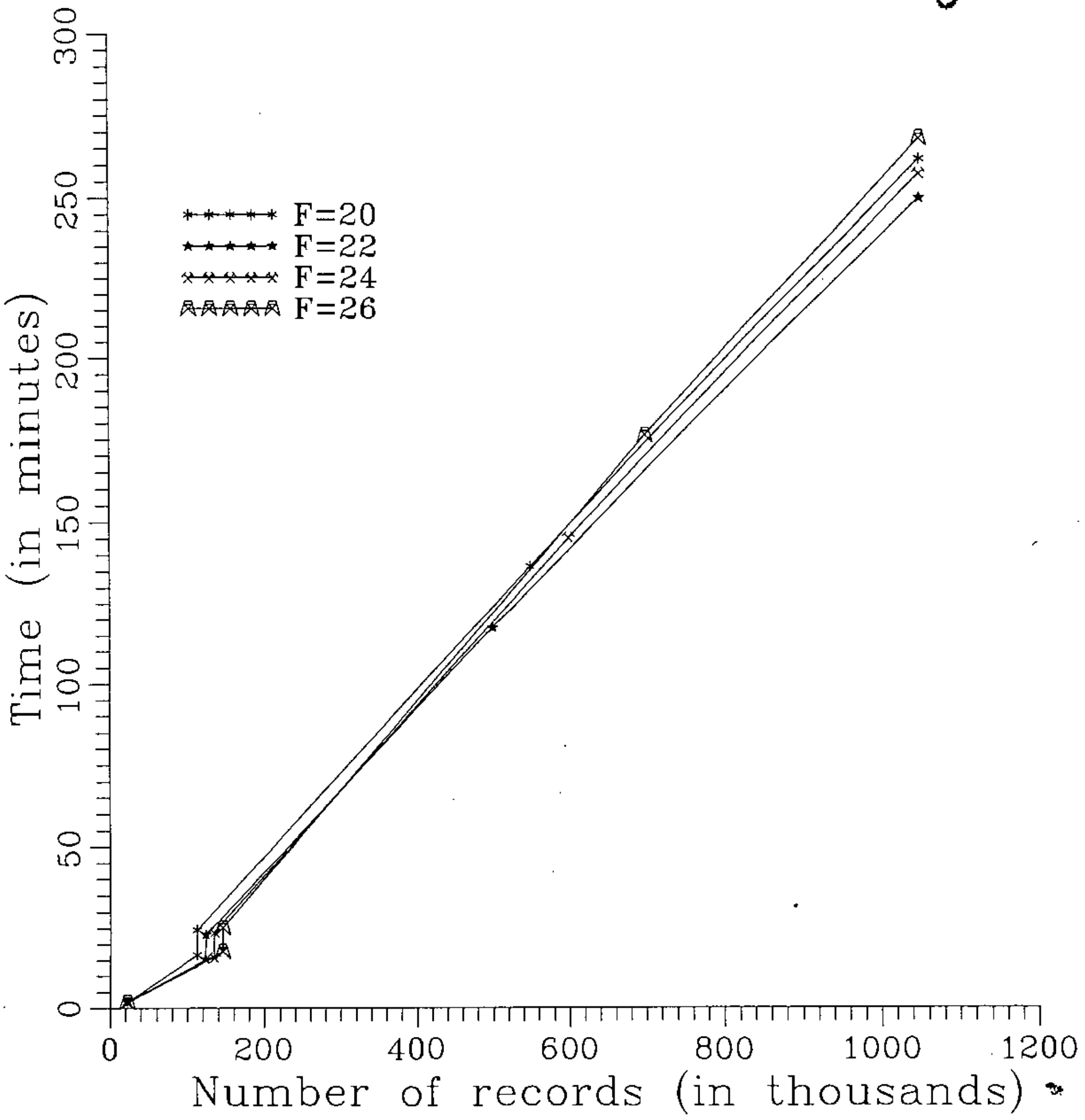
Graphs 6.6-6.9 show that for small number of records, the sorting time is not sensitive to *BR* and for all values of *BR* it is nearly same. But as the number of records increases, the sorting time gradually become more and more sensitive to *BR*.



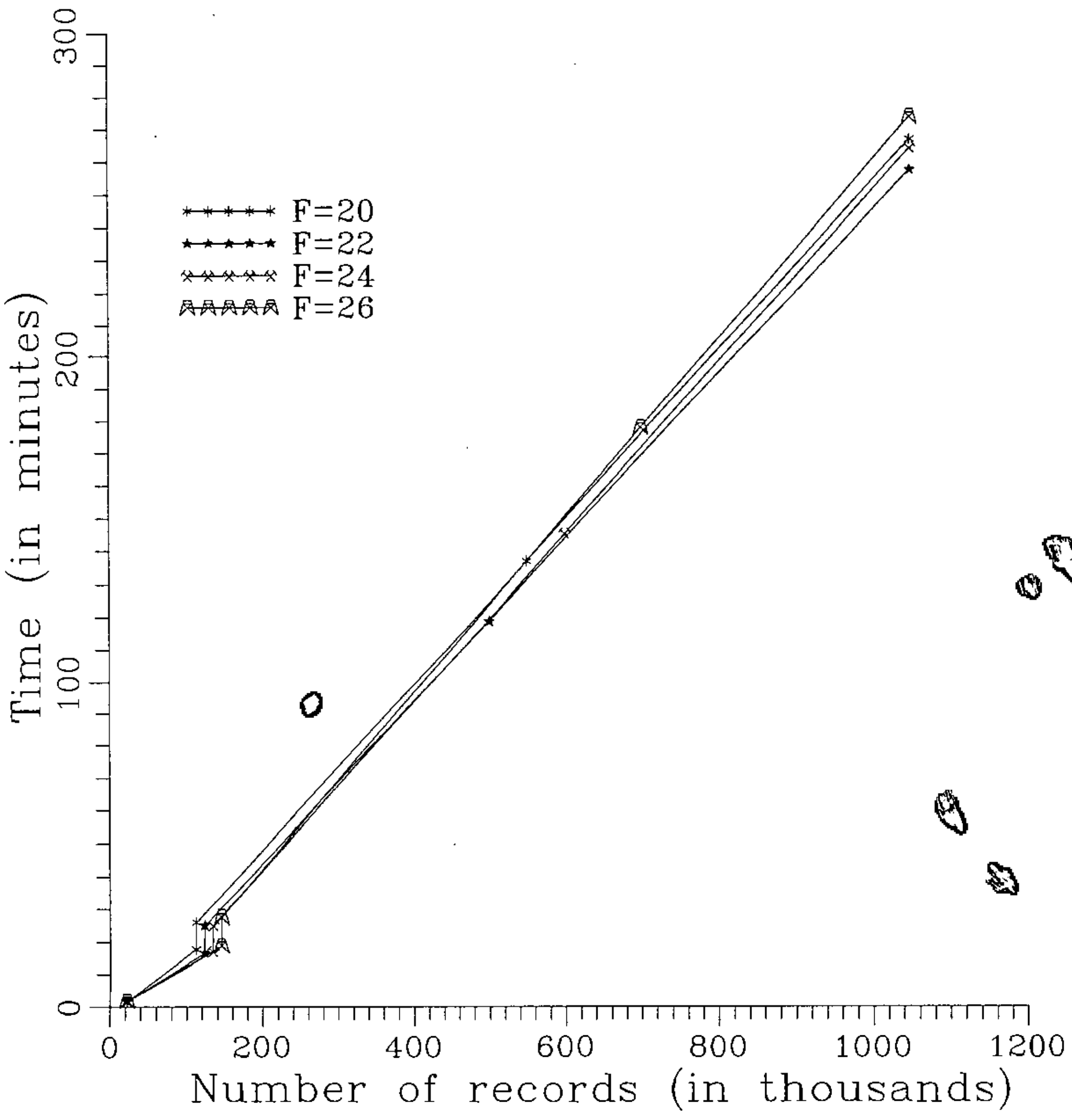
Graph 6.1 Balanced merge
(BR=4)



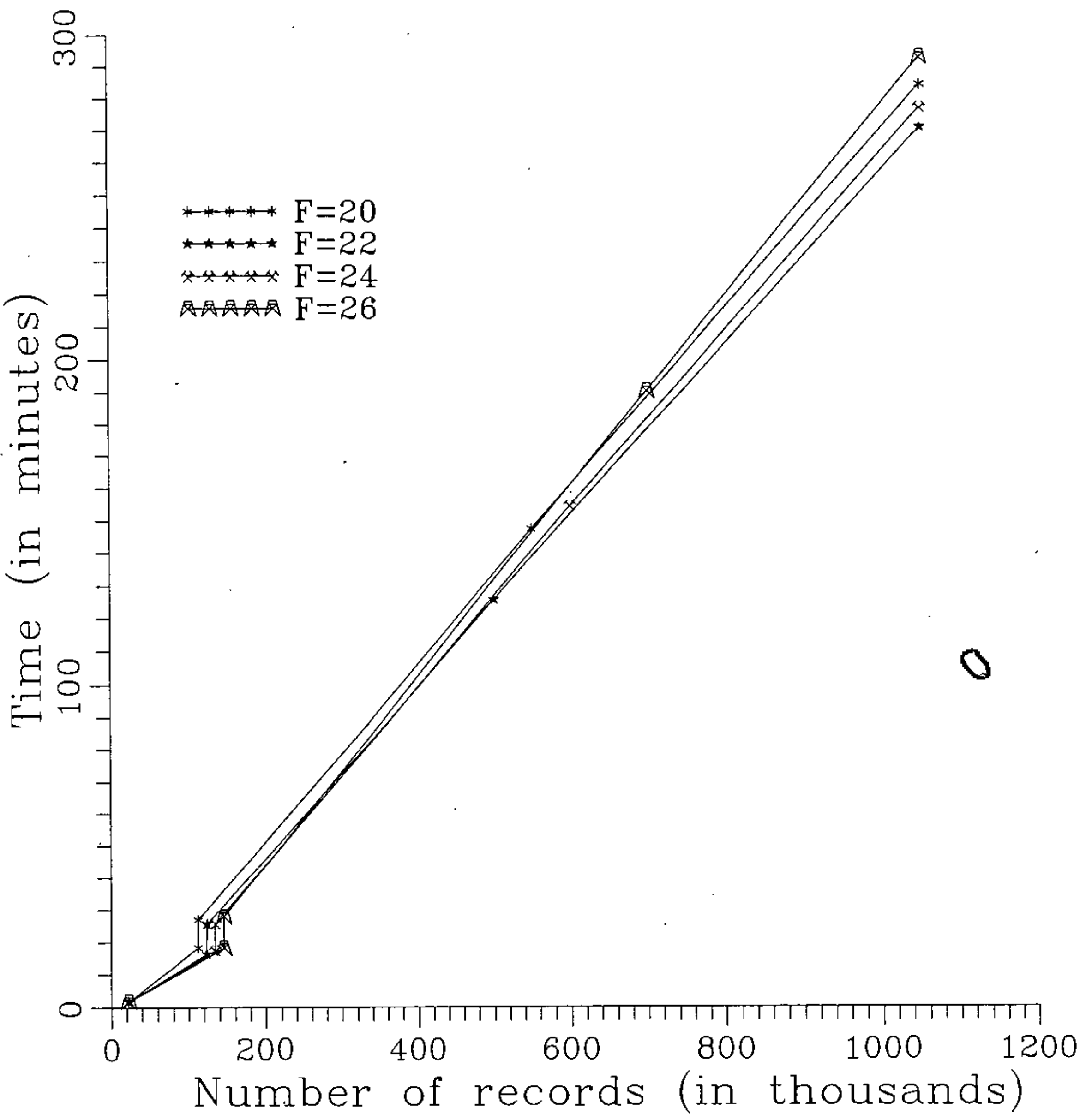
Graph 6.2 Balanced merge
(BR=8)



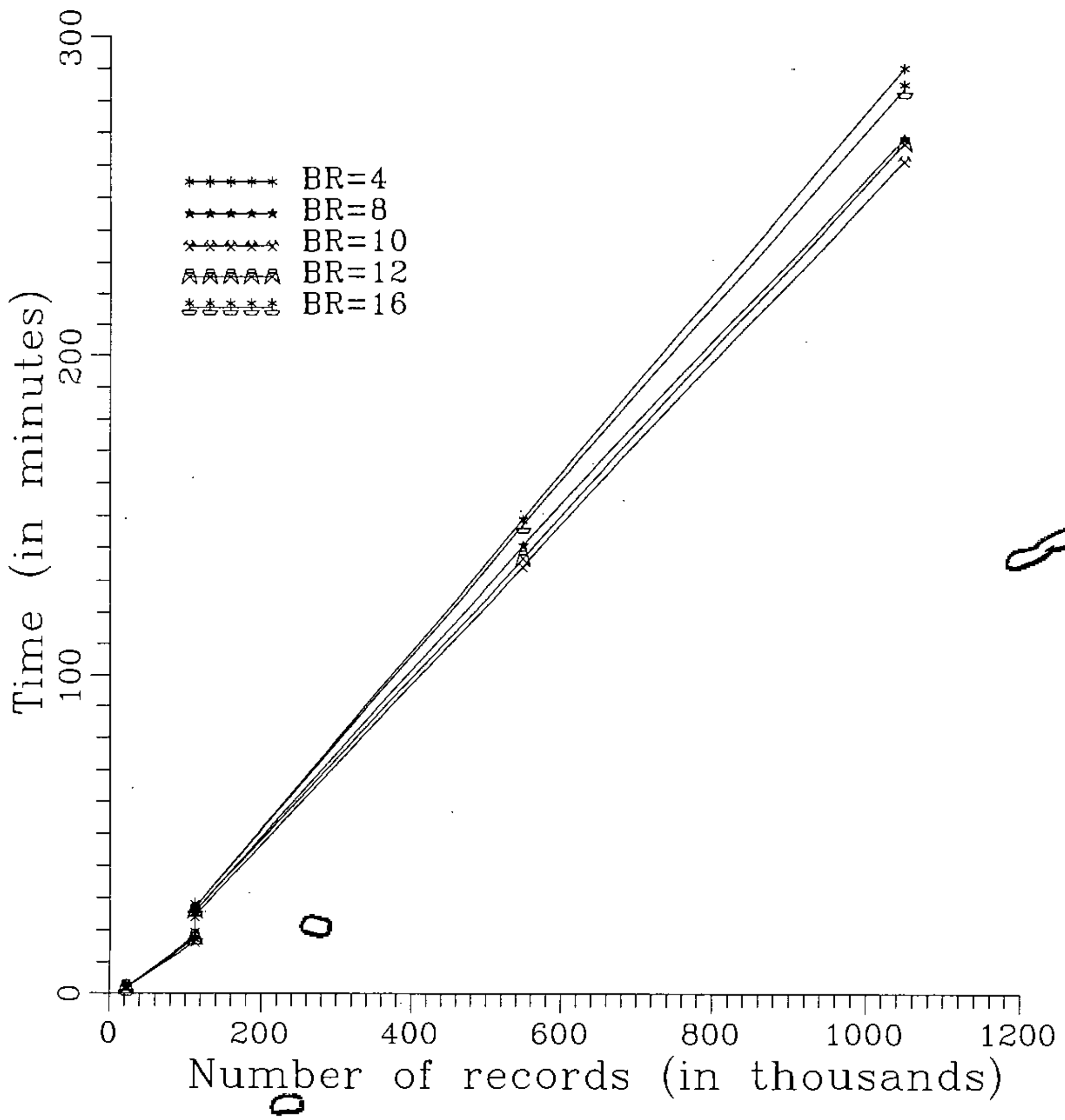
Graph 6.3 Balanced merge
(BR=10)



