JOIN DURING MERGE: AN IMPROVED SORT BASED ALGORITHM

M. NEGRI and G. PELAGATTI

Dipartimento di Elettronica, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy

1. Introduction

A join operation consists in selecting from the set of all pairs of records of two files F_1 and F_2 those pairs which possess some matching property. The matching property can be expressed as $A_1 \theta A_2$, where A_1 and A_2 are two attributes of the records of F_1 and F_2 respectively, and θ is a comparison operator. Without loss of generality we will assume that θ is the equality operator (equijoin).

Because of the importance of the join operation in database applications, several algorithms have been proposed for performing it efficiently. These algorithms can be classified at a first level into two main classes: those which are based on the existence of auxiliary data structures, like indexes and inter-file links, and those which do not use any auxiliary data structures. This paper deals with the latter class (i.e., it assumes that auxiliary data structures are not available for the join).

The algorithms for performing the join in absence of auxiliary data structures can be further partitioned into two main subclasses: the 'nested' algorithms and the 'sort-based' algorithm.

The nested algorithms consist in choosing one file, called external file, and comparing each record of it with all the records of the other file, called internal file. The main disadvantage of these algorithms is that the internal file is completely scanned several times. An optimization of this class of algorithms consists in reading large portions of the external file into a buffer, so that the number of internal file scans is reduced [1].

The sort-based algorithm consists in first sorting both files on the join attribute and then advancing in parallel on both sorted files, building the join. In this algorithm, the cost of performing two complete sort operations must be added to the cost of the actual join operation; however, the latter is clearly much simpler than the nested join.

The relative efficiency of the sort-based algorithm with respect to the nested algorithms depends on several parameters, the most important of which is the distribution of values of the join attributes: if the values of the join attributes are all different, so that they determine a total ordering of each file, then the maximum advantage is taken from the sort phase; on the opposite, if all records have the same value of the join attributes, then the sort operation is useless. Another relevant parameter is the relative size of the two files with respect to the available main memory buffer.

For the purpose of this paper, the comparison between nested and sort-based algorithms is not relevant; it is sufficient to know that there are several cases in which the sort-based algorithm is more convenient.

This paper describes and analyzes an improvement of the classical sort-based algorithm. The proposed improvement consists in avoiding to completely sort both files; the join is performed

when the files have been partially sorted into subfiles. The new algorithm is called 'join-during-merge', because the join operation is performed instead of the last merge pass of the sort operation. In fact, two different algorithms which are based on the above idea are proposed and analysed, called one-way and two-way join-during-merge.

The following assumptions are made in the remainder of this paper:

(a) A paged-memory environment is assumed and the number of page fetches from the disk is considered to be the primary cost measure of a join operation (this assumption is also made in [1,2,3]).

(b) The values of the join attributes in each file are all different, so that they determine a total ordering of the file; this assumption is meaningful, because the sort-based algorithm is more convenient in this case.

(c) A constraint exists on the available main memory buffer, otherwise it would be possible to read the whole file into main memory; in this case, the nature of the join problem would change completely. This constraint is indicated by the maximum available buffer size B_{MAX} (in pages).

The join-during-merge algorithm is compared with the classical sort-based algorithm with respect to the required page fetches and it is shown that it behaves always better. Therefore, this paper suggests to use the join-during-merge algorithm instead of the sort-based algorithm in all cases where the sort-based algorithm was considered convenient.

2. Sort-based join (SBJ)

Let us first recall the main characteristics of the classical sort-based join. A sort-based join is performed as shown in Fig. 1: both files are first completely sorted by performing a sequence of Sort/Merge (S/M) passes and then the two sorted files are joined in one pass (SJ). The Sort/Merge (S/M) passes are described in the literature on external sorting (see, for instance, [4]), while the Sorted Join (SJ) pass is described in [3].

For understanding this paper it is sufficient to consider the following aspects of a Sort/Merge algorithm for sorting a file: the file is first decomposed into many subfiles that are sufficiently small to fit into the available main memory buffer; the first S/M pass reads each subfile into the buffer and sorts it (internal sort); subsequent S/M passes merge the sorted subfiles until all records have been merged into one ordered file, which is the final sorted file.

