Duplicate Record Elimination in Large Data Files

Dina Friedland
David J. DeWitt

Computer Sciences Technical Report #445

August 1981

Duplicate Record Elimination in Large Data Files

Dina Friedland
David J. DeWitt

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin

This research was partially supported by the National Science Foundation under grant MCS78-01721, the United States Army under contracts #DAAG29-79-C-0165 and #DAAG29-80-C-0041, and the Department of Energy under contract #DE-AC02-81ER10920.

ABSTRACT

This paper addresses the issue of duplicate elimination in large data files in which many occurrences of the same record may appear. A comprehensive cost analysis of the duplicate elimination operation is presented. This analysis is based on a combinatorial model developed for estimating the size of intermediate runs produced by a modified merge-sort procedure. The performance of this modified merge-sort procedure is demonstrated to be significantly superior to the standard duplicate elimination technique of sorting followed by a sequential pass to locate duplicate records. The results can also be used to provide critical input to a query optimizer in a relational database system.

1. Introduction

Files of data frequently contain duplicate entries and the decision whether and when to remove them must be made. For example, in relational database management systems (DBMSs) the semantics of the projection operator require that a relation be reduced to a vertical sub-relation and that any duplicates introduced as a side-effect be eliminated. In general, duplicate records may be introduced in a file either by performing an incorrect update operation or by being given a restricted view of the file. Identifying record fields such as names are often masked from an application program or from an output file before it is delivered to a user. In these cases, the amount of duplication can be significant and the cost of removing the duplicates substantial.

Duplicate elimination on a single processor is almost universally done by sorting. Because of the expense of sorting, relational DBMSs do not always eliminate duplicates when executing a projection. Rather, the duplicates are kept and carried along in subsequent operations. Only after the final operation in a query is performed, is the resultant relation sorted and duplicates eliminated.

The decision whether to eliminate duplicates or not (at various stages of the query execution) lies with the query optimizer subsystem of a DBMS. The purpose of a query optimizer is to schedule instructions from a query in a manner that will minimize the total query execution time. Typical factors affecting the decisions of an optimizer are: the types of operations in the

query, the availability of auxiliary information about files (such as indices), the size of the input files, and the expected size of the intermediate and output files. For relational DBMSs, the expected size of intermediate relations is often kept in the form of "selectivity factors" which reflect the observed values of previously executed operations on the same relation.

To our knowledge, existing query optimizers do not adequately schedule duplicate elimination operations. The problem lies in the fact that the literature does not contain a model for analyzing the cost of this operation. In this paper we propose a combinatorial model for the use in the analysis of algorithms for duplicate elimination. We contend that this model can serve as a useful tool for a query optimizer to decide when to eliminate duplicates. We also describe a modified sorting method for eliminating duplicates and use our model to show its superiority over the accepted method of first sorting a relation and then eliminating duplicates with a linear scan.

In Section 2, we discuss particular aspects of duplicate elimination in relational DBMSs. We present three methods for performing duplicate elimination in Section 3. The rest of the paper concentrates on a performance evaluation of one of these methods - a modified merge-sort procedure. In Section 4, we develop a combinatorial model that enables us to estimate the size of intermediate sorted runs produced by merging. In Section 5, we present some numerical evaluations based on this model. Our conclusions and suggestions for potential applications and extensions of our results are presented in Section 6.

2. Duplicate Elimination in a Relational DBMS

In relational database management systems, duplicate elimination constitutes a major part of the projection operation. Projecting a relation requires the execution of two distinct phases. First, the source relation must be reduced to a vertical subrelation by discarding all attributes other than the projection attributes. Then, duplicate tuples that may have been introduced as a result of the first operation must be removed in order to produce a proper relation. The first operation, forming the projected tuples, can either be performed in a linear scan of the relation or may be performed in combination with an operation preceding the projection, in which case the cost of this step would be negligible.

For example, consider the "supply" relation:

supplier-no	part-no	source	destination	qty

If we want to know which suppliers supply which parts in quantities larger than 1000 units, the relation must be restricted to (qty>1000) and projected on (supplier-no, part-no). Rather than creating a temporary relation for the restriction and then scanning it to discard the fields qty, source and destination, these fields should be eliminated as the restriction is executed. Since in the "supply" relation there may be many tuples with the same supplier-no and part-no attribute values, the result will be a list of non-unique two-attribute tuples. The second part of the projection consists of eliminating the duplicate tuples that

are introduced by the first phase.

The amount of duplication introduced by the projection depends on the nature of the projection attribute(s). If we project on a primary key, then no duplicates will be introduced.[1] On the other hand, if we discard a primary key and project on other attributes, a large amount of duplication may appear among the resulting tuples.