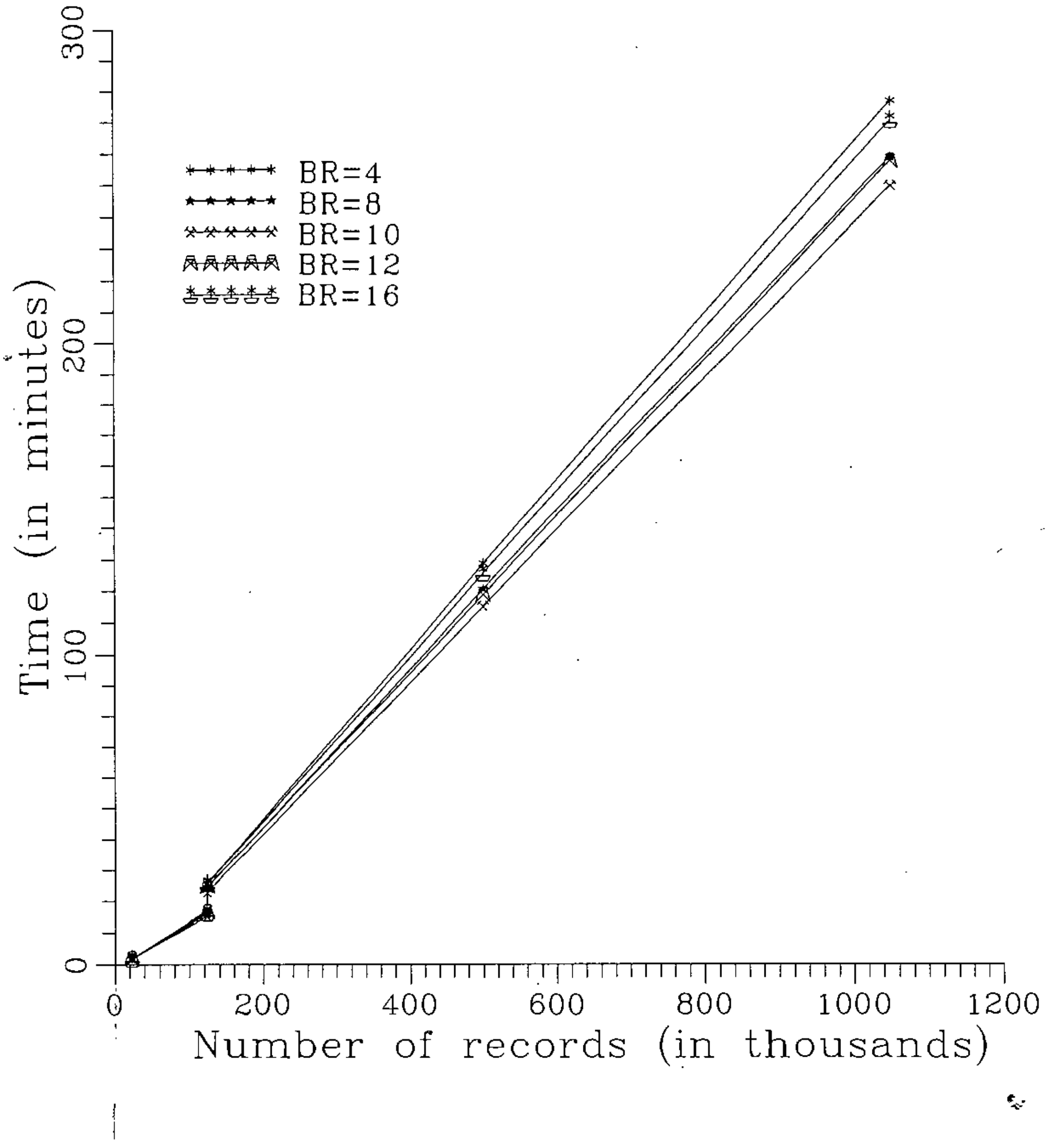
Graph 6.4 Balanced merge
(BR=12)



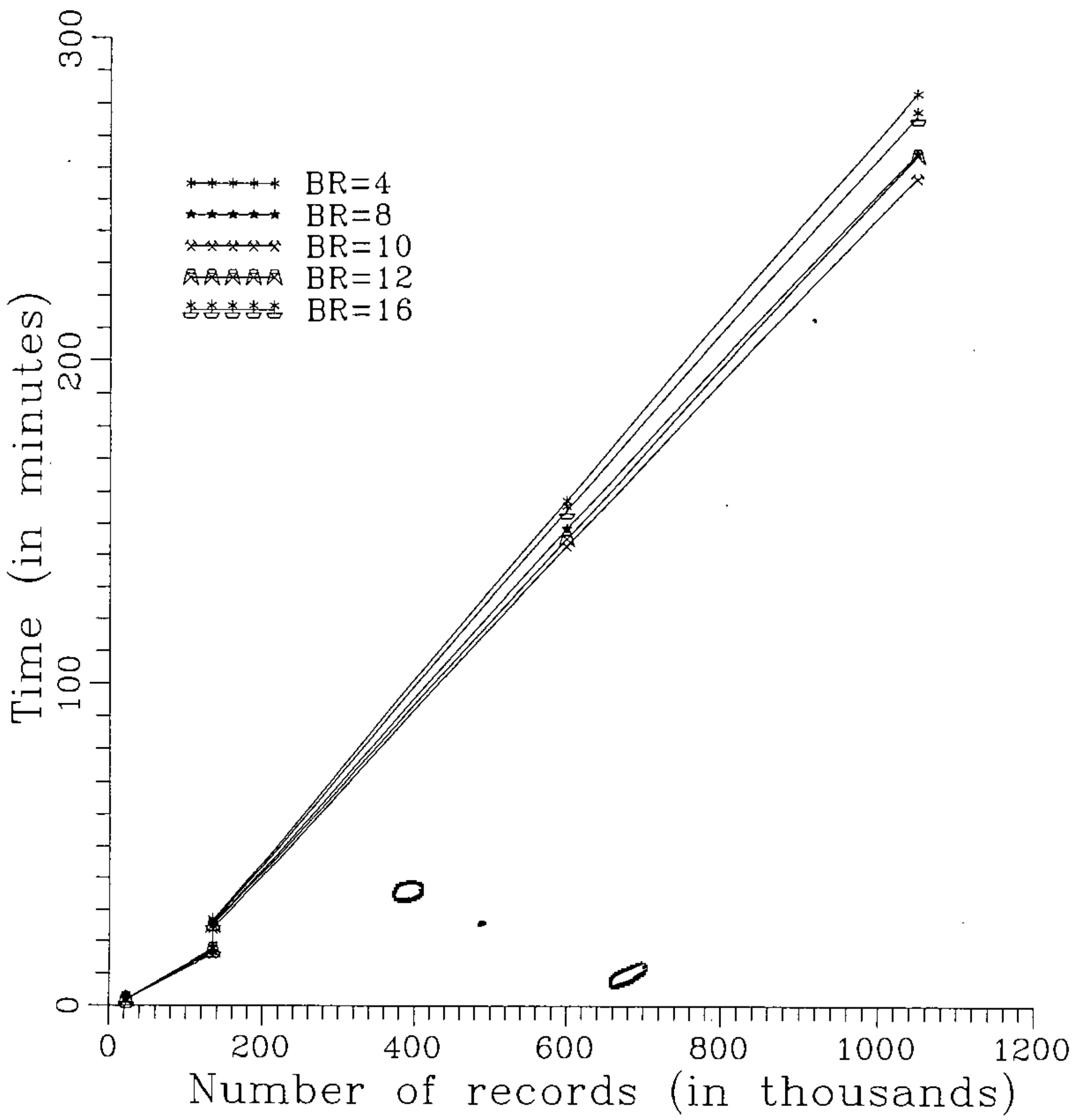
Graph 6.5 Balanced merge
(BR=16)



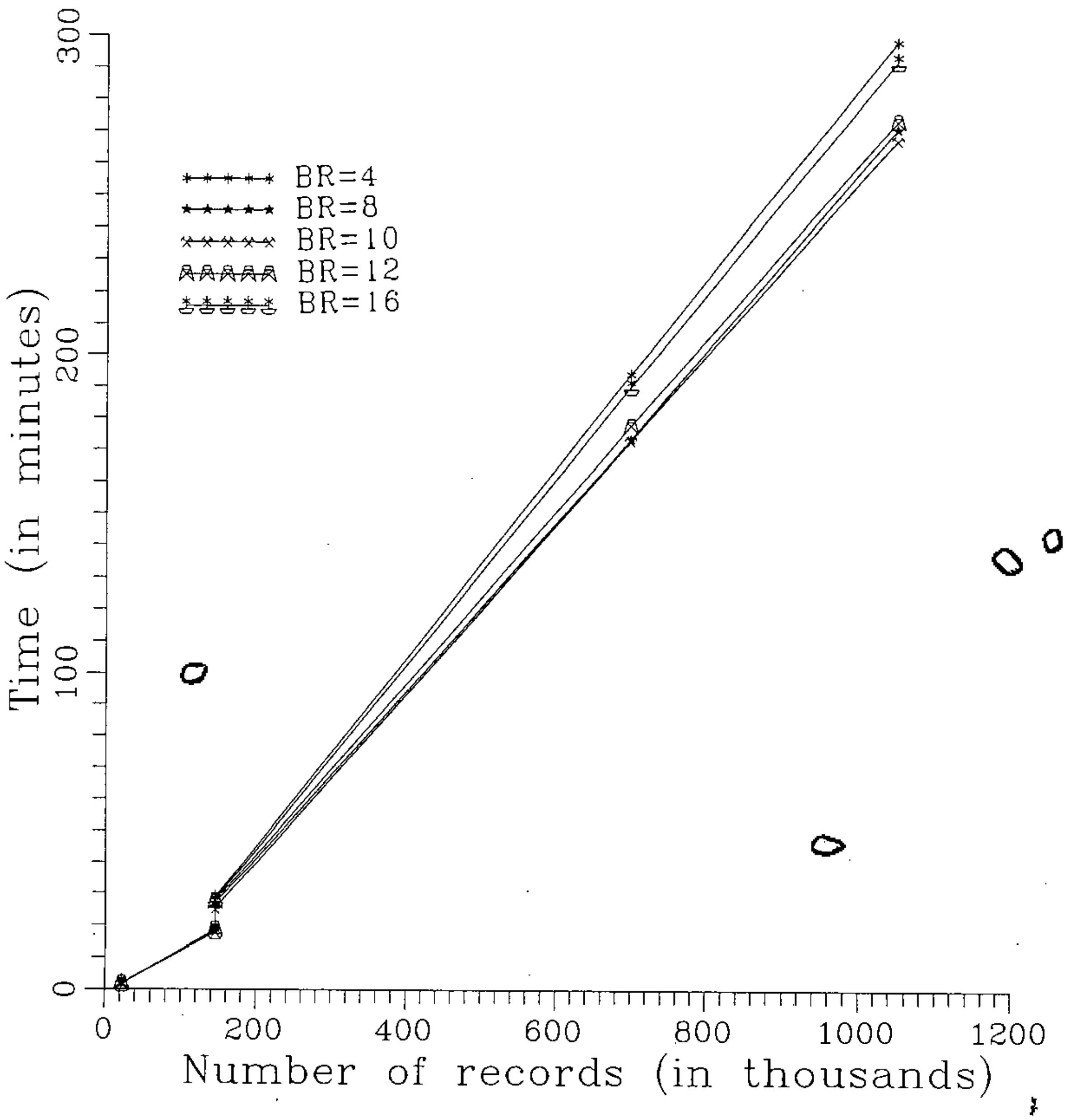
Graph 6.6 Balanced merge
(with F=20)



Graph 6.7 Balanced merge
(with F=22)



Graph 6.8 Balanced merge
(with F=24)



Graph 6.9 Balanced merge
(with F=26)

In this case the variation in sorting time for different number of files for different values of *BR* is not constant. From graphs 6.1-6.5 it is clear that this variation is smallest when *BR* is 8.

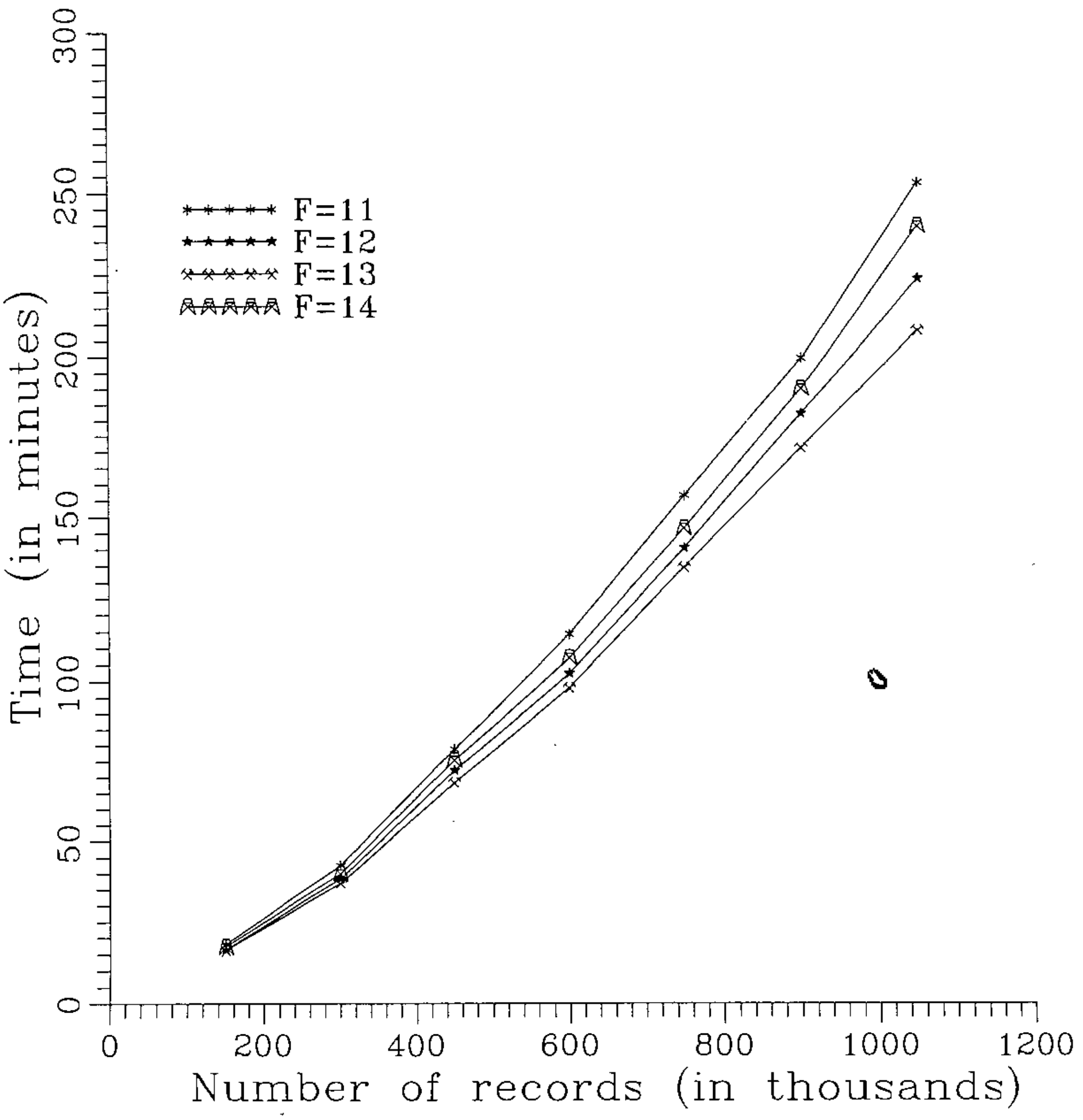
6.3 Polyphase Merge:

For the purpose of study the performance of polyphase merge algorithm under the variation of different sorting parameters, we have designed polyphase merge routine in C language, similar to that of balanced merging. We have taken *F* and *BR* as the parameters for this sorting technique. At the beginning, by several trials of sorting data, we have found that for $F=13$ the sorting time is minimum and we have taken the experimental data for $F=11$, 12, 13 and 14.