The size B_i (in pages) of the main memory buffer which is required for sorting a file F_i using n_i S/M passes, is

$$B_i = \left\lceil \sqrt[n_i]{P_i} \right\rceil, \tag{1}$$

where P_i is the number of pages of the file F_i. This formula can be explained as follows (see [3] and [5] for an extensive discussion): the Sort/Merge algorithm performs at each pass a B-way merge, since it has only B pages of main memory availa-

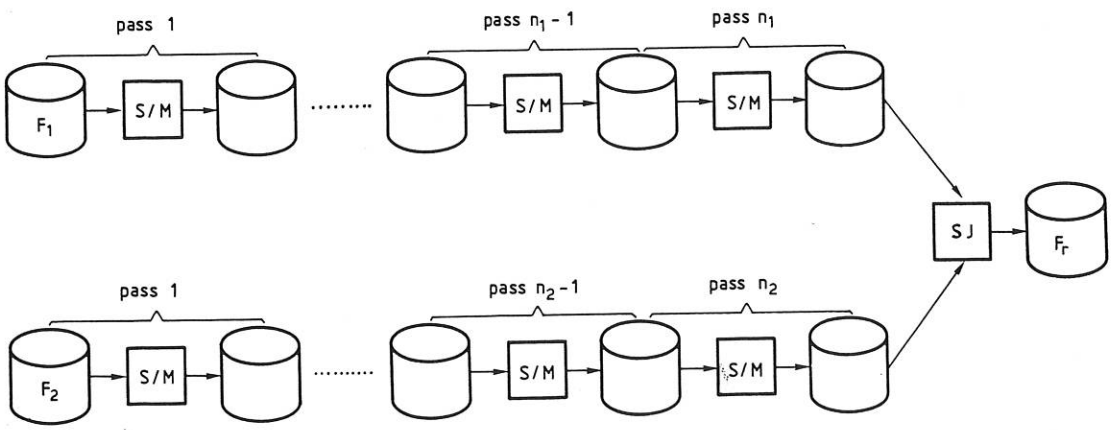

Fig. 1.

ble. Therefore, with n passes it can sort a file of at most B^n pages.

Since the last pass SJ can be performed using a buffer with size $B \geqslant 2$ independently from the size of the two sorted input files, the maximum buffer size which is required for performing a join using n_1 and n_2 S/M passes on the files F_1 and F_2 respectively is expressed by

$$B_{SBJ}(P_1, P_2, n_1, n_2) = \max\left\{\left\lceil \sqrt[n_1]{P_1} \right\rceil, \left\lceil \sqrt[n_2]{P_2} \right\rceil\right\}. \quad (2)$$

3. One-way join-during-merge (JM1)

The one-way join-during-merge (JM1) algorithm consists in completely sorting one file and only partially sorting the other file and then performing the join. We will assume that file F_2 is completely sorted while file F_1 is only partially sorted. Algorithm JM1 works as follows:

Step 1. Sort one file (say F_2) in join column order.

Step 2. Perform n_1 sort/merge (S/M) passes on the other file (F_1), so that a partially sorted file F_1' is obtained consisting of m subfiles $F_{11}', \ldots, F_{1m}'$ (the condition for determining m and n_1 depends on the maximum available buffer size B_{MAX} and will be stated in the sequel).

Step 3. Read the first page of F_2 and of each subfile $F_{11}', \ldots, F_{1m}'$ into the buffer.

Step 4. Repeat until F_2 and F_1' have been completely scanned:

(a) Join the records of the page of F_2 contained in the buffer with the matching records of the pages of the subfiles F_{1i}', $1 \leqslant i \leqslant m$ contained in the buffer.

(b) Write the joined records to the output file.

(c) For each input page in the buffer which is completely processed, read a new page from the associated file F_2 or subfile F_{1i}', $1 \leqslant i \leqslant m$.