2.1. Implementation of the Projection Operation

Despite the fact that the duplicate elimination is an integral part of the projection operation as defined in Codd's relational algebra, relational database systems do not automatically implement it. Relations with duplicate tuples (which are not proper relations according to the relational algebra semantics) are in fact operated on by restrictions, joins, and other relational operators. Because duplicate elimination is expensive and because the tradeoffs between performing it or putting up with some inconsistent and redundant data are not clear, most database systems implementations (including relational systems such as System R [ASTR76] and INGRES [STON76]) postpone it to the very last stage of query processing. At that stage, the result tuples are sorted and duplicates are eliminated. If duplicate elimination is not systematically performed with every projection, the sets operated on by the relational operators such as selection and joins are not relations, and the database manage-

[1] A primary key is an attribute or a set of attributes which uniquely identifies a tuple.

ment system does not really conform to the relational model. Katz and Goodman [KATZ81] are currently investigating an extension of the relational model that allows for duplicates in relations. This extension deals with multisets that are viewed as a generalization of relations.[2] [KATZ81] shows how the relational algebra operators selection, join, and projection can be generalized to multisets, and introduces a new operator to explicitly specify duplicate elimination. Our study does not deal with the semantics of duplicate elimination. Its goal is to develop tools for implementing and evaluating the cost of this operation, whether or not it is done in the context of a relational database management system. In the case of a relational database, establishing an accurate cost formula for the projection can make improved query processing strategies possible. The first step in establishing such a formula is to evaluate the number of tuples in the projection.

2.2. Size of the Projected Relation

It is usually assumed that the database system dictionary can supply a reliable estimate of the size of a relation projected on any specified subset of attributes. This estimate may be based on an "a priori" knowledge about the number of distinct values that the projection attribute(s) can take on. It is reasonable to assume that this kind of information would be stored for all permanent relations, since it indicates the cardinality

---

[2] [KNUT73] defines a multiset as a set of non-unique elements. In Section 3, we define more specifically multisets that are relevant to our study.

of the attributes domains. However, the same information will neither be readily available for temporary relations created during intermediate stages of a query execution nor be inexpensive to compute. In the event that the size of a projected relation must be quickly estimated, a reasonable approximation may be achieved by assuming that a relation size is proportional to its tuple length [KERS80]. Let $|R|$ and $|R_p|$ denote the number of tuples in the source relation and in the projected relation, respectively. Then,

$$|R_p| = f_p * |R|$$

where the "projectivity" f_p equals the tuple length in bytes divided by the length of the projection attribute(s) in bytes. When indices on the projection attributes exist, the size of the projection can be estimated as

$$|R| * \prod_{a_j \in A} (1/s_j)$$

where A is the set of the projection attributes and s_j is the index selectivity for attribute a_j [YAO79].

Assuming that we have a reliable estimate for the size of the source relation and the size of its projection, the cost of the projection can be essentially defined as the cost of eliminating duplicates in a multiset of records, knowing the size of the multiset and the number of distinct records in it.

3. Algorithms for Duplicate Elimination

Using any sorting method with the entire record taken as the comparison key will bring identical records together. Since many

fast sorting algorithms are known, sorting appears to be a reasonable method for eliminating duplicate records. This section briefly describes three methods for duplicate elimination, two of which are based on sorting. The first method is an external 2-way merge-sort followed by a scan that removes the duplicates. The second method is a modified version of an external 2-way merge-sort, which gradually removes duplicates as they are encountered. The third method consists of using an auxiliary bit array that is obtained by hashing the record fields. This method was introduced by [BABB79], for efficiently realizing the relational join and projection operations. We discuss it for the sake of completeness, but we do not compare it with the other methods as it requires the use of specialized hardware for efficient operation.

We assume that the file resides on a mass storage device such as a magnetic disk (although for very large files, magnetic tape may be used as the storage media). It consists of fixed size records that are not unique. The amount of duplication is measured by the "duplication factor" f which indicates how many duplicates of each record appear in the file, on the average. The records are grouped into fixed size pages. An I/O operation transfers an entire page from disk storage to the processor's memory or from memory to disk. The file spans N pages, where N can be arbitrarily large, but only a few pages can fit in the processor's memory. The cost of processing a complex operation such as sorting or duplicate elimination can be measured in terms of page I/O's because I/O activity dominates computation time for

this kind of operations.[3]