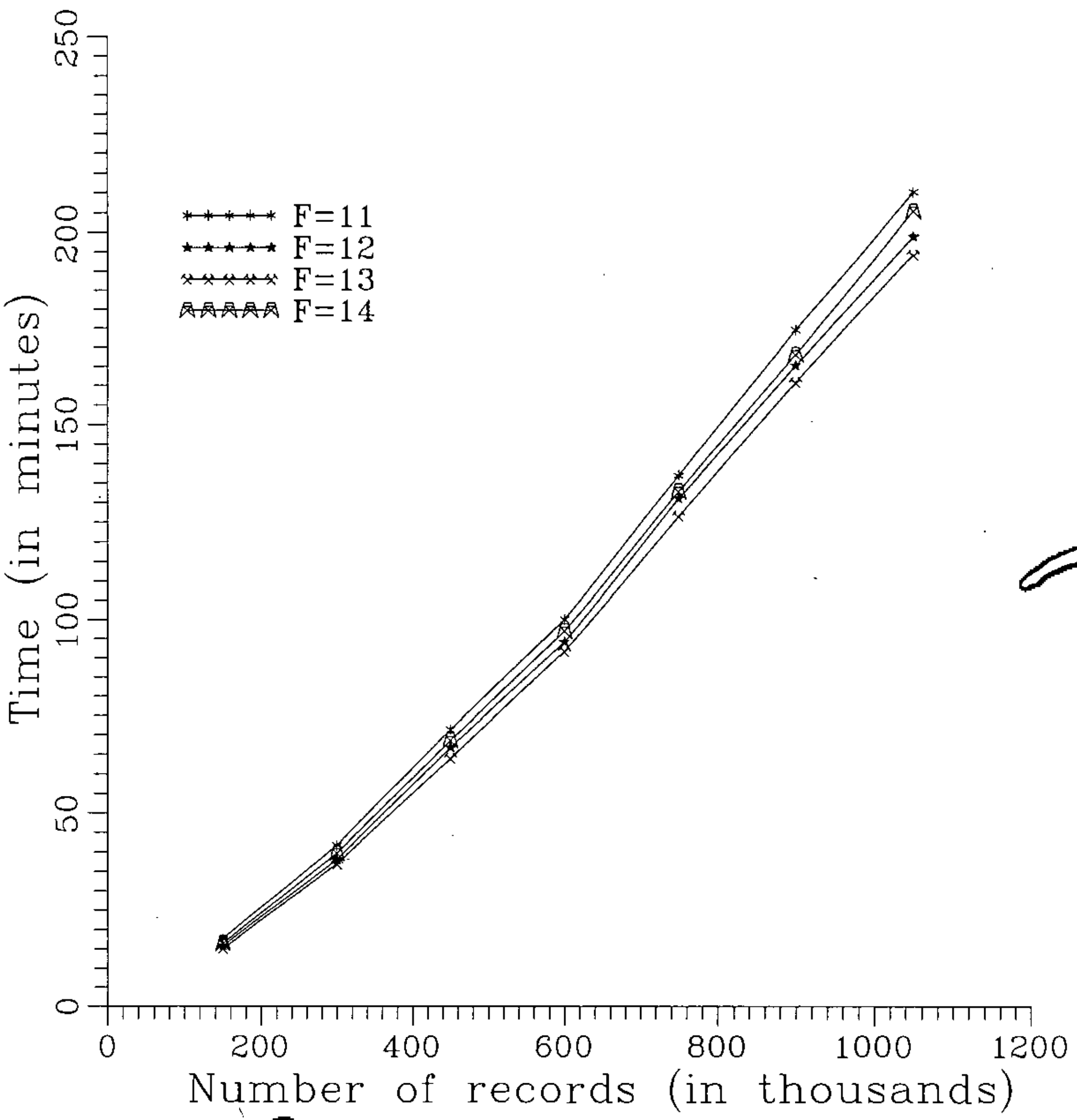
For different values of *BR* (4, 8, 10, 12 and 16) by varying *F*, sorting times have been taken. Using these data graphs 6.10-6.18 have been plotted. Graphs 6.10-6.14 reveal that for $F=13$ sorting time is minimum for all values of *BR*. Again, sorting time is minimum of all when $F=13$ and $BR=10$, which is shown in graph 6.12.

For smaller number of records the sorting time is not very sensitive to *BR* and *F*. But as the number of records increases the sorting time become more sensitive to *BR* and *F*, similar to balanced merging. The variation in sorting time for different number of files with different *BR* is not constant. From graphs 6.10-6.14, it is observed that this variation is minimum when $BR=8$ (see graph 6.11).

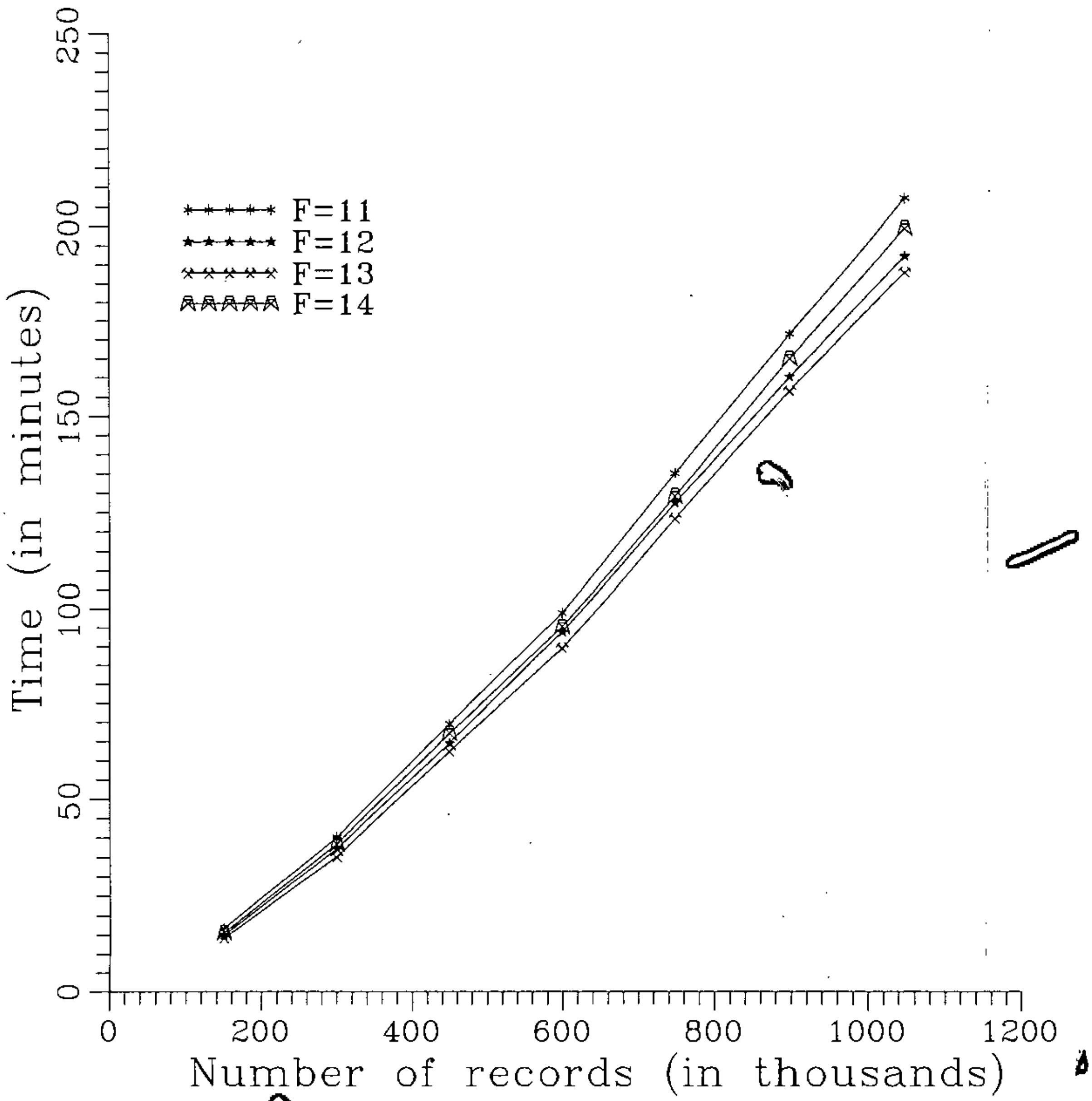
Again graphs 6.15-6.18 show that for small and larger values of *F* compared to the optimal one (which is 13), the sorting time increases sharply for values of *BR* significantly different from the optimal one.



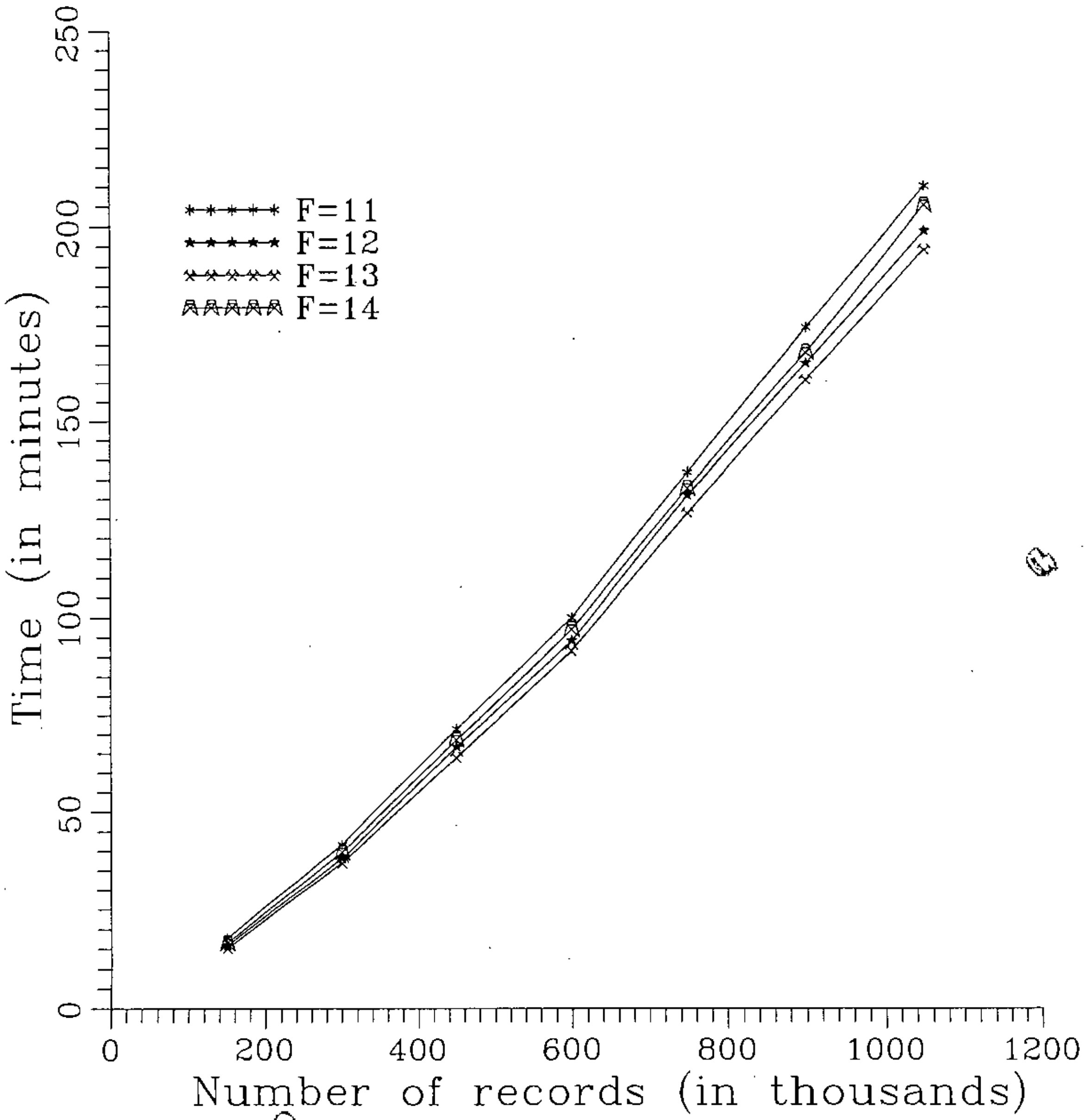
Graph 6.10 Polyphase merge
(BR=4)



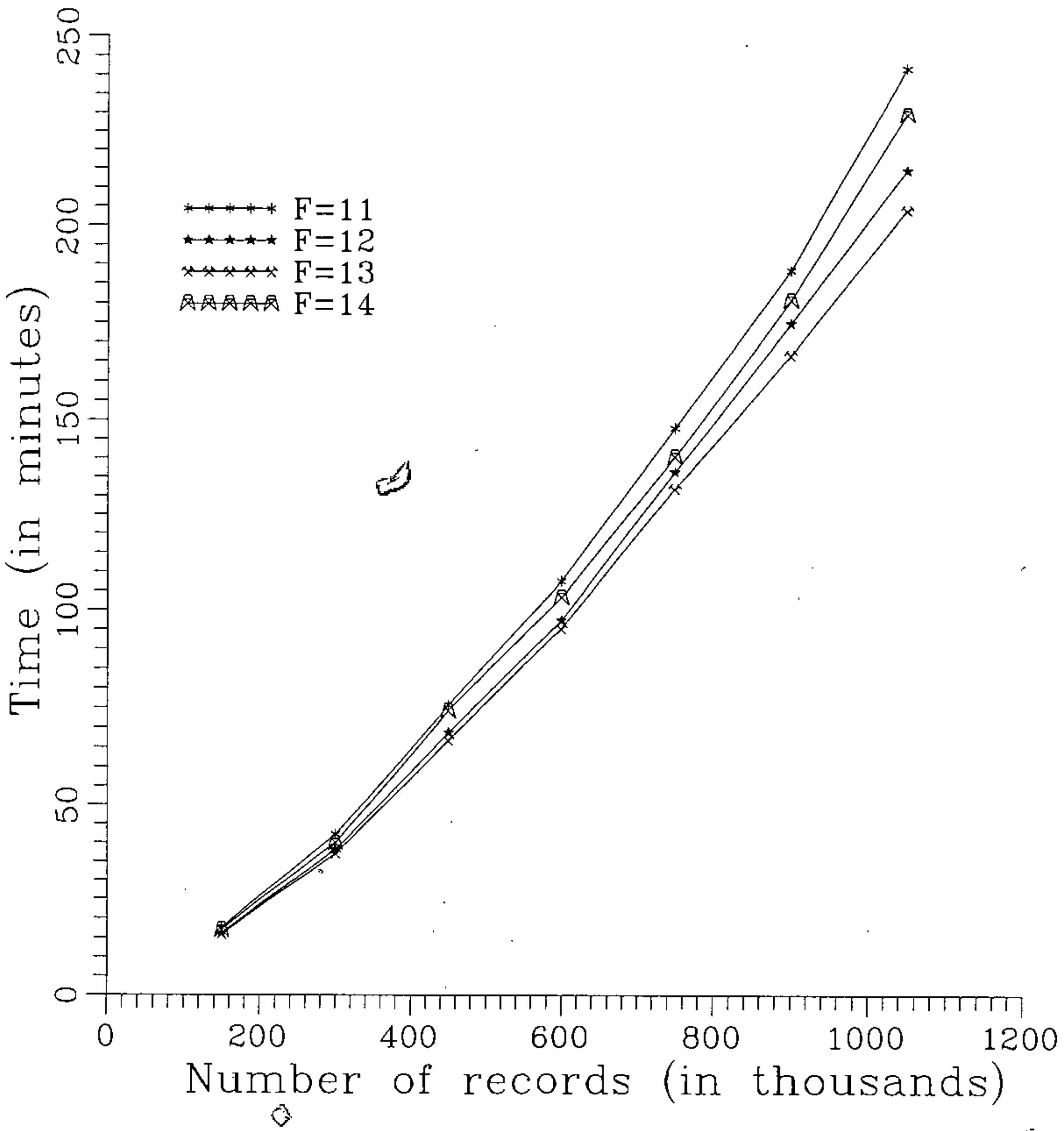
Graph 6.11 Polyphase merge
(BR=8)



