IMPLEMENTATION: Data Structures

Relational Information Systems — Chapter 1.2 (Revised 99/9)

Sequential Files

September 28, 1999

The author gratefully acknowledges support from the taxpayers of Québec and of Canada who have paid his salary and research grants while this work was developed at McGill University, and from his students (who built the implementations and investigated the data structures and algorithms) and their funding agencies.

1 Sequential Files

A *sequential file* has a sequence of records which may be accessed only by starting at the first record and traversing all the others until the desired record is reached. It may be organized on secondary storage either by storing the blocks in physical sequence, or by using pointers to retrieve the blocks in sequence. (Pointers may not be used to retrieve the *records* individually in sequence, except through their location in the blocks, because this would require transfers to and from secondary storage of smaller units of data than the block, prohibited by considering the access/transfer ratio.)

While the records are thus always in a fixed sequence, they may or may not be *ordered* according to the value of some field or fields within the record. A file of records which are not ordered according to any value is called an *unordered* file; a file of records which are ordered according to one of the fields, or according to one set of fields, is called *ordered.* These are the only two kinds of sequential file, and they behave differently, as figure 1 shows. Figure 1 gives the expected number of accesses to a sequential file in a search for one of its records (assuming all records are equally likely to be sought: we modify this assumption later). In the successful searches, the record sought may be at the beginning of the file (1 access), it may be at the end of the file (n accesses), or, anywhere else: on the average we will search

	Successful	Unsuccessful
Ordered	$n/2$	$n/2$
Unordered	$n/2$	n

Figure 1: Searching Ordered and Unordered Sequential Files

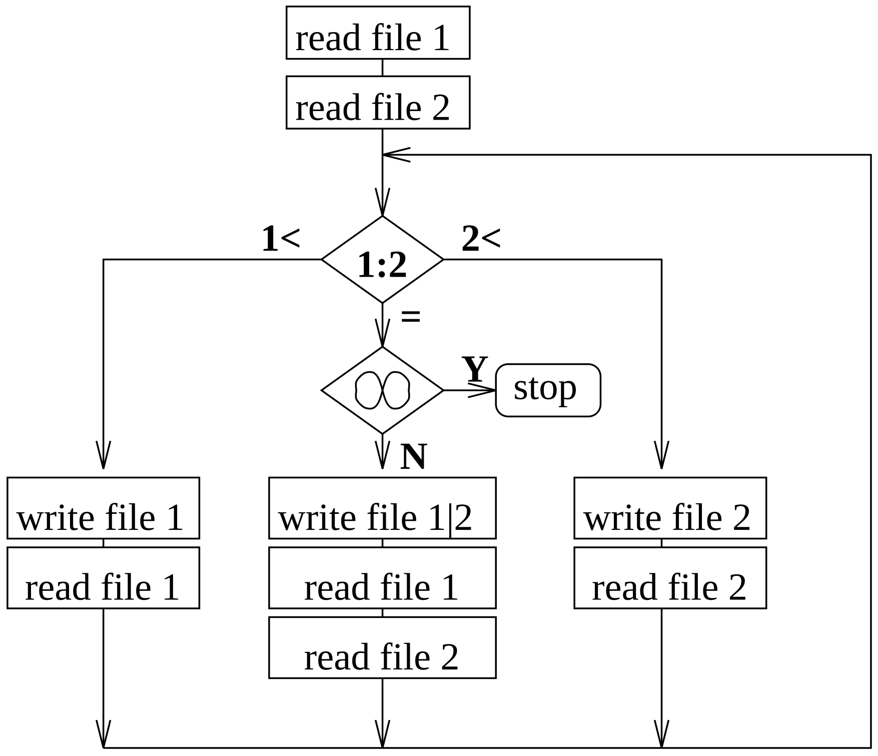

Figure 2: Merge Logic to Find the Union of Sorted Files

half the file. It does not matter here whether we are searching on a field on which the file is ordered or not. For the unsuccessful searches, the search on the ordered file can halt as soon as the position where the record should be has been passed, and this is again expected to be halfway along the file. There is no such stopping criterion for an unsuccessful search on an unordered file, and the search must continue to the end of the file in all cases.

Searching a sequential file for a single record does not seem to be a particularly good thing to do. Merging two sequential files is a much more common and useful operation. We must suppose that both files are sorted, that is, ordered by the field that is to be used as a comparison in the merge (or by the set of fields so used). Figure 2 shows the logic needed to merge two sorted files in order to find the set union of the files. This supposes that the files are compared on one (set of) field(s) and that all other fields play no role in the operation. In particular, if records in the two files match on the comparison field(s), the remaining fields are supposed to be identical. Altering the logic if this is not the case is very simple and we do not dwell on it. In the figure, "read file .." means read a record, return ∞ at end of file

(EOF); “1:2” means compare key fields;and “1<” means the key of file 1 is low.