3.1. The Traditional Method

For a large data file, duplicates are usually eliminated by performing an external merge-sort and then scanning the sorted file. Identical records are clustered together by the sort operation, therefore they are easily located and removed in a linear scan of the sorted file. We assume that the processor's memory size is about three pages and some working space. In this case, the file can be sorted using an external 2-way merge sort [KNUT73]. First each page is read into main memory, internally sorted, and written back to disk. Then main memory is partitioned into 2 input buffers and one output buffer, each large enough to hold a page. The external merge procedure starts with pairs of pages brought into memory that are then merged into sorted runs with a length of 2 pages. Each subsequent phase merges pairs of input runs produced by the previous phase into a sorted run twice as long as the input runs. Note that only one output buffer is required, since after one page of the output run has been produced, it can be written to disk allowing the merge operation to proceed. However, for the algorithm to be correctly executed, one must make sure that either consecutive pages of a run are written contiguously on disk, or that they are written at random locations but can be identified as consecutive pages of

---

[3] In fact, since for algorithms such as merge-sort the sequence of pages to be read is known in advance, pages can be prefetched enabling computation time to be completely overlapped with I/O time.

the same run by some address translation mechanism. With this provision made, the merge procedure can proceed and produce runs of size 4 pages, 8 pages, ... , N pages.[4] A 2-way merge procedure requires $\log_2 N$ phases with N page reads and N page writes at each phase (since the entire file is read and written at each phase).

After the file has been sorted, duplicate elimination is performed by reading sorted pages one at a time and copying them in a condensed form (i.e. without duplicates) to an output buffer. Again an output buffer is written to disk only after it has been filled, except for the last buffer which may not be filled. Thus the number of page I/O's required for this stage is:

$$N \text{ (reads)} + \text{ceil}(N/f) \text{ (writes)}$$

The total cost for duplicate elimination measured in terms of page I/O operations is:

$$2N\log_2 N + N + \text{ceil}(N/f)$$

3.2. The Modified Merge-sort Method

Most sorting methods can be adapted to eliminate duplicates gradually. [MUNR76] establishes a computational bound for the number of comparisons required to sort a multiset, when duplicates are discarded as they are encountered. Since we are dealing with large mass storage files, we are solely interested in working with an external sorting method. A two-way merge-sort

---

[4] For the sake of simplicity, we assume that N is a power of 2. However this is not required by the algorithm. A special delimiter may be used to signal the end of a run, so that a run may be shorter than 2^i pages at phase i.

procedure can be easily modified to perform a gradual elimination of duplicates. If 2 input runs are free of duplicates, then the output run produced by merging them should retain only one copy of each record that appears in both input runs (see Figure 1). Whenever two input tuples are compared and found to be identical, only one of them is written to the output run and the other is discarded (by simply advancing the appropriate pointer to the next tuple). The cost of the duplicate elimination process using this modified merge-sort is then determined by two factors: the number of phases and the size of the output runs produced at each phase.

3.2.1. Number of Phases

The number of phases required to sort a file with graduate duplicate elimination is the same regardless of the number of duplicate tuples. That is, $\log_2 n$ merge phases are required to sort a file of n records. This is true even in the extreme case when all the file records are identical. In this particular

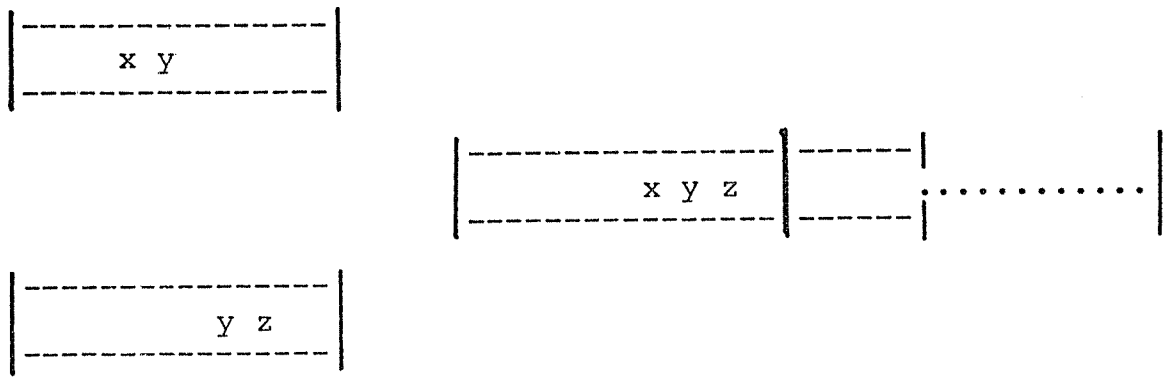

Figure 1 : Modified Merge

case, the run size is always 1 and every merge operation consists of collapsing a pair of identical elements into one element. By the same argument, if we start an external merge-sort with N internally sorted pages, the number of phases required is $\log_2 N$, whether or not duplicates are eliminated.