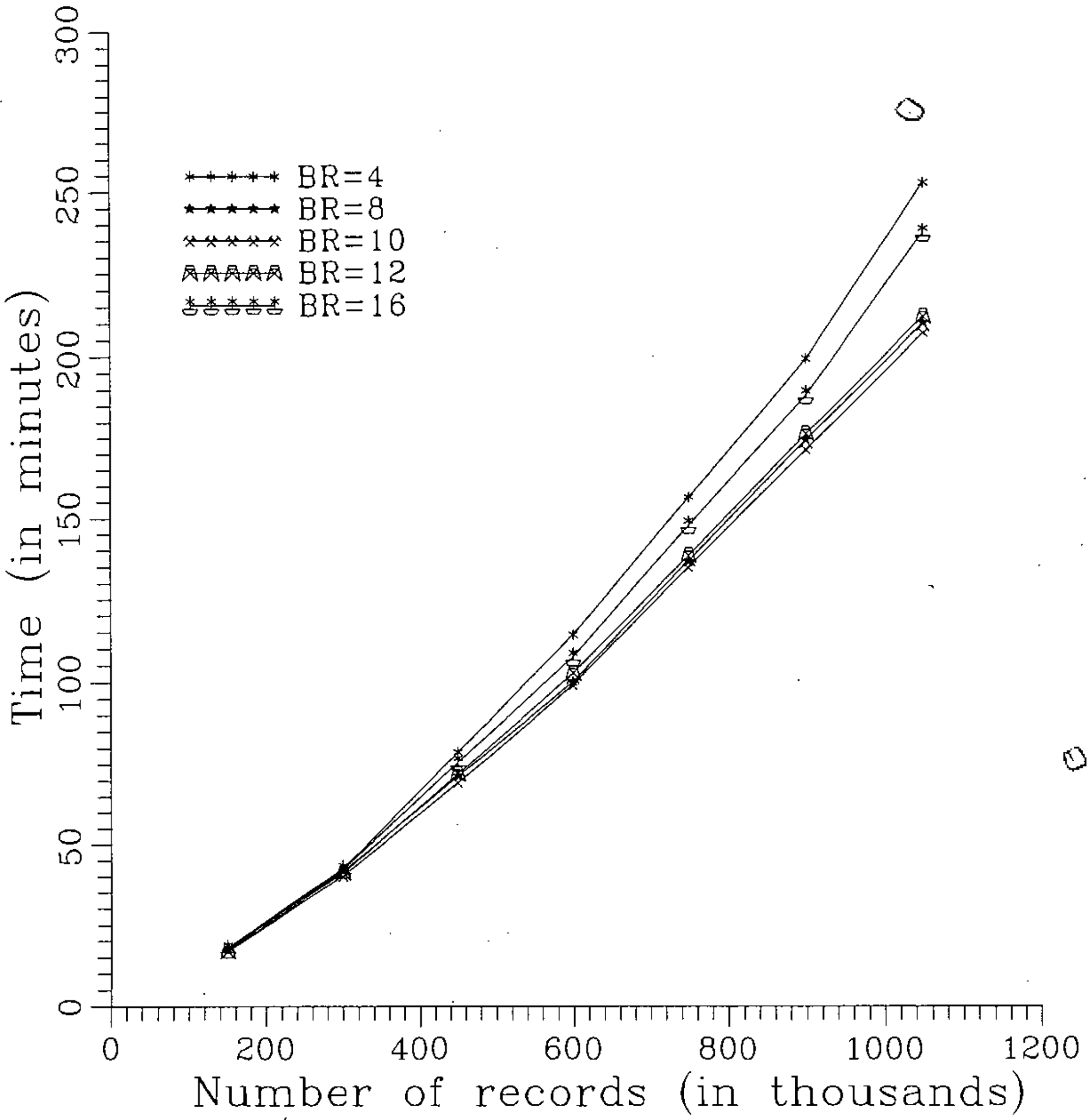
Graph 6.12 Polyphase merge
(BR=10)



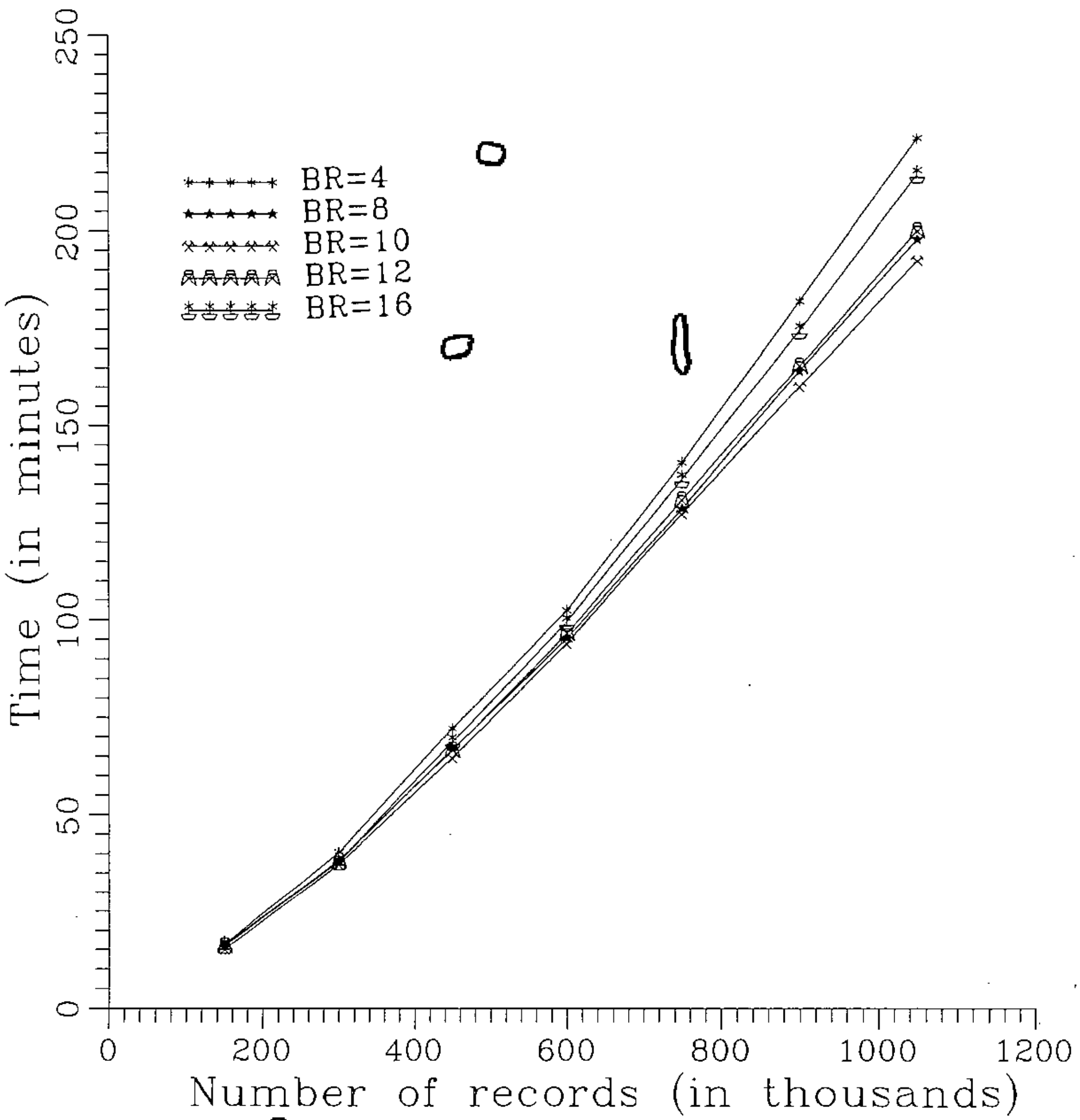
Graph 6.13 Polyphase merge
(BR=12)



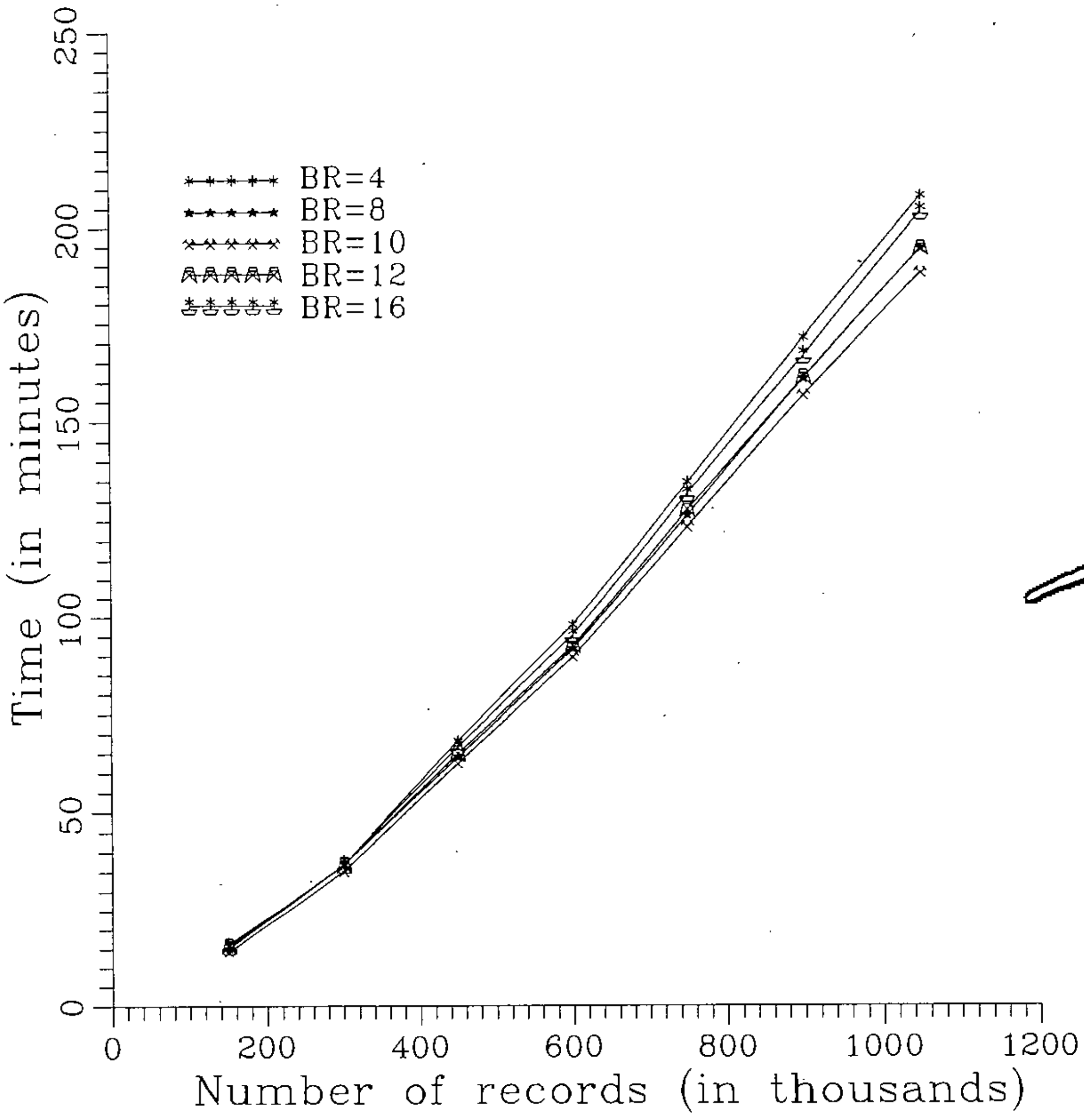
Graph 6.14 Polyphase merg
(BR=16)



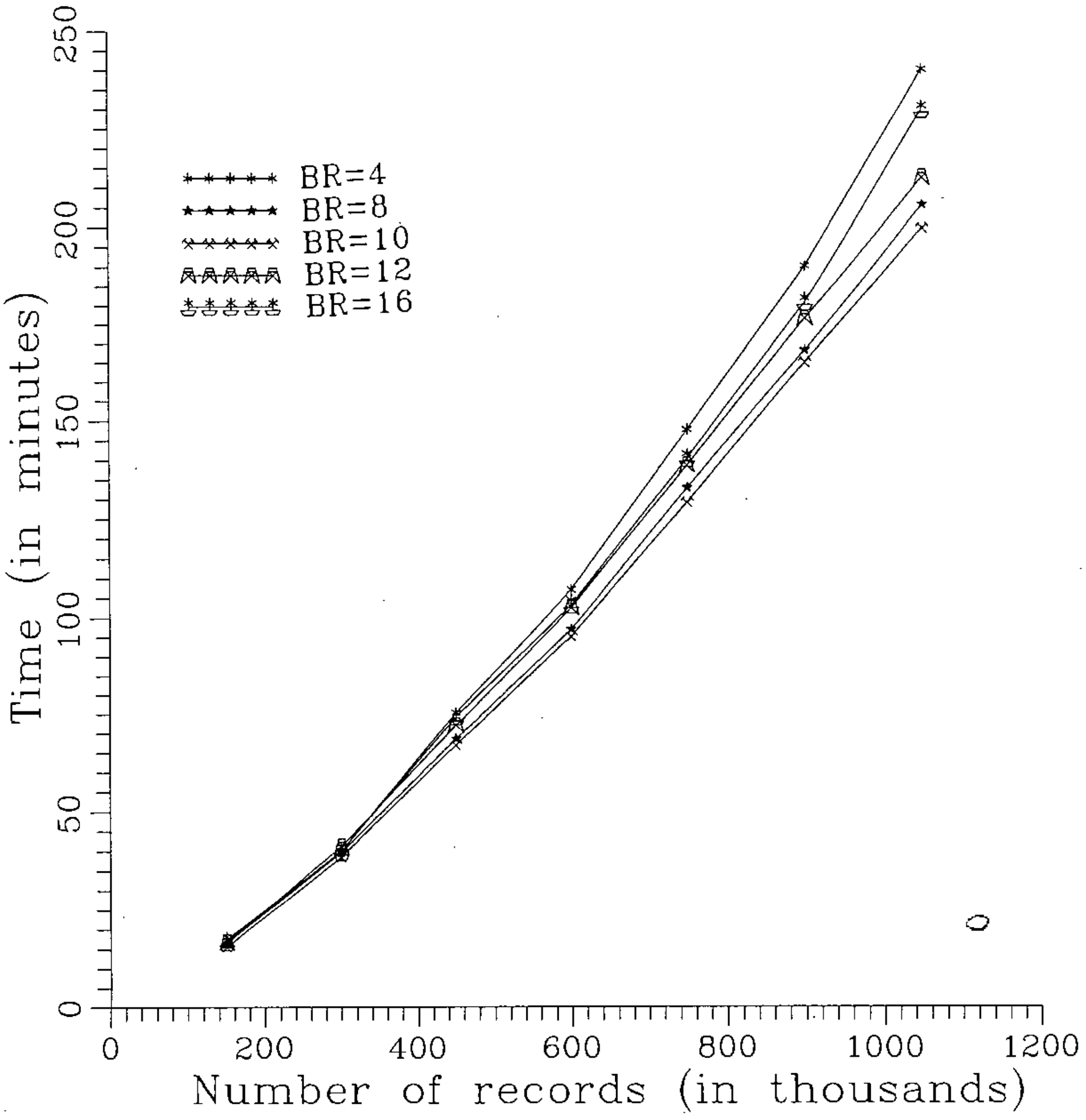
Graph 6.15 Polyphase merge
(with F=11)



Graph 6.16 Polyphase merge
(with F=12)



Graph 6.17 Polyphase merge
(with F=13)