Consider now the problem of determining n_1 and m at Step 2. These must be such that the following Steps 3 and 4 can be performed with the available buffer B_{MAX}. Let $B_1(P_1, n_1)$ be the size of the buffer which is needed for performing the n_1 S/M passes on file F_1 and Steps 3 and 4 of the algorithm. The condition for determining n_1 and m in Step 2 is therefore $B_1(P_1, n_1) \leqslant B_{MAX}$. The function $B_1(P_1, n_1)$ is expressed by the following formula:

$$B_1(P_1, n_1) = \min\{x : x \geqslant \lceil P_1/x^{n_1} \rceil + 1, x \text{ integer}\}. \quad (3)$$

Proof of formula (3). Let x represent possible buffer sizes, in pages. After n_1 sort/merge passes, the file F_1' contains

$$m = \lceil P_1/x^{n_1} \rceil$$

sorted subfiles. The final steps (Steps 3 and 4) of JM1 require m + 1 pages of buffer; therefore, the buffer size must satisfy the condition

$$x \geqslant \lceil P_1/x^{n_1} \rceil + 1.$$

Formula (3) now follows from the fact that the minimum buffer size which satisfies this necessary condition is chosen.

Taking also into account the necessity of sorting file F_2, the required buffer for performing the join with the JM1 algorithm is expressed by

$$B_{JM1}(P_1, P_2, n_1, n_2) = \max\left\{B_1(P_1, n_1), \left\lceil \sqrt[n_2]{P_2} \right\rceil\right\}. \quad (4)$$

4. Comparison of the SBJ and JM1 algorithms

The cost of performing the join (i.e., the number of required page-fetches) is expressed by the following formula:

$$C = 2P_1\left(n_1 + \tfrac{1}{2}\right) + 2P_2\left(n_2 + \tfrac{1}{2}\right) + P_r. \quad (5)$$

Formula (5) is determined as follows:

- each S/M pass on a file F_i requires to read and write P_i pages, hence $2P_i n_i$ page fetches are required (read and write operations are assumed to have the same cost),
- both F_1 and F_2 have to be read by the last phase (the join pass) and then the result F_r is written, hence $P_1 + P_2 + P_r$ page fetches are required.

Formula (5) shows that the cost of performing the

ble. Therefore, with n passes it can sort a file of at most B^n pages.

Since the last pass SJ can be performed using a buffer with size $B \geqslant 2$ independently from the size of the two sorted input files, the maximum buffer size which is required for performing a join using n_1 and n_2 S/M passes on the files F_1 and F_2 respectively is expressed by

$$B_{SBJ}(P_1, P_2, n_1, n_2) = \max\left\{\left\lceil \sqrt[n_1]{P_1} \right\rceil, \left\lceil \sqrt[n_2]{P_2} \right\rceil\right\}. \quad (2)$$

3. One-way join-during-merge (JM1)

The one-way join-during-merge (JM1) algorithm consists in completely sorting one file and only partially sorting the other file and then performing the join. We will assume that file F_2 is completely sorted while file F_1 is only partially sorted. Algorithm JM1 works as follows:

Step 1. Sort one file (say F_2) in join column order.

Step 2. Perform n_1 sort/merge (S/M) passes on the other file (F_1), so that a partially sorted file F_1' is obtained consisting of m subfiles $F_{11}', \ldots, F_{1m}'$ (the condition for determining m and n_1 depends on the maximum available buffer size B_{MAX} and will be stated in the sequel).

Step 3. Read the first page of F_2 and of each subfile $F_{11}', \ldots, F_{1m}'$ into the buffer.

Step 4. Repeat until F_2 and F_1' have been completely scanned:

(a) Join the records of the page of F_2 contained in the buffer with the matching records of the pages of the subfiles F_{1i}', $1 \leqslant i \leqslant m$ contained in the buffer.

(b) Write the joined records to the output file.

(c) For each input page in the buffer which is completely processed, read a new page from the associated file F_2 or subfile F_{1i}', $1 \leqslant i \leqslant m$.