3.2.2. Size of Output Runs

Since we know the number of phases, the number of I/O operations required to execute the modified merge-sort will be completely determined if we have a method to measure how the runs grow as the modified merge-sort algorithm proceeds. When a two-way merge-sort is performed, the size of the runs grows by a factor of 2 at each step. However, if the merge procedure is modified in order to eliminate duplicates as they are encountered, the size of the runs does not grow according to this regular pattern. Suppose that the modified merge-sort procedure is executed without throwing away the duplicates as they are encountered. Instead, the duplicates would only be marked so that at any step of the algorithm they can be rapidly identified. Then, the size of an output run produced at phase i would still be 2^i but the number of distinct elements in the run would only be equal to the number of unmarked elements. Thus, it seems that a reasonable estimate for the average size of a run produced at the ith phase of a modified merge procedure is the expected number of distinct elements in a random subset of size 2^i of a multiset. In Section 4, we present a combinatorial model that provides us with such an estimate.

3.3. The Hashing Method

Essentially, this method works as follows. A bit array (M(I), I:1..k) with about as many entries as the number of distinct records is used to check for duplication. Rather than comparing the records themselves, a hash function provides a way to establish if 2 records are identical. Initially, all the entries in M are set to zero. Then, for each record read, all the fields are concatenated and the resulting string is hashed to provide the appropriate index, say I. If M(I)=0, the current record is written to the output list and M(I) is set to 1. This procedure ensures that no record is written twice to the output list, since identical records must hash to the same address. However, some records may be left out only because they "collide" with previously read records. Thus, each collision means loosing a "good" record. For this reason, it is very important to minimize the number of collisions. One way to achieve this goal is to increase the size of the bit array ([BABB79] recommends 4 times as many entries as the number of distinct records). A further improvement can be achieved by using several hash functions rather than a single one. The source file is scanned once for each hash function, and an output file (of non-colliding records) is created for each scan. Then the union of the output files is taken as the result file. The probability of missing records can be substantially reduced by using several statistically independent hash functions, since it is unlikely that different records will collide for each of these functions.

[BABB79] shows that the hashing method can be very fast,

when specialized hardware is used. The main problem remains the probability of missing any records. If there is not enough a priori knowledge on the data in order to determine that the expected number of missing records will be extremely small [5], or if no chance to miss a record can be taken, the cost of performing duplicate elimination with this method becomes extremely high (since it would require scanning the source file again to check if no record has been left out of the output file).

On a conventional computer, it seems that any duplicate elimination method not based on sorting would require an exhaustive comparison of all records and therefore lead to a slower performance. In the future, parallel processing may offer other alternatives. Different parallel architectures and algorithms are investigated for the elimination of duplicates in two recent studies ([BORA80], [GOOD80]). It may be the case that features such as broadcast of data to several processors will be the source for faster algorithms for duplicate elimination. The results that have been in this area are not conclusive and will not be presented as they are beyond the scope of this paper.

4. A Combinatorial Model for Duplicate Elimination

In this section, we consider the problem of finding all the distinct elements in a multiset. A multiset is a set $\{x_1, x_2, \ldots, x_n\}$ of not necessarily distinct elements. We assume that any two of these elements can be compared yielding $x_i > x_j$, $x_i = x_j$

[5] For 50K distinct records, [BABB79] estimates that a bit array of 1M bits and 4 scans can reduce the error rate to 10^{-11}.

or $x_i<x_j$. The x_i's may be real numbers or alphanumeric strings that can be compared according to the lexicographic order. Or they may be records with multiple alphanumeric fields, with one (or a subset of fields) used as the comparison key. The elements in the multiset are duplicated according to some distribution f_1, f_2, ..., f_m. That is there are f_1 elements with a "value" v_1, f_2 elements with a value v_2, ... , f_m elements with a value v_m, and $\sum f_i = n$. When n is large and the values are uniformly distributed, we may assume that

$$f_1 = f_2 = \ldots = f_m = f$$

and therefore

$$n = f*m$$

In this case, we define f as the "duplication factor" of the multiset.

4.1. Combinations of a Multiset

Consider the following problem. Suppose we have a multiset of n elements with a duplication factor of f and m distinct elements so that n=f*m. Let k be any integer less or equal to n. How many distinct combinations of k elements can be formed where all the m distinct elements appear at least once? This number is denoted by $c_{fm}(k)$. We consider combinations rather than arrangements because we are interested in the identity of the elements in a subset but not in their ordering within the subset. The notation $\binom{p}{q}$ is used to represent a q-combination of p distinct elements, with the convention $\binom{p}{q}=0$ for q>p.