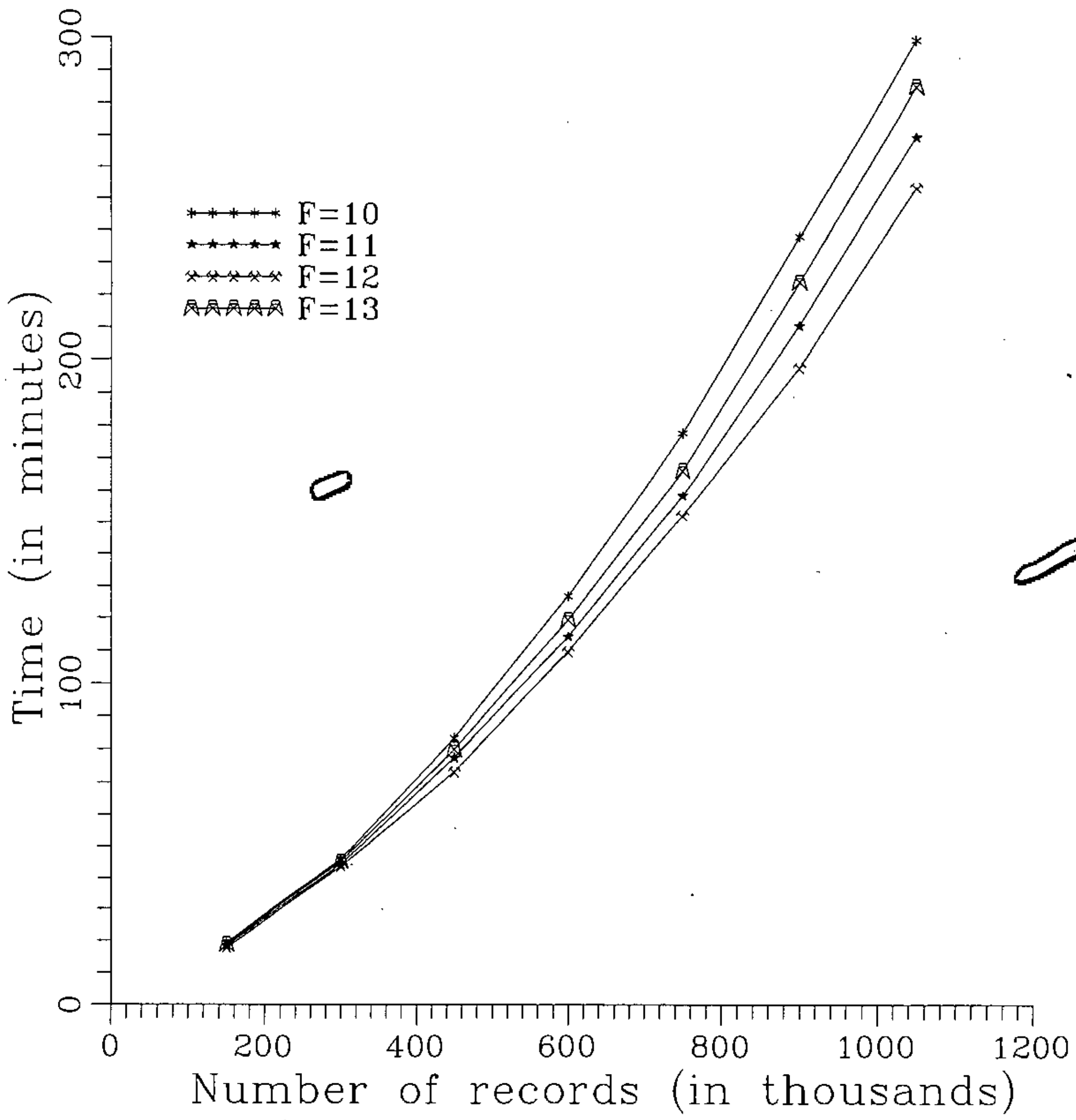
Graph 6.18 Polyphase merge
(with F=14)

6.4 Cascade Merge:

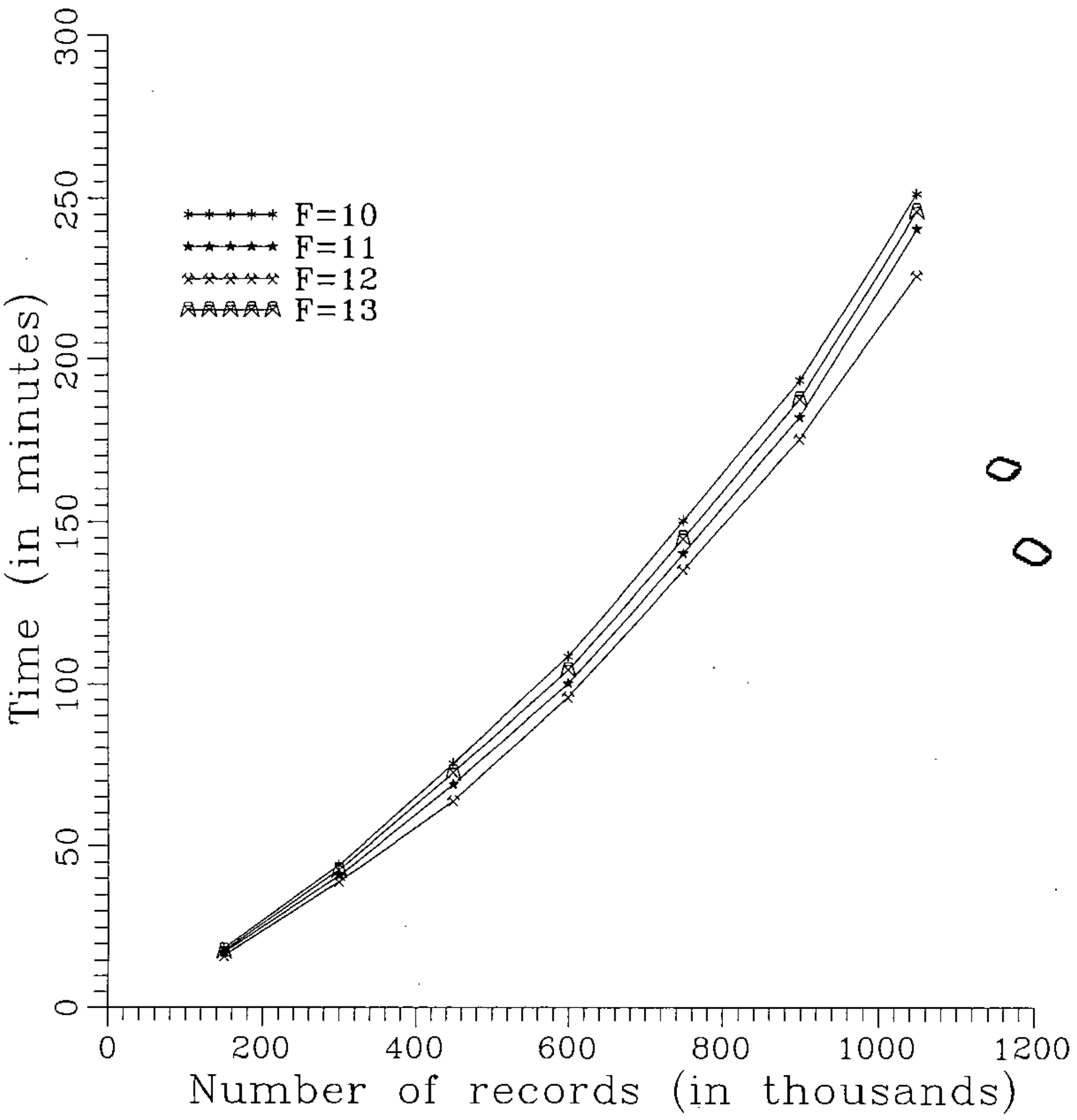
Cascade merge routine has been developed in C, to study the performance of cascade sorting technique under the variation of different sorting parameters. The variable *F* and *BR* have been taken as sorting parameters for this algorithm also. At the start of the experiment using several trials, the optimum value of *F* for which the sorting time is minimum is found. For this algorithm the optimum value of *F* is 12. For other values of *F* the sorting time is greater than that of the *F*=12. The experimental data are generated for *F*=10, 11, 12 and 13 with different values of *BR*.

The data obtained from the above experiments have been plotted in graphs 6.19-6.27. From graphs 6.24-6.27, it is apparent that the sorting time is minimum when *BR*=10. Graphs 6.19-6.23 reveals that for *F*=22, sorting time is minimum for all values of *BR*. Thus, sorting time is minimum of all when *F*=12 and *BR*=10. Sorting time changes sharply for values of *F* and *BR* significantly different from the corresponding optimal values.

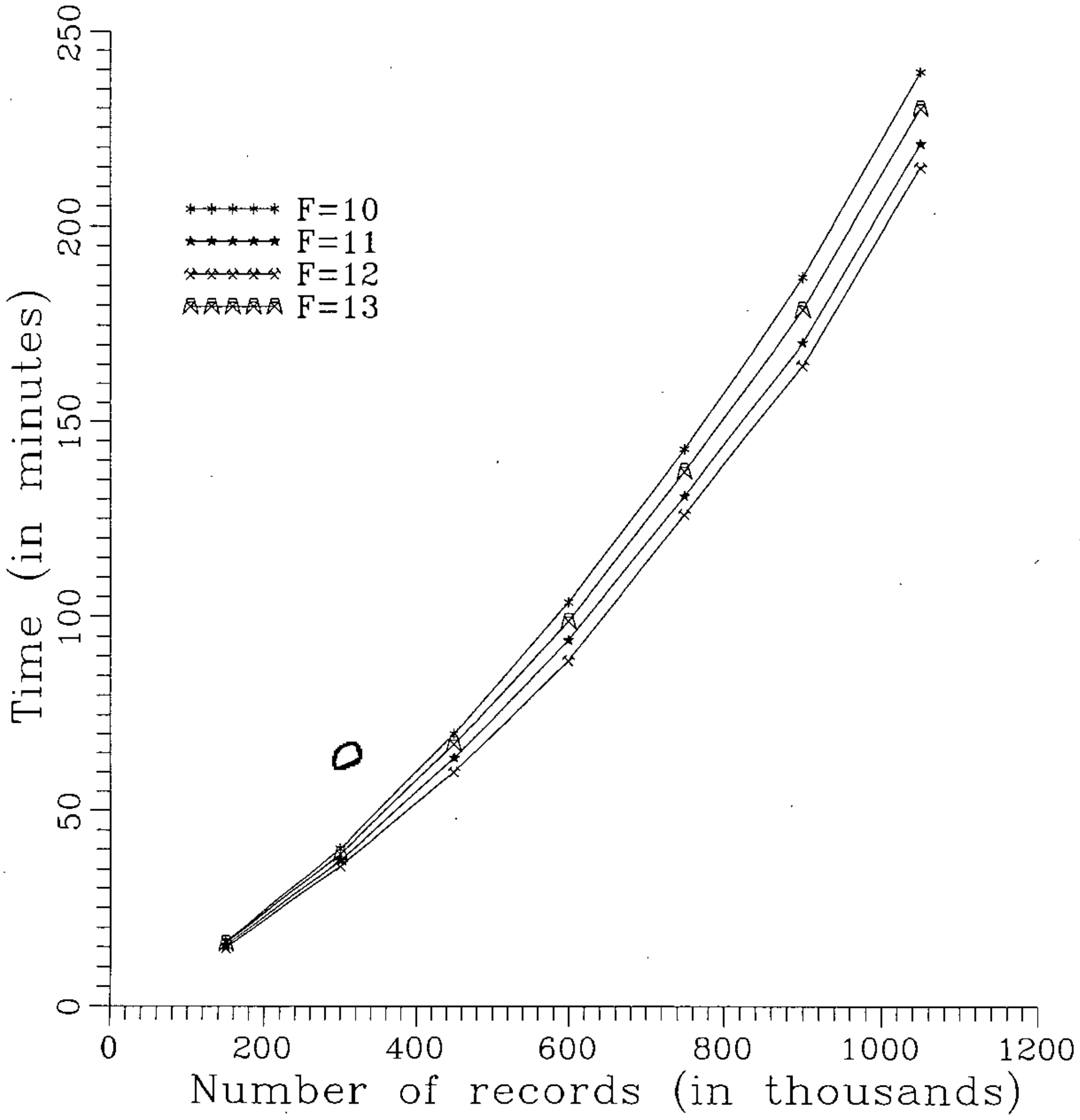
It can be seen from graphs 6.19-6.23 that the sorting time changes smoothly for consecutive number of initial runs, which is different from balanced merging but similar to polyphase merging. It is also apparent from these graphs that the curves in every plot become divergent as the number of records increases. It can be concluded from here that difference in sorting time for different order of merging is more sensitive as the number of records increases.



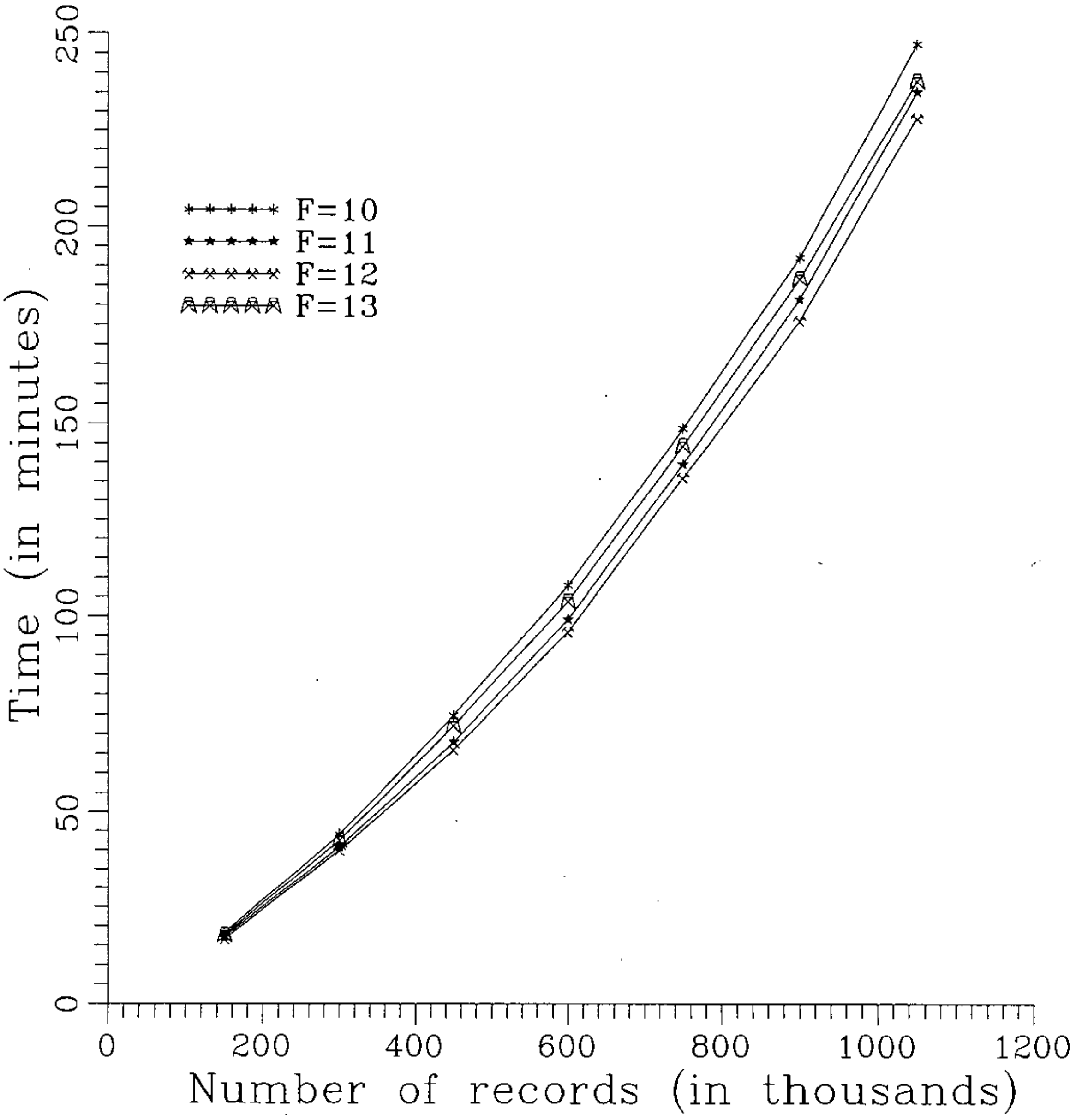
Graph 6.19 Cascade merge
(BR=4)