Consider now the problem of determining n_1 and m at Step 2. These must be such that the following Steps 3 and 4 can be performed with the available buffer B_{MAX}. Let $B_1(P_1, n_1)$ be the size of the buffer which is needed for performing the n_1 S/M passes on file F_1 and Steps 3 and 4 of the algorithm. The condition for determining n_1 and m in Step 2 is therefore $B_1(P_1, n_1) \leqslant B_{MAX}$. The function $B_1(P_1, n_1)$ is expressed by the following formula:

$$B_1(P_1, n_1) = \min\{x : x \geqslant \lceil P_1/x^{n_1} \rceil + 1, \text{ x integer}\}. \quad (3)$$

Proof of formula (3). Let x represent possible buffer sizes, in pages. After n_1 sort/merge passes, the file F_1' contains

$$m = \lceil P_1/x^{n_1} \rceil$$

sorted subfiles. The final steps (Steps 3 and 4) of JM1 require m + 1 pages of buffer; therefore, the buffer size must satisfy the condition

$$x \geqslant \lceil P_1/x^{n_1} \rceil + 1.$$

Formula (3) now follows from the fact that the minimum buffer size which satisfies this necessary condition is chosen.

Taking also into account the necessity of sorting file F_2, the required buffer for performing the join with the JM1 algorithm is expressed by

$$B_{JM1}(P_1, P_2, n_1, n_2) = \max\left\{B_1(P_1, n_1), \left\lceil \sqrt[n_2]{P_2} \right\rceil\right\}. \quad (4)$$

4. Comparison of the SBJ and JM1 algorithms

The cost of performing the join (i.e., the number of required page-fetches) is expressed by the following formula:

$$C = 2P_1\left(n_1 + \tfrac{1}{2}\right) + 2P_2\left(n_2 + \tfrac{1}{2}\right) + P_r. \quad (5)$$

Formula (5) is determined as follows:

- each S/M pass on a file F_i requires to read and write P_i pages, hence $2P_i n_i$ page fetches are required (read and write operations are assumed to have the same cost),
- both F_1 and F_2 have to be read by the last phase (the join pass) and then the result F_r is written, hence $P_1 + P_2 + P_r$ page fetches are required.

Formula (5) shows that the cost of performing the

join grows linearly with the number of required S/M passes n_1 and n_2. Therefore, the comparison of algorithms SBJ and JM1 can be done by comparing the numbers of S/M passes which are required by each one of them for performing the same join operation with a limited buffer B_{MAX}. The main relationship which is needed for comparing the two algorithms is expressed by the following proposition.

Proposition 4.1. *The sizes* $B_{SBJ}(P_1, P_2, n_1, n_2)$ *and* $B_{JM1}(P_1, P_2, n_1 - 1, n_2)$ *of the buffers which are required by algorithms* SBJ *and* JM1 *for performing the same join operation in* (n_1, n_2) *and* $(n_1 - 1, n_2)$ S/M *passes respectively, satisfy the following relationship*:

$$B_{JM1}(P_1, P_2, n_1 - 1, n_2) \leqslant B_{SBJ}(P_1, P_2, n_1, n_2) + 1.$$

Proof. Consider formulas (2) and (4). Since the term $\left\lceil \sqrt[n_2]{P_2} \right\rceil$, which is due to the sort of file F_2, is the same in both formulas, we need only to compare the other term.

Let $B = \left\lceil \sqrt[n_1]{P_1} \right\rceil$ be the buffer size which is required to completely sort file F_1. After $n_1 - 1$ S/M passes performed using a buffer of size B, m ordered subfiles are obtained. Clearly, $m > 1$, otherwise the last ($n_1 th$) S/M pass would not be required for completely sorting the file, and $m \leqslant B$, otherwise the last S/M pass would not be sufficient to completely sort the file.

The last pass of the JM1 algorithm requires $m + 1$ buffer pages, because it must keep in main memory one page for each of the m subfiles $F'_{11}, \ldots, F'_{1m}$ and one page of file F_2. Therefore,

(1) if $m < B$, the last pass of the JM1 algorithm can be performed using B pages,

(2) if $m = B$, the last pass requires $B + 1$ pages.

□

A useful corollary of Proposition 4.1 is the following.