Lemma 1:

$$c_{fm}(k) = \binom{f*m}{k} - \binom{m}{1}\binom{f(m-1)}{k} + \binom{m}{2}\binom{f(m-2)}{k} - \ldots + (-1)^{m-1}\binom{m}{m-1}\binom{f}{k}$$

Proof: The intuitive meaning of lemma 1 is that the number of combinations with exactly m distinct elements is equal to the number of combinations with at most m distinct elements minus the number of combinations with m-1, m-2, ... ,1 distinct elements. To prove the lemma, we express the total number of combinations of size k in terms of the number of combinations of size k with m distinct elements, of the number of combinations of size k with m-1 distinct elements, etc.

$$\binom{f*m}{k} = c_{fm}(k) + \binom{m}{1} c_{f(m-1)}(k) + \binom{m}{2} c_{f(m-2)}(k) + \ldots$$

$$\binom{f(m-1)}{k} = c_{f(m-1)}(k) + \binom{m-1}{1} c_{f(m-2)}(k) + \binom{m-1}{2} c_{f(m-3)}(k) + \ldots$$

...

By combining these expressions to form the right-hand side sum in lemma 1, all the c(k) cancel each other except for $c_{fm}(k)$. Notice also that k might be greater than f(m-i) for some i>0 which according to our notation, would imply that some of the terms in the right hand side sum become zero.

4.2. The Average Number of Distinct Elements

Starting with a multiset that has m distinct values and a duplication factor f, there are $\binom{fm}{k}$ subsets of size k. Thus, the probability that a random subset of size k contains exactly d specific distinct elements ($d \leq m$) is equal to:

$$c_{fd}(k) / \binom{f*m}{k}$$

The expected number of distinct elements in a random subset of

size k can be computed by averaging over all possible values of d. The lowest possible value of d is $\lceil k/f \rceil$ since d distinct elements cannot generate a set larger than f*d. On the other hand, there can be at most m distinct values since we are considering subsets of a multiset with m distinct values.

Lemma 2: The expected number of distinct elements in a subset of size k is:

$$av_{fm}(k) = \{ \sum_{d=\lceil k/f \rceil}^{\min(k,m)} d*\binom{m}{d}*c_{fd}(k) \} \; / \; \binom{f*m}{k}$$

Lemma 3: For i>1

$$(m-i) - (m-i+1)*\binom{i}{1} + (m-i+2)*\binom{i}{2} - \ldots = 0$$

Proof: Let us consider the product $x^{m-i}(1-x)^i$. By expanding the second factor, we have

$$x^{m-i}(1-x)^i = x^{m-i} - \binom{i}{1}x^{m-i+1} + \binom{i}{2}x^{m-i+2} - \ldots$$

and

$$\frac{d}{dx}\{x^{m-i}(1-x)^i\} = (m-i)x^{m-i-1} - (m-i+1)\binom{i}{1}x^{m-i} + (m-i+2)\binom{i}{2}x^{m-i+1} - \ldots$$

For x=1 and i>1 this derivative is equal to zero.

Lemma 4: For k≥m

$$\sum_{d=\lceil k/f \rceil}^{k} d\binom{m}{d}*c_{fd}(k) = m\binom{f*m}{k} - m\binom{f(m-1)}{k}$$

Proof: Let

$$S= \sum_{d=\lceil k/f \rceil}^{m} d*\binom{m}{d}*c_{fd}(k)$$

Since for each k, $c_{fd}(k)$ is a linear combination of terms of the form $\binom{f(m-i)}{k}$ and since the upper bound for d in S is m, S may be rewritten backwards as a linear combination of terms of the form $\binom{f(m-j)}{k}$, j=0,1,... Then the coefficient of $\binom{fm}{k}$ is $m\binom{m}{m}$. The coefficient of $\binom{f(m-1)}{k}$ is $(m-1)\binom{m}{1} - m^2 = -m$.
For i>1, the coefficient of $\binom{f(m-i)}{d}$ is:

$$(m-i) - (m-i+1)\binom{i}{1}+(m-i+2)\binom{i}{2} - \ldots$$

which is null by lemma 3.

Theorem 1: If $k \geq m$

$$av_{fm}(k) = m - m*\{ \binom{f(m-1)}{k} / \binom{f*m}{k} \}$$

It is interesting to notice that when f is large (that is the duplication factor is high), $av_{fm}(k)$ becomes a function of m and k only. This can be proven as follows:

$$\binom{f(m-1)}{k} / \binom{f*m}{k} = \prod_{i=1}^{k} (f*m-f-k+i)/(f*m-k+i)$$

which is approximately equal to $(m-1/m)^k$ for large f.
Therefore, $av_{fm}(k) \simeq m(1 - (m-1/m)^k)$ for large m.