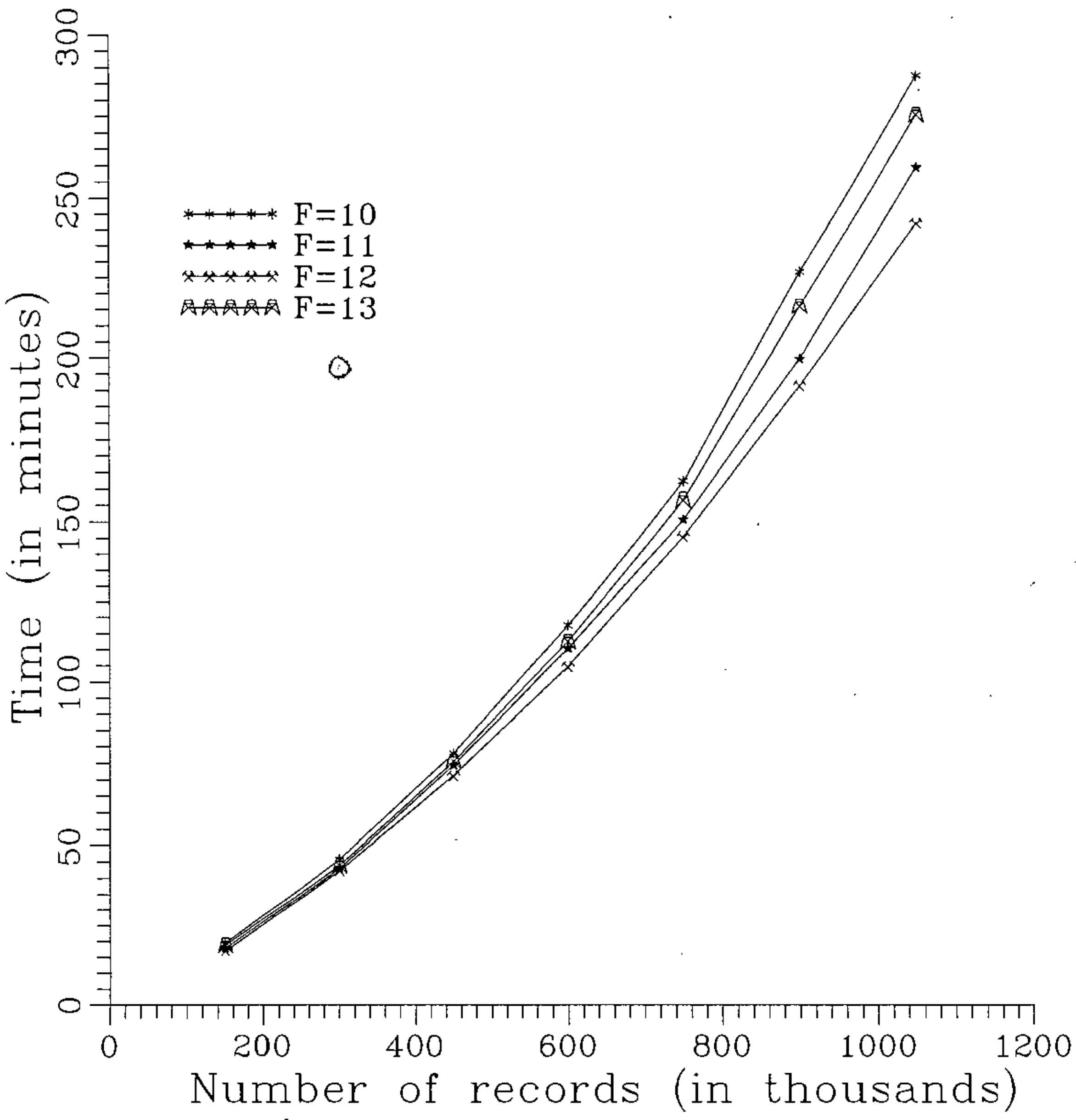
Graph 6.20 Cascade merge
(BR=8)



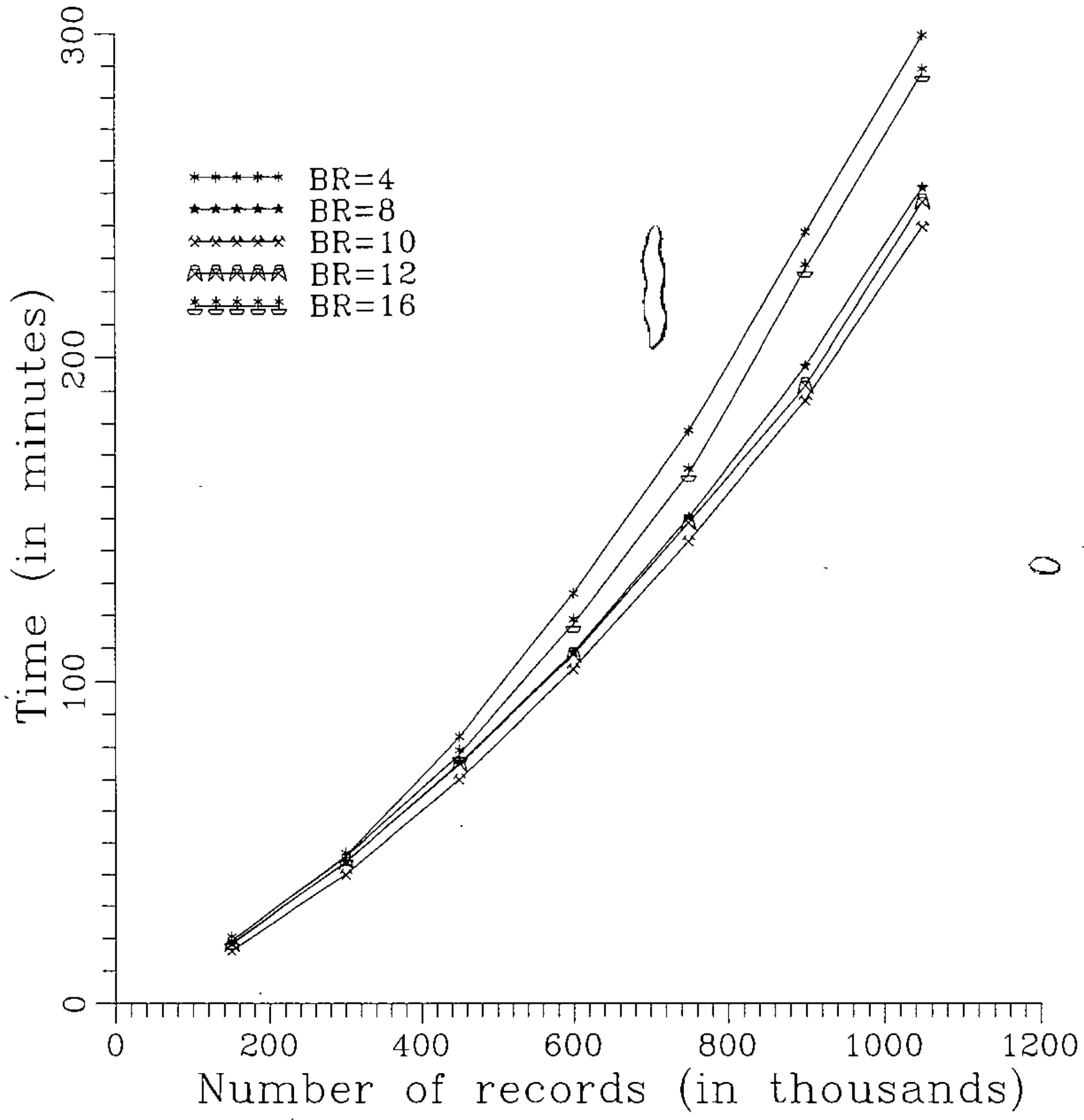
Graph 6.21 Cascade merge
(BR=10)



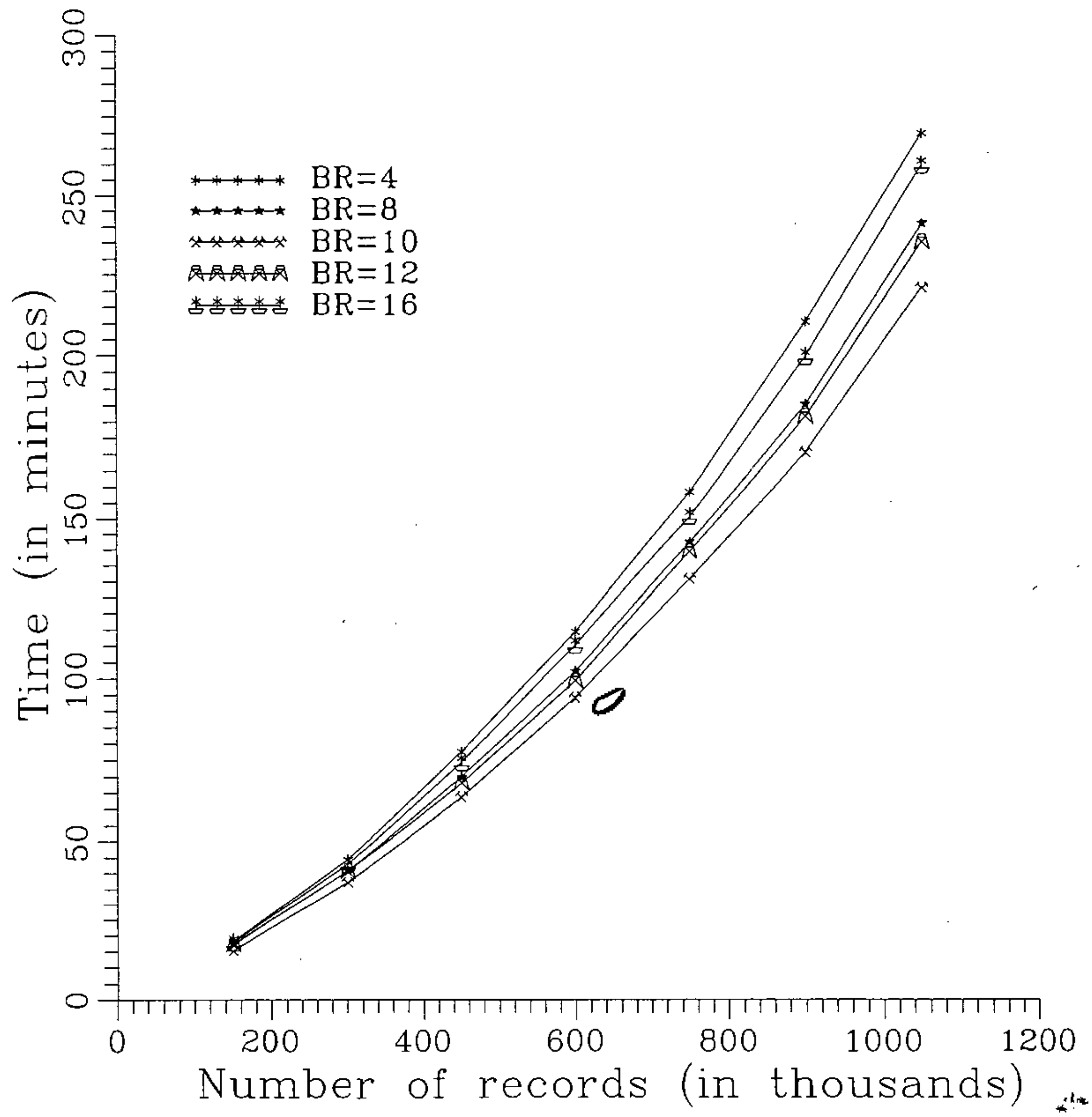
Graph 6.22 Cascade merge
(BR=12)



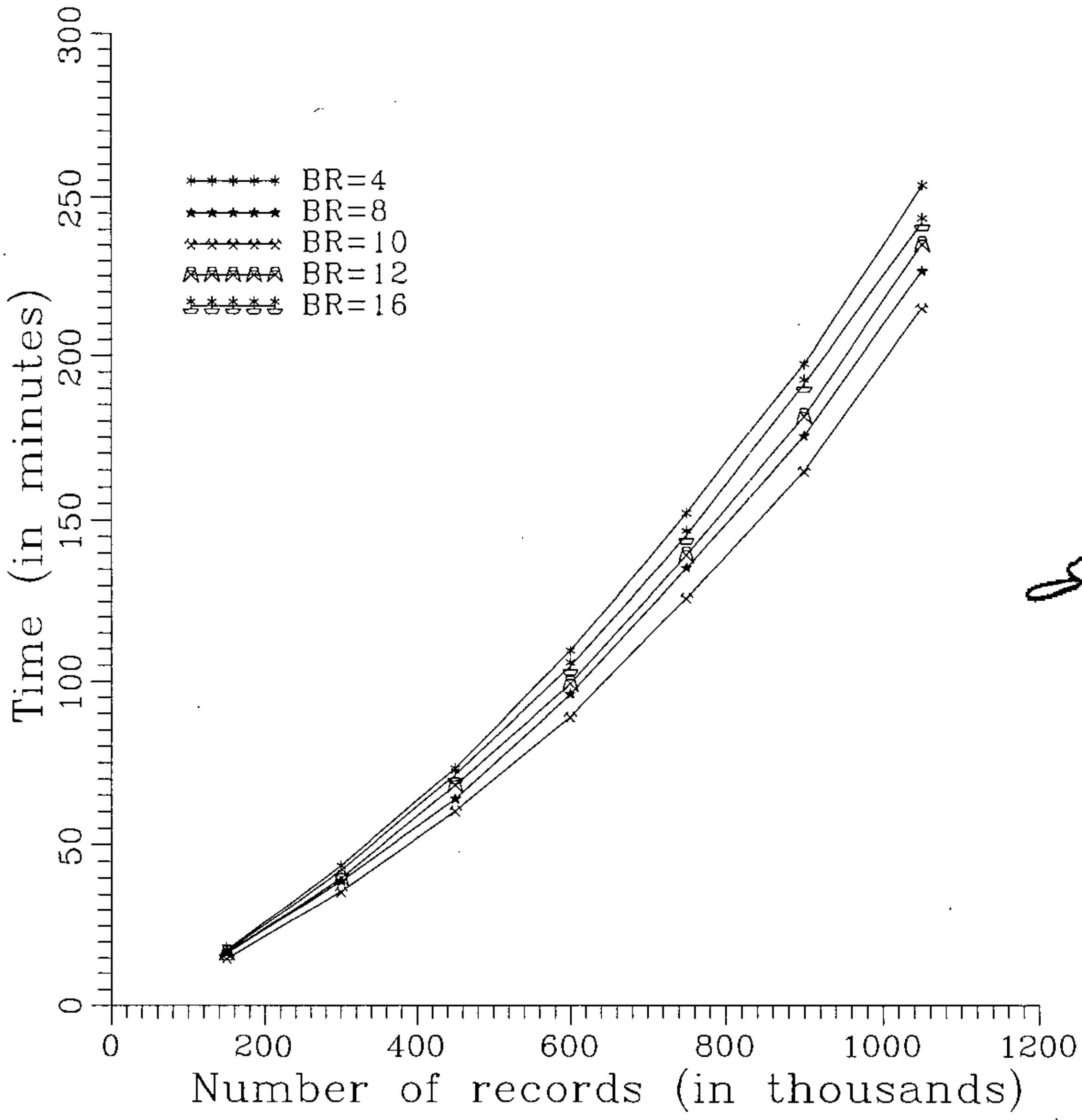
Graph 6.23 Cascade merge
(BR=16)