Corollary 4.2. *The following relationship holds on the sizes of the buffers which are required for performing the* SBJ *and* JM1 *algorithms using the same number of* S/M *passes*:

$$B_{JM1}(P_1, P_2, n_1, n_2) \leqslant B_{SBJ}(P_1, P_2, n_1, n_2).$$

Sketch of proof. It is sufficient to show that the terms which refer to file F_1 in formulas (2) and (3) are related in the following way:

$$\min\{x : x \geqslant \lceil P_1/x^{n_1} \rceil + 1\} \leqslant \left\lceil \sqrt[n_1]{P_1} \right\rceil,$$

and the same reasoning for the proof of Proposition 4.1 can be used. □

Proposition 4.1 shows that it is possible to perform the same join saving one S/M pass by using the JM1 algorithm instead of the SBJ algorithm, provided that it is possible in some cases to use one additional buffer page. In the case that an additional buffer page is required and that it is impossible to obtain it (i.e., $B_{MAX} = B_{SBJ}(P_1, P_2, n_1, n_2)$ and $B_{JM1}(P_1, P_2, n_1 - 1, n_2) > B_{MAX}$), the JM1 algorithm can be performed using the same number of S/M passes and a buffer which is equal or smaller to the buffer which is required by the SBJ algorithm. As a consequence, we can state that *the* JM1 *algorithm dominates the* SBJ *algorithm*.

Concluding remarks on JM1

(1) It is convenient to completely sort the smaller one of the two files, thus saving one S/M pass of the larger one.

(2) Since, with today's technology, large main memory buffers are available, typical values of n_1 and n_2 are in the order of 2 to 4 passes (for example, with a page size of 1 Kbyte, a 10 Mbyte file can be sorted on a 22 Kbyte memory buffer in $n = 3$ passes) and therefore the relative gain of saving one pass is not irrelevant.

5. Two-way join-during-merge (JM2)

If the available buffer B_{MAX} is large enough, it is possible to save the last S/M pass on both files F_1 and F_2. This is done by algorithm JM2. We first describe algorithm JM2 and then state the condition which allows performing it.

Step 1. Perform n_1 and n_2 S/M passes on files F_1 and F_2, so that two partially sorted files F'_1 and F'_2 are obtained consisting of m_1 and m_2 subfiles $F'_{11}, \ldots, F'_{1m_1}$ and $F'_{21}, \ldots, F'_{2m_2}$ (the condition for

determining n_1, n_2, m_1 and m_2 depending on the maximum available buffer size B_{MAX} will be stated in the sequel).

Step 2. Read the first page of each subfile F'_{11}, ..., F'_{1m_1} and F'_{21}, ..., F'_{2m_2} into the buffer.

Step 3. Repeat until F'_1 and F'_2 have been completely scanned:

(a) Join the records of the pages of the subfiles F'_{1i}, $1 \leqslant i \leqslant m_1$ with the matching records of the pages of the subfiles F'_{2j}, $1 \leqslant j \leqslant m_2$.

(b) For each page in the buffer completely processed, read a new page from the associated subfile F'_{1i}, $1 \leqslant i \leqslant m_1$ or F'_{2j}, $1 \leqslant j \leqslant m_2$.

The following formula expresses the relationship between the required buffer size $B_{JM2}(P_1, P_2, n_1, n_2)$ and the numbers n_1 and n_2 of S/M passes which have to be performed on files F_1 and F_2 respectively in order to obtain the partially sorted files F'_1 and F'_2:

$$B_{JM2}(P_1, P_2, n_1, n_2) = \min\{x : x \geqslant (\lceil P_1/x^{n_1} \rceil + \lceil P_2/x^{n_2} \rceil)\}. \qquad (6)$$

Proof of formula (6) The proof is essentially the same as the proof of formula (3). Let x be a possible buffer size (in pages); after the n_i S/M passes performed in Step 1, each file F'_i contains $m_i = \lceil P_i/x^{n_i} \rceil$ sorted subfiles. Steps 2 and 3 require $m_1 + m_2$ pages of buffer. Formula (6) now follows from the fact that the minimum buffer size which satisfies this condition is chosen.

The condition for determining n_1, n_2, m_1 and m_2 in Step 1 of the JM2 algorithm is clearly $B_{JM2}(P_1, P_2, n_1, n_2) \leqslant B_{MAX}$.