This result confirms the intuitive idea that the number of distinct elements in a random subset of a large multiset depends

only on the size of the subset and on the number of distinct values in the multiset.

For smaller duplication factors, as we keep f and m constant, $av_{fm}(k)$ increases monotonically as a function of k, until for some $k=k_0$ it reaches the value m. From then it remains constant as k increases from k_0 to f*m. Figure 2 displays the function $av_{fm}(k)$ for f*m=32768 and for three different values of f (8, 16, 32). The value k_0 is of particular interest. It indicates how large a random subset of a multiset must be in order to contain at least one copy of all the distinct elements.

It is interesting to note that the problem we address here is somehow related to the classical "occupancy problem" of the Bose-Einstein statistics [FELL68]. The n elements of a multiset may be identified with n particles and the m distinct elements in the multiset may be pictured by m distinct cells in which the particles can fall. All the duplicates of one element are then represented by the set of particles that fell into a single cell. Assuming that all elements are equally duplicated, i.e. n=f m and that there are f copies of each element, is equivalent to assuming that all cells end up containing the same number of particles. In fact, this is only one of many possible states and a more accurate modeling of duplication should consider the number of particles in one cell as a random variable with mean equal to f=n/m. It is known that for large n and m, the distribution of this variable is a Poisson distribution with mean n/m [FELL68]. Estimating the number of distinct records in a multiset is similar to estimating the number of occupied cells. Here again, it