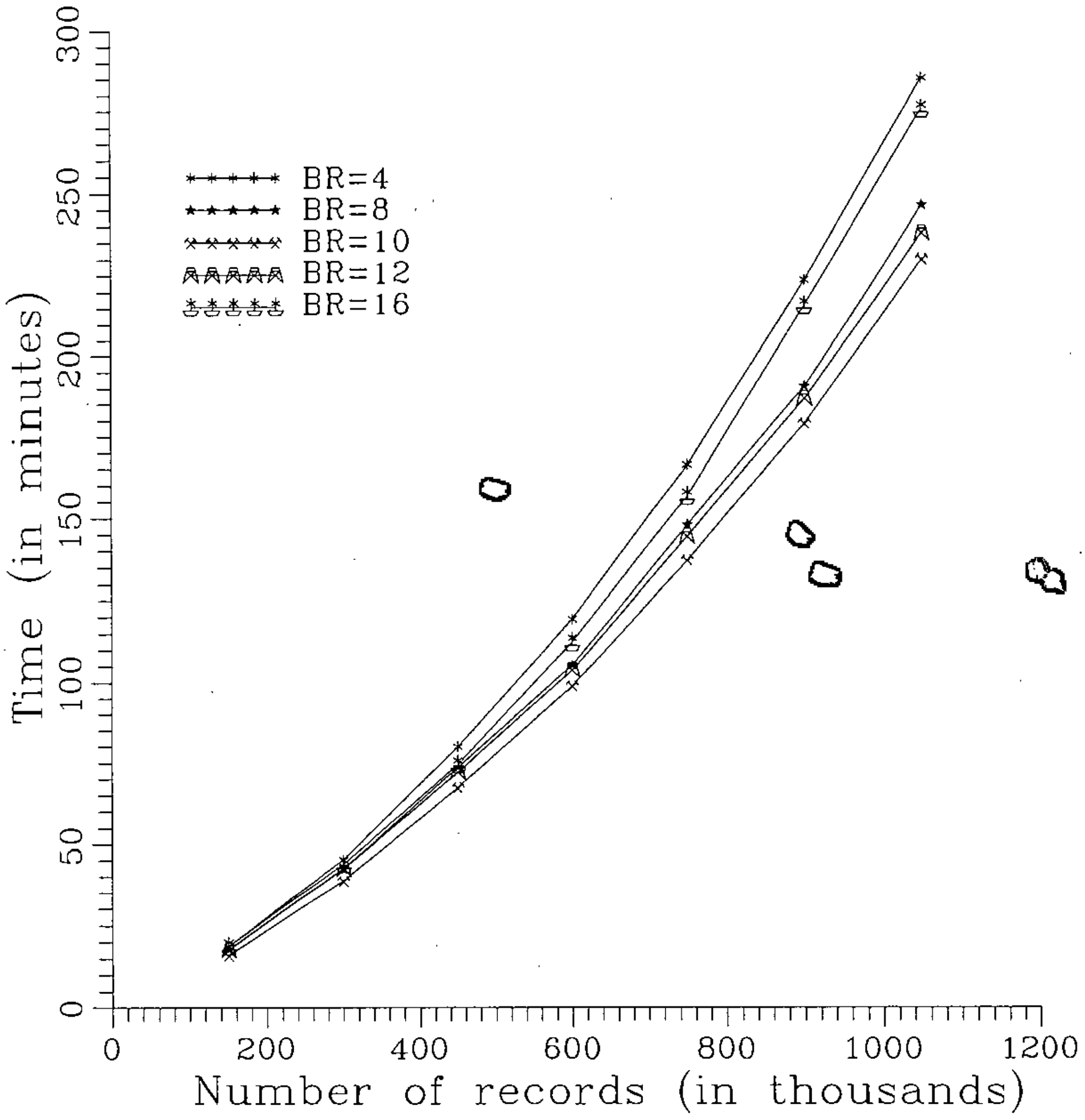
Graph 6.24 Cascade merge
(with F=10)



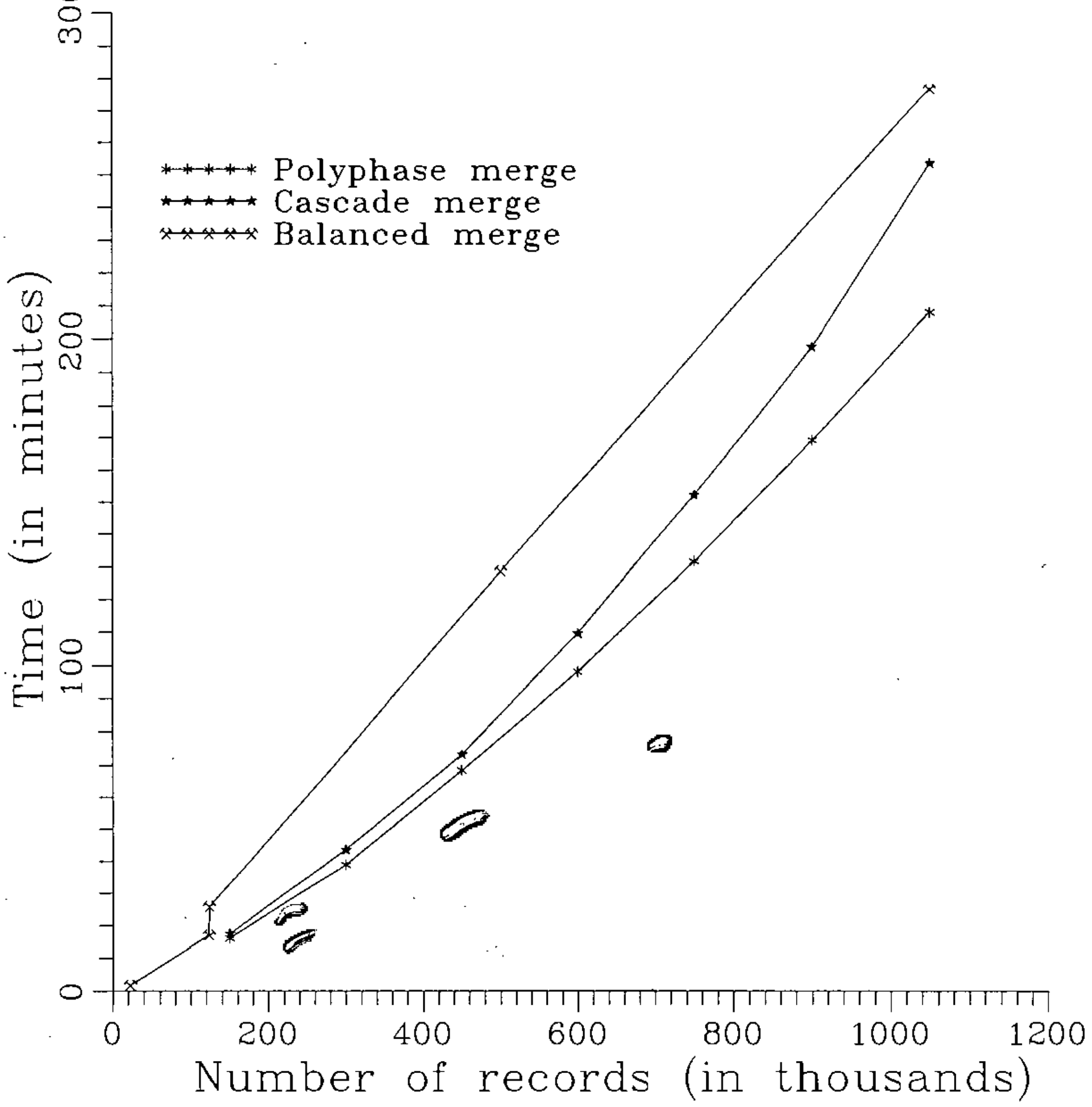
Graph 6.25 Cascade merge
(with F=11)



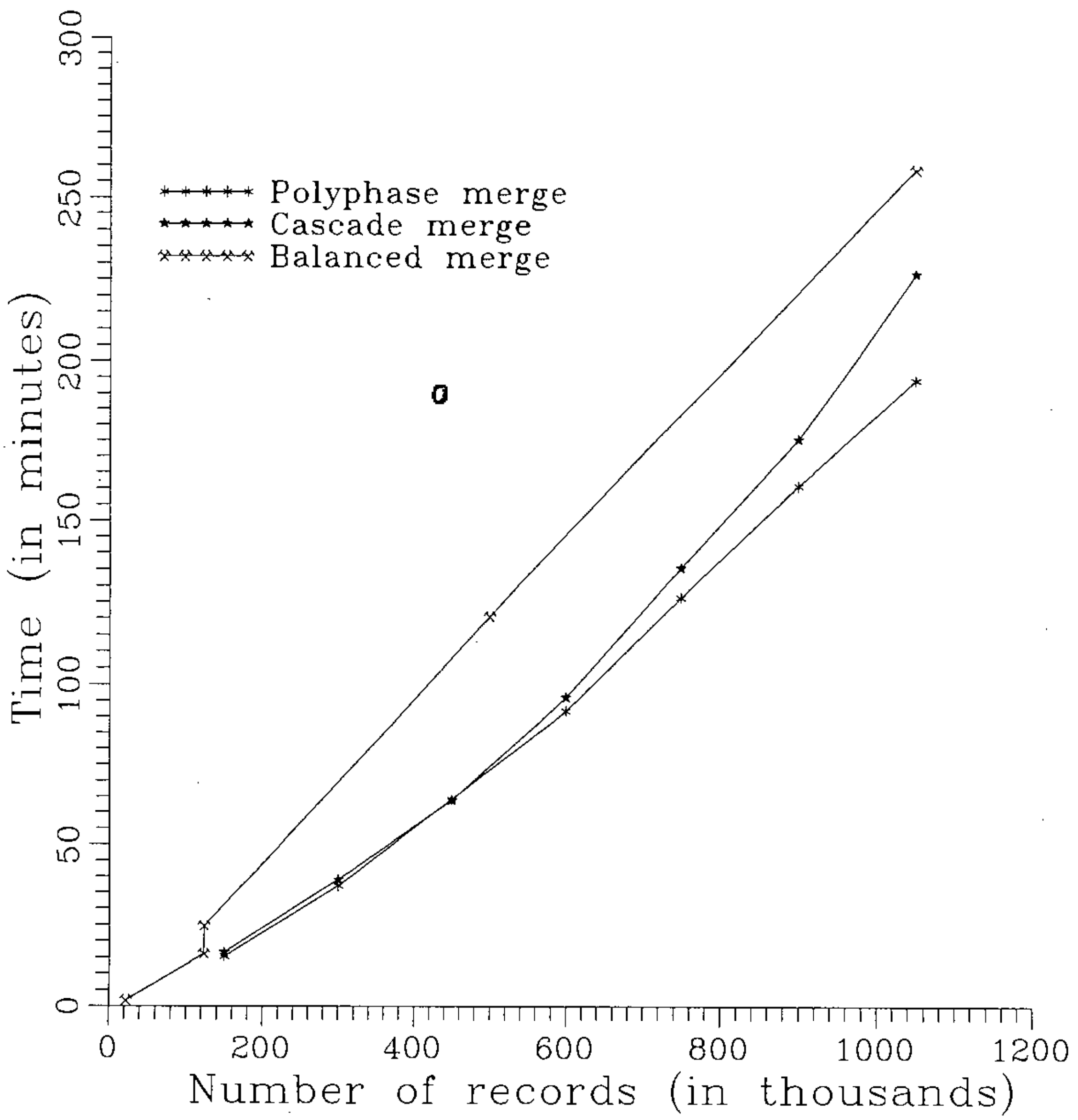
Graph 6.26 Cascade merge
(with F=12)



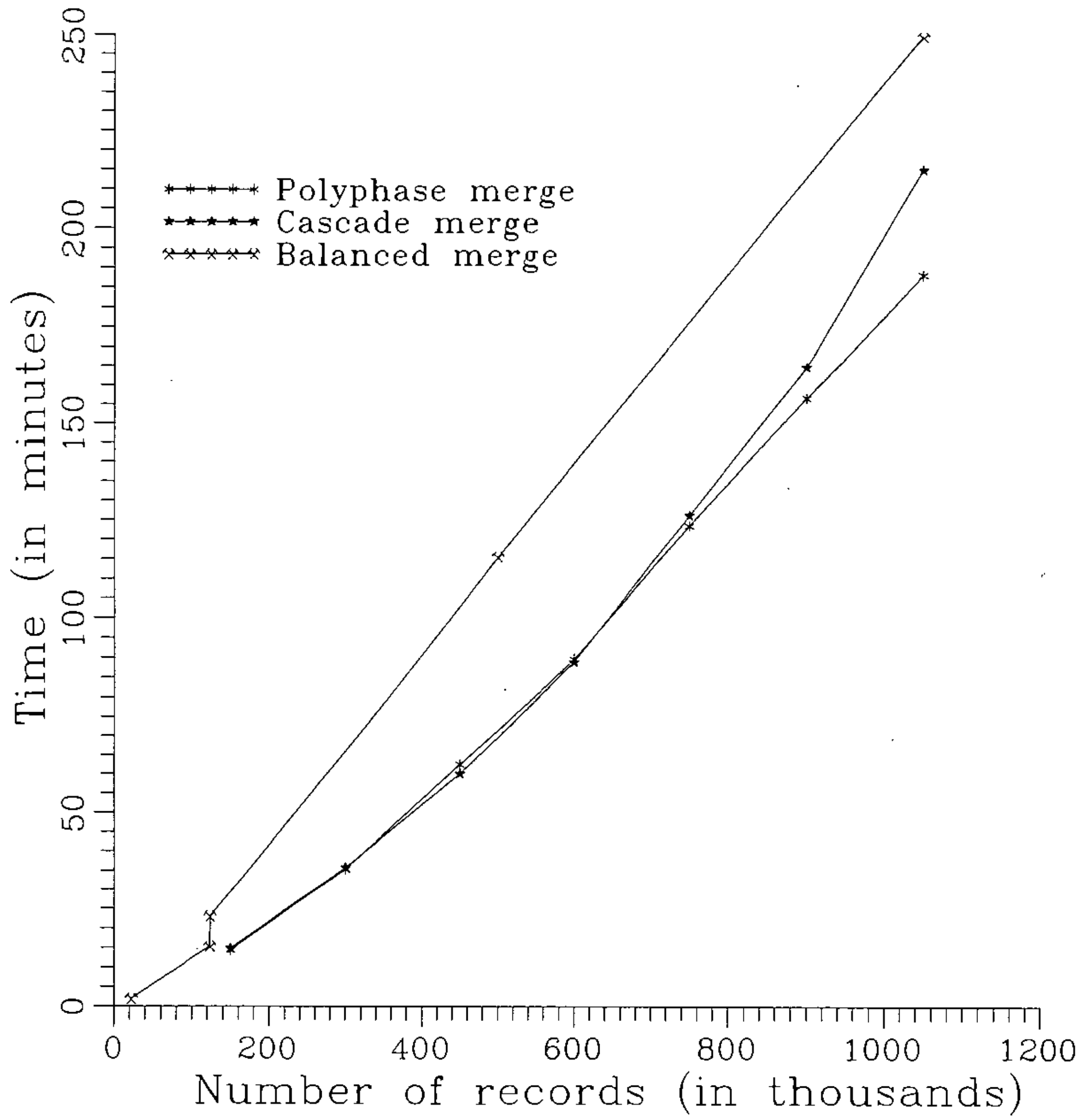
Graph 6.27 Cascade merge
(with F=13)