The convenience of using algorithm JM2 instead of algorithm JM1 depends on the available buffer size B_{MAX} and on the file sizes P_1 and P_2. Assume that B_{MAX}, P_1, and P_2 are such that the SBJ algorithm can be performed using n_1 and n_2 S/M passes and the JM1 algorithm can be performed in $n_1 - 1$ and n_2 passes. Then, the JM2 algorithm should be used if and only if the condition

$$B_{JM2}(P_1, P_2, n_1 - 1, n_2 - 1) \leqslant B_{MAX}$$

holds, because in this case the JM2 algorithm allows saving one S/M pass on file F_2 with respect to algorithm JM1.

6. Example and conclusion

Two join-during-merge algorithms have been described and analyzed. Their superiority with respect to the traditional Sort-Based algorithm has been shown. Table 1 shows, with an example, that the percentual gain which can be achieved by using JDM instead of SBJ is significant. In Table 1 it is assumed that two files F_1 and F_2, having sizes $P_1 = 10\,000$ and $P_2 = 1000$ pages respectively, are joined using buffers of sizes from 10 to 999 pages. For each algorithm (SBJ, JM1, and JM2) the numbers n_1 and n_2 of required passes on files F_1 and

Table 1

Buffer size	Algorithms						The best algorithm	Percentual gain
	SBJ		JM1		JM2			
	n_1	n_2	n_1	n_2	n_1	n_2		
10	4	3	4	3	4	3	JM1	–
11–12	4	3	3	3	3	3	JM1	20.4
13–21	4	3	3	3	3	2	JM2	22.4
22	3	3	2	3	2	3	JM1	25.6
23–31	3	3	2	3	2	2	JM2	28.2
32–36	3	2	2	2	2	2	JM1	26.3
37–99	3	2	2	2	2	1	JM2	28.9
100	2	2	2	1	2	1	JM1	3.6
101–105	2	2	1	2	1	2	JM1	35.7
106–999	2	2	1	2	1	1	JM2	39.3

F_2 are listed, then the most convenient algorithm is indicated and in the last column the percentual gain of the best algorithm with respect to SBJ is shown.

The percentual gain has been calculated under the most negative assumption that the size P_r of the result file F_r is maximum ($P_r = 1000$); since F_r must be written by all algorithms, if the result file is smaller (i.e., few tuples have matching values), then the percentual gain is higher.

Table 1 shows that: (1) except for the two cases $B_{MAX} = 10$ and $B_{MAX} = 100$ the JDM algorithms achieve a relative gain varying from 20% to 39%, and (2) that for most values of B_{MAX} the JM2 algorithm can be used.

Recall that even for the case $B_{MAX} = 10$, although no gain can be achieved in page fetches, the use of JM1 is still convenient, because it uses a smaller buffer than SBJ (Corollary 4.2).

Let us finally state that, since the choice of the best algorithm for a given join depends only on the values P_1, P_2, and B_{MAX}, it is possible to choose the optimal algorithm *before* performing the join. Therefore, the stated gains can be effectively achieved in reality.

References

[1] W. Kim, A new way to compute the product and join of relations, Proc. ACM/SIGMOD Internat. Conf. on Management of Data, 1980.

[2] S.B. Yao, Optimization of query evaluation algorithms, ACM/TODS 4 (2) (1979).

[3] M.W. Blasgen and K.P. Eswaran, Storage and access in relational data bases, IBM Systems J. 16 (4) (1977).

[4] H. Lorin, Sorting and Sort Systems (Addison-Wesley, Reading, MA, 1975).

[5] P. Valduriez and G. Gardarin, Join and semijoin algorithms for a multiprocessor database machine, ACM/TODS 9 (1) (1984).

[6] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching (Addison-Wesley, Reading, MA, 1973).

[7] D. Bitton and D.J. DeWitt, Duplicate record elimination in large data files, ACM/TODS 8 (2) (1983).

[8] G.M. Sacco and S.B. Yao, Query optimization in distributed database systems, in: Advances in Computers (Academic Press, New York, 1982) 225–275.

[9] W. Kim, On optimizing and SQL-like nested query, ACM/TODS 7 (3) (1982).