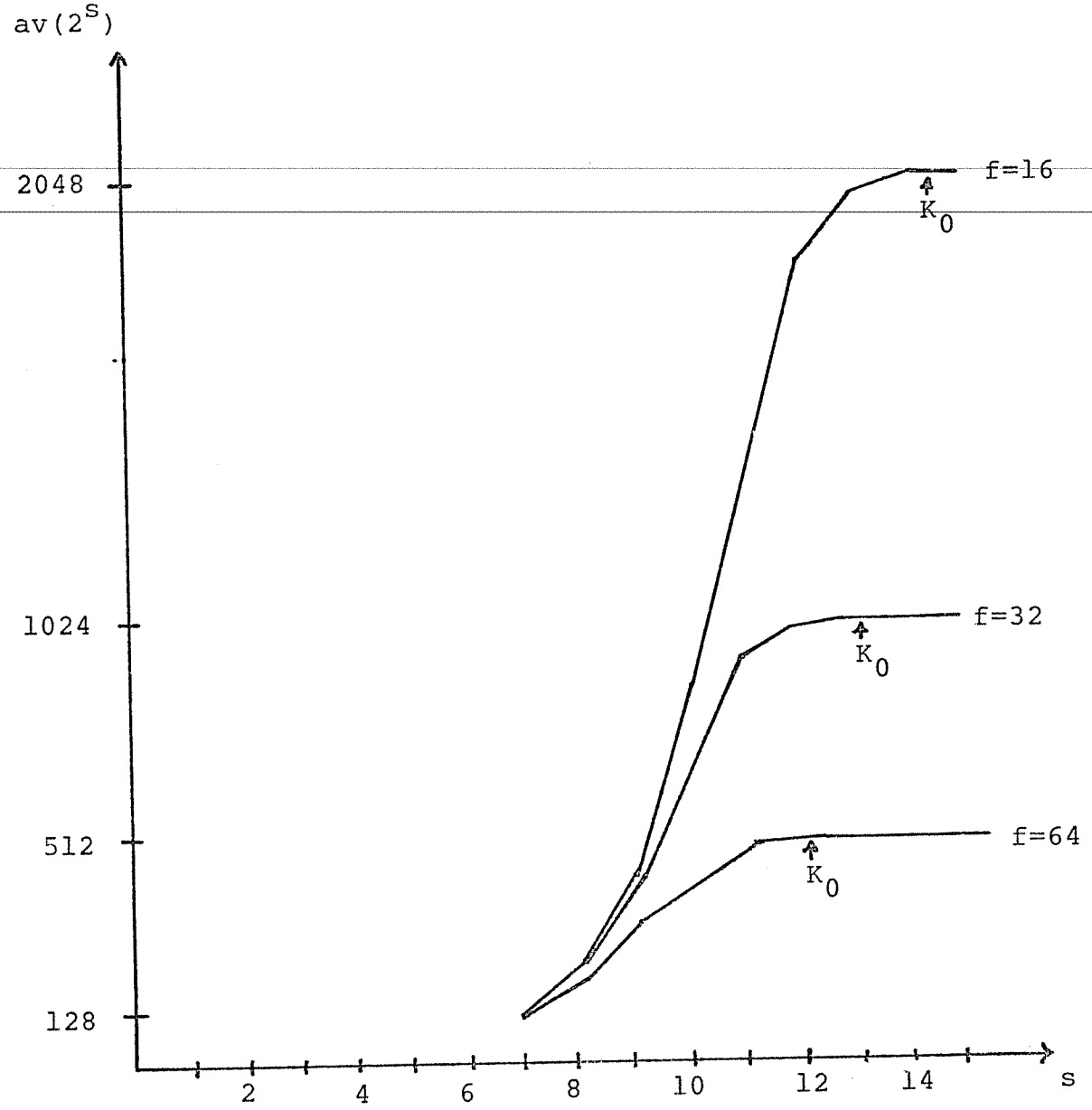

Figure 2

Number of Distinct Records in Successive Runs

has been shown that the number of empty cells (that is m-the number of occupied cells) is a random variable with a Poisson distribution. Thus, for very large duplication factors, the probability of finding a specified number of distinct records in a subset of records can be estimated using the tools developed for the classical occupancy problem.

5. Cost Analysis of Duplicate Elimination

As discussed in Section 3, the cost of a modified merge sort is completely determined if the size of intermediate output runs can be estimated. In this section, we evaluate this cost and compare it to the cost of a traditional merge-sort. We assume that the source file consists of n non-unique records, with a duplication factor f. The modified merge algorithm will produce an output file of m=n/f distinct records. Both the source file and the output file records are grouped into pages and all pages, except possibly one, contain the same number of records ("page size" below). The cost of duplicate elimination is measured by the number of pages read and written, assuming that the main memory can fit no more than two page input buffers and one page output buffer. When an intermediate run is produced, records are also grouped into full pages before any output buffer is written out. Only the last page of a run may not be full. Therefore, the number of pages written when an output run is produced will be:

ceil(number of records in a run / page size).

We assume that the external merge procedure starts with

internally sorted pages, and that each of these pages is free of duplicates. This assumption is legitimate if the records are uniformly distributed across pages in the source file, and if the number of distinct records is much larger that the number of records that a single page can hold.

If there were no duplicates, the number of records in each input run read at phase i would be 2^{i-1} times the page size, since the merge procedure is started with runs that are one page long. Similarly, the number of records in an output run produced at phase i would be 2^i times the page size. Suppose the page size (measured in number of records that a page can hold) is p. Therefore, when duplicates are gradually eliminated, the expected number of records in an input run read at phase i is equal to $av_{fm}(2^{i-1}p)$, and the expected number of records in an output run produced at phase i is equal to $av_{fm}(2^ip)$, using the notation defined in Section 4. On the other hand, the number of runs produced at phase i is $n/2^ip$ (where n=fm is the number of records in the source file). Therefore, the number of pages read at phase i is

$$2 * \text{ceil}[\ av_{fm}(2^{i-1}p)/p] * n/2^i*p$$

and the number of pages written is

$$\text{ceil}[\ av(2^ip)/p] * n/2^i*p$$

Using these formulae, we have summarized in Figure 3 the total number of page I/O's required to eliminate duplicates from a file of 131072 records. With 128 records per page, this file spans

1024 disk pages. We have considered various duplication factors from 2 (i.e. there are 2 copies of each record) to 64 (64 copies of each record) The results indicate that a modified merge sort requires substantially less page I/O's than a standard merge sort, especially when the amount of duplication is large. When a standard merge sort is used to eliminate duplicates, it must be followed by a linear scan of the sorted file. Therefore, we also show this augmented cost in the rightmost column of the table in Figure 3.

A further reduction of page I/O's can be achieved by terminating the modified merge procedure as soon as the runs have achieved the result file size. When this happen, all the output runs will essentially be identical and each of them contains all the distinct records. As we observed in Section 4, this may occur a few phases before the final phase, e.g. at phase number $(\log_2 N)-i$, for some $i>1$ (N being the number of pages spanned by

f	modified merge	standard merge	std merge+scan
2	19008	20480	22046
4	17400	20480	21760
8	15664	20480	21632
16	13840	20480	21568
32	12000	20480	21536
64	10192	20480	21520

Figure 3 : Cost of duplicate elimination

the source file). When this phase is reached, a single run may be taken as the result file since it contains all the distinct records and no duplicates. Therefore, the elimination process is complete and one may save the additional I/O operations which serve only to collapse together several identical runs. Figure 4 shows the I/O cost of this "shortened" procedure, compared to the cost of a complete modified merge sort: For this file size, the savings in page I/O's can reach up to 7% of the total cost. For a smaller file size (32K records) and a small duplication factor, we have observed an improvement of the order of 10%. When varying the file size and the duplication factor, we have observed that the improvement was greater for very small or very large duplication factors, while it was smaller in the mid-range values (e.g f=8 and f=16 in Figure 4).