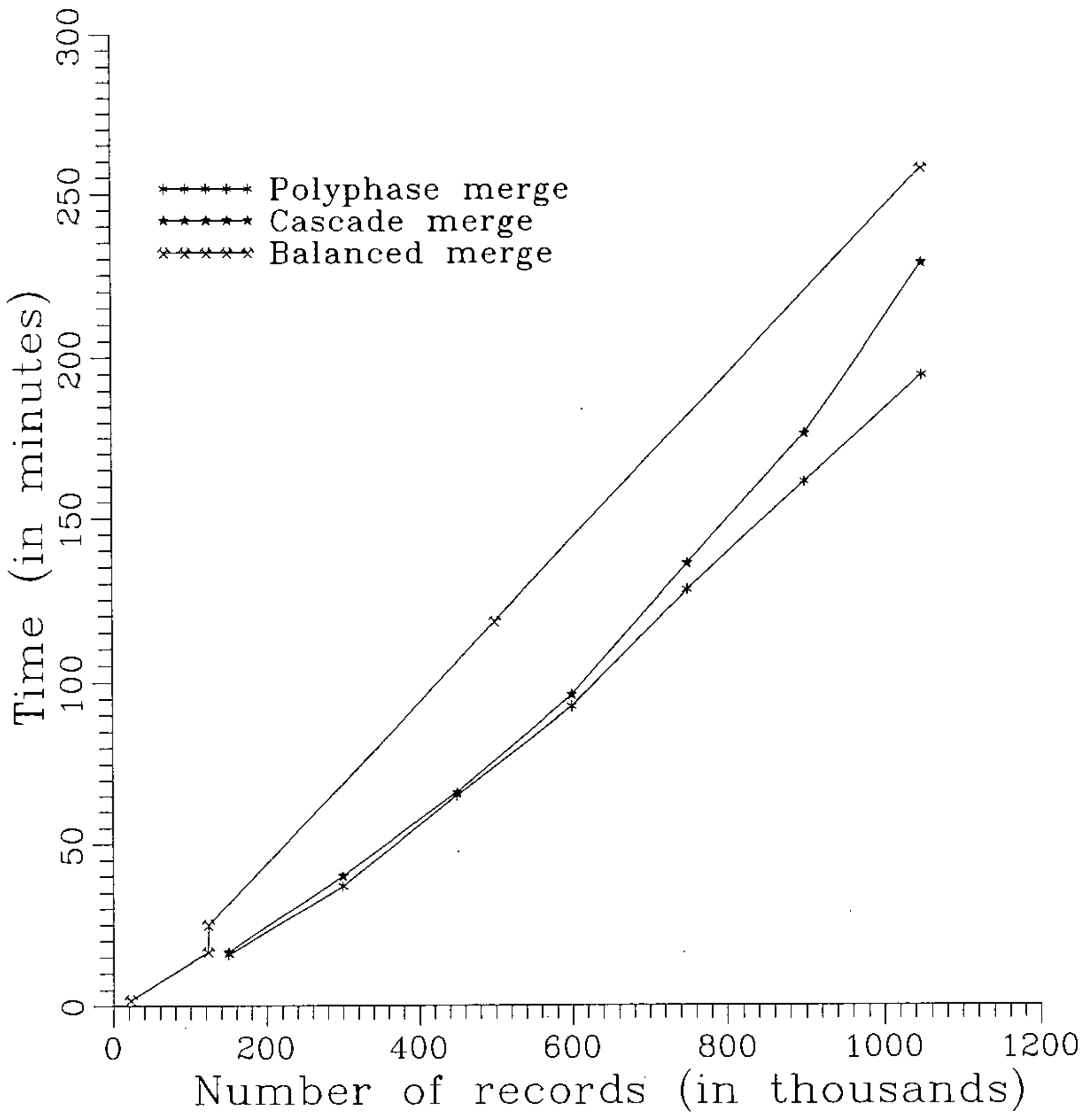
Graph 6.28 Comparison of minimum sorting times (BR=4)



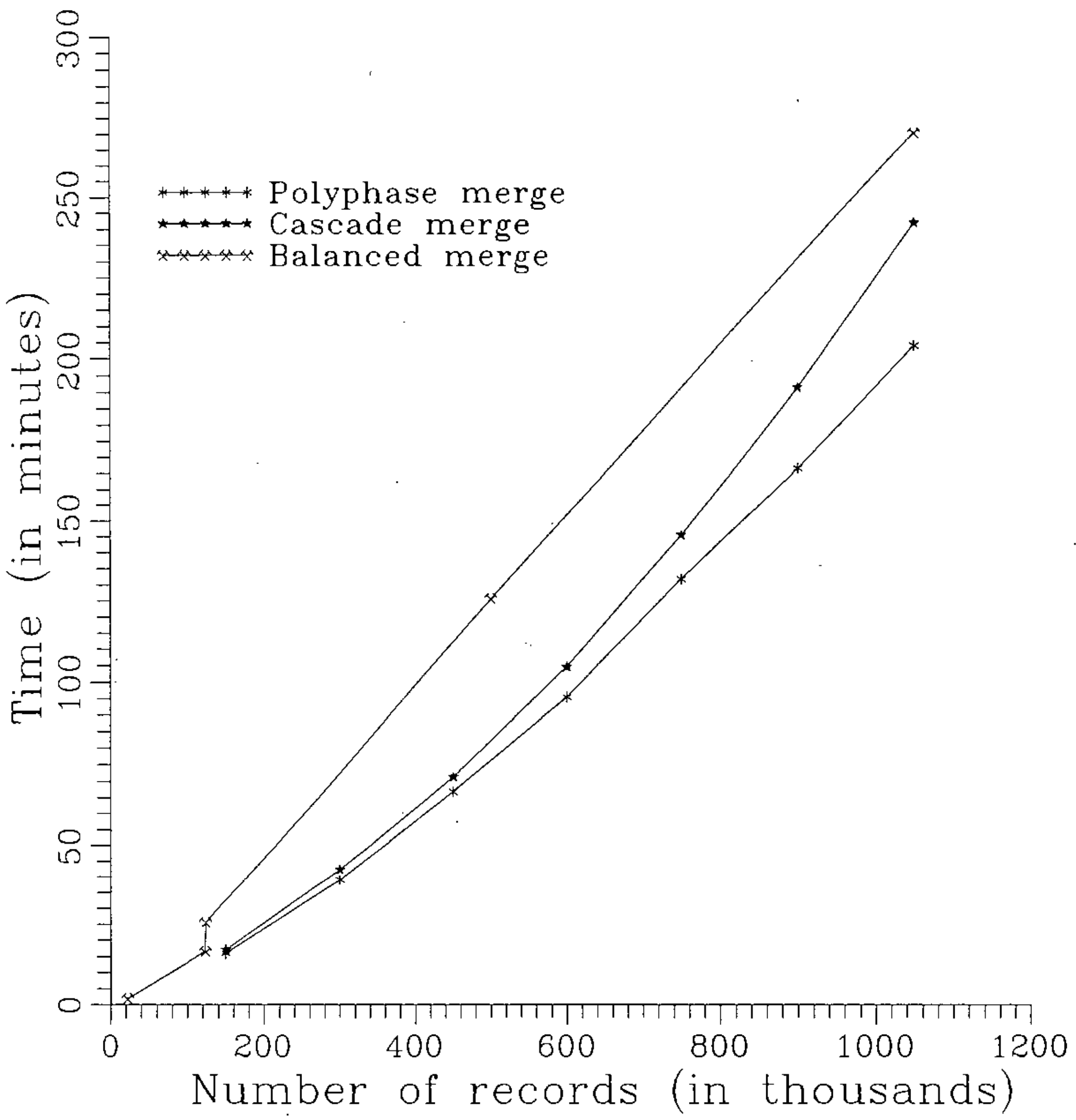
Graph 6.29 Comparison of minimum soritng times (BR=8)



Graph 6.30 Comparison of minimum sorting times (BR=10)



Graph 6.31 Comparison of minimum sorting times (BR=12)



Graph 6.32 Comparison of minimum sorting times (BR=16)

6.5 Comparison of Optimum Sorting Times:

Graphs 6.28-6.32 shows the comparison of balanced, polyphase and cascade merge sorting techniques for different values of *BR* and optimal values of *F*. It is apparent from these plots that the sorting time for polyphase and cascade merging are nearly same for small number of records, but for large number of records polyphase merge is more efficient than the cascade merge. Graph 6.30 is the most efficient among these five graphs, where value of *BR* is 10. The sorting time for balanced merging is not enough good for any number of records except for very small number of records.

CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS

7.1 Conclusions:

From the results given in chapter six, it is obvious that performance of various external sorting algorithms is sensitive to certain parameters. Graphs 6.1-6.5 show the response of balanced merge algorithm under different output to input buffer ratios (*BR*). For small number of records the sorting time for all values of *BR* is nearly the same. But as the number of records increases the sorting time becomes sensitive to the values of *BR*. When *BR* is 10, the balanced merge time is minimum.

For this algorithm, there is an abrupt change in sorting time when the number of records is around 125 thousand, because at these points, the number of times each run is to be handled is increased by one, where the number of initial runs distributed on files exceeds the order of merging. Thus we can say that these number of records, at which these abrupt changes occur, increases with the increase in the order of merging, as can be seen from the graphs 6.1-6.5.

We have seen previously that if *S* runs were produced during the internal sorting phase, and if $m^{k-1} < S \leq m^k$, then *m*-way balanced merge procedure makes exactly $k = \lceil \log_m S \rceil$ merging passes over all the data. This statement clarifies, why for a higher order of merging, more records are required to reach the point of this abrupt change of sorting time.

It can also be seen from graphs 6.1-6.5 that for small number of records the sorting time is minimum when F is 26. But it is interesting to note that as the number of records increases, the sorting time for 26 files deteriorates. Actually, for small number of records the order of merging is more important. As the amount of data handling is small, the number of seek is small, so higher order of merging requires small amount of time. But when amount of data handling increases, total seek time becomes more dominant, and therefore, minimization of the number of seeks becomes crucial for performance. It can be seen from the graphs 6.1-6.5 that for large number of records optimum value of F is 22.

It can also be seen that response time graph for balanced merge is piecewise linear since balanced merge works on the complete passes over the data. The constant of slope in different intervals increases as the number of records increases, which confirms the lower bound theory that no comparison based sorting algorithm of order less than $n\log_m n$ can exist, where n and m are positive integers.

Graphs 6.6-6.9 show the response of balanced merge algorithm under different number of files with changes in the values of BR. These graphs show that for small number of records, the sorting time is not sensitive to BR. But as the number of records increases, the sorting time gradually become more and more sensitive to BR. For any number of files the sorting time is minimum when BR is 10. For other values of BR sorting time is worse.

For balanced merging the variation in sorting time for different number of files for different values of *BR* is not constant. From graphs 6.1-6.5 it can be seen that this variation is minimum when *BR* is 8.

Graphs 6.10-6.14 show the time response of polyphase merge under different values of *BR* and *F*. In this case the minimum sorting time occurs when *F* is 13. For other values of *F* the sorting time increases. In this algorithm the sorting time is not piecewise linear and there is no abrupt change in the sorting time, since polyphase merge do not always make complete passes over the data and this algorithm makes partial passes over the data depending on the number of initial runs distributed on different files.

For smaller number of records the sorting time is not very sensitive to the values of *BR* and *F*, but as the number of records increases the sorting time becomes more sensitive to them, as in the case of balanced merge. Optimal value of *BR* for polyphase is 10 irrespective of the order of merging. For this algorithm the variation in sorting time for different values of *BR* and *F* is not constant. From graphs 6.10-6.14 it is apparent that this variation is minimum when *BR* is 8. Graphs 6.15-6.18 show that for small and large values of *F* compared to the optimal one (which is 13), the sorting time increases sharply for values of *BR* significantly different from the optimal one.

Graphs 6.19-6.23 show the variation of external sorting time with variation of *BR* and *F* for cascade merge. For all the combinations, the sorting time changes smoothly for consecutive number of initial runs. It is apparent from these graphs that the curves in every plot become divergent as the number of records increases. Thus, it can be concluded that difference in sorting time for different order of merging is more sensitive as the number of records increases.

For this merging technique also the minimum sorting time occurs when *BR* is 10. For other values of *BR* sorting time increases. Among different number of files the sorting time is mimimum when *F* is 12. From graphs 6.24-6.27 it is clear that the sorting time changes sharply also for cascade merge for values of *F* and *BR* significantly different from the corresponding optimal values.

Graphs 6.28-6.32 shows the comparison of balanced, polyphase and cascade merge techniques for different values of *BR* and optimal values of *F*. For small number of records the response time for polyphase and cascade merge techniques are nearly similar. But as the number of records increases polyphase merge becomes more efficient than cascade merge. Among graphs 6.28-6.32, graph 6.30 is the most efficient, where value of *BR* is 10.

7.2 Suggestions for Further Study:

If more than one physically separate hard disk units are available in a computer system to be used in external sorting then the total sorting time can be significantly reduced. But due to the unavailability of such computer system, we have restricted our study with a single disk unit. In the case of multiple disk units on the same computer system, a significant percent of total disk read, write and internal computation can be overlapped by maintaining appropriate buffering strategy. Therefore, one can attempt to design external sorting routines on a multi-disk computer system and study their responses with the variation of different sorting parameters.

The amount of random access memory has a profound influence on the external sorting time. Thus, one can undertake a task of designing external sorting routines for a computer systems with larger amount of random access memory. Also throughout this thesis we have confined our study to a data file of fixed record size. Attempt should be made to design external sorting routines to handle variable-length records and study their responses with the variation of different external sorting parameters.

BIBLIOGRAPHY

[1] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Co., Reading, Mass., 1983.

[2] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, Mass., 1974.

[3] Betz, B. K., *Unpublished Memorandum*, Minneapolis-Honeywell Regular Co. (1956).

[4] Betz, B. K., and W. C. Carter, *New Merge Sorting Techniques*, Proc. ACM, 14th National Conference (1953).

[5] Black, N. A., *Optimum Merging from Mass Storage*, Comm. ACM 13 (1957) pp. 745-749.

[6] Eckert, J. P., and J. N. Mauchly, *Progress Report on the EDVAC*, Moore School of Electrical Engineering, September 30, 1945.

[7] Friend, E. H., *Sorting on Electronic Computer Systems*, JACM 3, pp. 134-168, 1956.

[8] Gassner, B. J., *Sorting by Replacement Selection,* Comm. ACM 10 (1956), pp. 89-93.

[9] Gilstad, R. L., Proc. AFIPS Eastern Jt. Computer Conference 18 (1960), 134-148.

[10] Gilstad, R. L., *Read-Backward Polyphase Sorting*, ACM 6. (1951), pp. 220-223.

[11] Goetz, M. A., *Internal and Tape Sorting Using the Replacement Selection Techniques,* Comm. ACM 6 (1958), pp. 201-206.

[12] Horowitz, E., and S. Sahni, *Fundamentals of Computer Algorithms,* Galgotia Publications, New Delhi, 1990.

[13] Hubbard, G. V., *Some Characteristics of Sorting in Computing Systems Using Random-Access Storage Devices,* Comm. ACM 6 (1965) pp. 284-255.

[14] Johnsen, T. L., *Efficiency of the Polyphase Merge and a Related Methods*, BIT 6 (1966), pp. 129-143.

[15] Johnson, L. R., and R. D. Pratt, *An Introduction to the Complete UNIVAC II Sort*, UNIVAC Review, 1958.

[16] Klerer, M., and G. A. Korn, *Digital Computer User's Handbook,* McGraw-Hill Book Co., New York, 1967.

[17] Knuth, D. E., *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison-Wesley Publishing Co., Reading, Mass., 1973.

[18] Knuth, D. E., *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley Publishing Co., Reading, Mass., 1973.

[19] Knuth, D. E., *Computing Surveys* 2 (1970), 247-260.

[20] Reingold, E. M., and W. J. Hansen, *Data Structures*, Little, Brown and Company, New York, 1983.

[21] Sackmam, B., and T. Singer, *A Vector Model for Merge Sort Analysis*, presented at the ACM Sort Symposium, November, 1962.

[22] Schlegel, V., *El Progreso Mathemático* 4 (1894), 173-174.

[23] Seward, H. H., *Digital Computer Laboratory Report* (1954), 29-30.

[24] Shell, D. L., *Optimizing the Polyphase Sort*, Comm. ACM 14 (1961) pp. 713-719.

[25] Tremblay, J. E., and P. G. Sorenson, *An Introduction to Data Structures with Applications*, McGraw-Hill Book Co., New York, 1984.

[26] Truesdell, L. E., *The Development of Punch Card Tabulation*, U. S. Bureau of the Census, Washington, 1965.