Since we have only estimated the expected size of the runs,

f	modified merge	shorter merge	improvement
2	19008	17728	1280
4	17400	16264	1136
8	15664	15280	384
16	13840	13264	576
32	12000	11328	672
64	10192	9472	720

Figure 4 : Early termination of modified merge

our numbers are only accurate provided that the actual run size does not fall too far away from that average. This will certainly not happen if the records are uniformly distributed in the source file. Finally, it is very important to note that if there is no a priori information about the number of duplicate records present in the source file, the modified sort-merge can still be used to eliminate duplicates and the procedure can be terminated as soon as the run size stop growing. When this condition is verified, a single run can be taken as the result file, although a precise statement about the probability that such a run indeed contains all the distinct records requires a more elaborate statistical model than the one we have used.

6. Conclusions

A model for evaluating the cost of duplicate elimination has been presented. We have shown how, by modifying a 2-way merge-sort, duplicates can be gradually eliminated from a large file at a cost which is substantially less that the cost of sorting. Accurate formulas have been established for the number of disk transfers required to eliminate duplicates from a mass storage file. These formulas can be used whenever there exists an a priori estimate for the amount of duplication present in the file. When such an estimate is not available, it is argued that the modified merge-sort method should still be used. In this case, a condition for testing that all duplicates have been removed is described.

We have based our analysis on a combinatorial model that

characterizes random subsets of multisets. Only a particular category of multisets has been considered, where all elements have the same order. Thus, our results are only accurate for files with a uniform duplication factor (i.e. each record is replicated the same number of times in the entire file). Refining our analysis would require the use of more sophisticated statistical tools to model more accurately the distribution of duplicates. However, for files with a large number of records and with many duplicates, our model would provide a reasonable approximation.

In addition to generalizing our cost evaluation model to the case where the records are not uniformly duplicated, it would be of interest to model the cost of duplicate elimination on parallel computers. As mentioned in Section 3, algorithms for performing this operation have already been developed ([BORA80], [GOOD80]). However, measuring the execution time of these algorithms is difficult because of the additional complexity involved. We hope that the tools developed in this study for modeling duplicate elimination on a conventional computer will be of some use for a parallel computer as well. In particular, we plan on measuring the effect of duplicate elimination on the performance of the parallel merge-sort algorithm presented in [BORA80].

The motivation for our work was the need for a method to evaluate the cost of duplicate elimination. To our knowledge most query optimizers in relational DBMSs schedule a duplicate elimination operation in an ad hoc manner. The model developed

in this paper can serve as a tool to be used by a query optimizer in estimating the cost of eliminating duplicates from a relation. Using this estimated cost an optimizer can schedule operations so that the total execution time of the query is minimized.

7. Acknowledgements

We gratefully acknowledge the comments and helpful suggestions made by Haran Boral.

REFERENCES

[ASTR76] Astrahan, M., et al., System-R: A Relational Approach to Database Management, ACM TODS, Vol. 1, No. 2, June 1976.

[BABB79] Babb, E., Implementing a Relational Database by Means of Specialized Hardware, ACM TODS, Vol. 4, No. 1, March 1979.

[BORA80] H. Boral, D. J. Dewitt, D. Friedland and W. K. Wilkinson, Parallel Algorithms for the Execution of Relational Database Operations, University of Wisconsin Computer Science Technical Report #402.

[FELL68] Feller W., "An Introduction to Probability and its Applications," Wiley 1968.

[GOOD80] J. R. Goodman and A. M. Despain, A Study of the Interconnection of Multiple Processors in a Database Environment, Proceedings of the 1980 Conference on Parallel Processing.

[KATZ81] R. H. Katz and N. Goodman, On the Semantics of Duplicate Elimination, Computer Corporation of America Technical Report.

[KERS80] L. Kerschberg, P.D. Ting, and S.B. Yao, Query Optimization in Star Computed Networks. Bell Laboratories Database Research Report No. 2, March 1980.

[KNUT73] D. E. Knuth, "The Art of Computer Programming, " Vol. 3, Addison-Wesley, 1973.

[MUNR76] I. Munro and P. M. Spira, Sorting and Searching in Multisets, Siam J. Comput. Vol. 5, No. 1, March 1976.

[STON76] Stonebraker, M., The Design and Implementation of

Ingres, ACM TODS, Vol. 1, No. 3, September 1976.

[YAO79] Yao , B. , Optimization of Query Evaluation Algorithms, ACM TODS, Vol. 4, No. 2, June 1979.