Simple modifications of the logic can be used to find intersection, difference, etc., and to allow the values of one file to update the values of the other.

We can summarize activity, volatility, and symmetry in the case of sequential files. *Activity* should be high if we plan to use sequential files. For instance, to look up a single student in a student record file of 30,000 entries, we would expect to look up half the file, according to figure 1: not an appealing prospect. If on the other hand we wanted to calculate the grade point average of all students, we would need to process every record, and the cost would be n block acceses, or one pass. In the first case, the activity is 1/30,000, or 0.003%; in the second case, it is 100%.

(We will find that above an activity of some 0.01%, sequential files are faster than the best possible organization for low activity, direct files. This is a surprizingly low number. It is of some practical importance to note it, because sequential files are very much easier to build and run than anything else. We tend to speak of an activity below this “breakeven” threshold as “low”, and an activity above it as “high”. You should wait for the derivation later before using 0.01% as a breakeven activity: it depends on tha access/transfer ratio, which has changed as technology has improved, and on record size.)

Sequential files are not generally good for high *volatility*. Inserting or deleting records in the middle of a sequential file generally means moving later records towards the end or towards the beginning of the file. Changing values in the records does not require reorganization, unless the file is sorted and the change is to the sort field: then the record may have to be deleted from its current position and added to another, with concomitant shifting of all the other records. However, this is just a rule of thumb: a sequential file ordered by time to record a time sequence, can be added to indefinitely. It just grows at the high-time end, which is the back of the file.

Symmetry is considered high if at least two fields can be queried independently of each other at the same cost. A sorted file has low symmetry because it is sorted on one field (or set of fields: we do not make a distinction). Figure 1 makes this clear. The cost of searching on the sort field (such as *name* in a telephone book) is given by the top row. The cost of searching on any other field (such as *address* in a telephone book) is given by the bottom row.

On the other hand, perversely, an unordered file has high symmetry because all fields may be retrieved with equal difficulty (the bottom row of figure 1).

Since we have supposed in most of the above discussion that the files are sorted, we should discuss *sorting on secondary storage*. This turns out to be more straightforward than sorting in RAM, because the constraints of secondary storage permit a smaller variety of methods. The external sort is a merge sort, starting with the generation of “initial runs”, fully sorted subsets of the original data, and merging these into ever longer runs, until all the data is in one run, the final sorted file.

Figure 3 illustrates the second step, merging. It shows five initial runs of various lengths being merged in two passes of the data, first to two runs, then to one.

There are two important considerations due to secondary storage. First, the emphasis is on doing complete passes of the data: a *pass* is a sequential read or write from the beginning to the end of the file. This is because, if the data blocks of the file are stored contiguously and if there is no contention by other users for the disk, a file can be read or written from beginning to end without need to wait for disk latency or arm movement: the access/transfer ratio comes into play only at the very beginning of the file. In the merge, the write operations at any one level can be taken to form a single pass. The read operations need access to

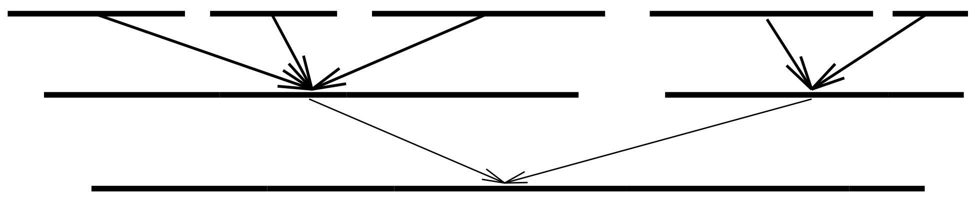

Figure 3: The Merge Phase of a Mergesort

several runs at once, and so can cause disk contention within the merge process, unless there are more than one disk, or at least more than one arm of read/write heads. [1]

The second is a significant difference between secondary storage and RAM. The "fanin" of the merge can be greater than two, and, in fact, the larger the fanin the better. The *fanin* is the number of runs which are merged at any one time. We earlier discussed binary merge, with fanin = 2. A similar process gives multiway merge: instead of testing for "low" or not, we must find the least value of all the inputs and write that. In RAM, the best way to find the least of several values is to place them all in a binary tree (called here a "heap"). The result is that the number of passes is some $\log_2$ of the number of items being merged. On secondary storage, the cost is not in the comparisons made between values, but in the accesses made to disk. (We suppose, as a guiding rule (although things are not always so simple), that comparisons, and other operations done in RAM after the block has been transferred, are free.) So, on secondary storage, a high fanin means a high base for the logarithm, which in turn means fewer passes than if the base were 2. In RAM, the base of the logarithm is 2 no matter what. On secondary storage, it can be made as high as is practical in light of other considerations.

In figure 3, the (maximum) fanin is 3, and so the five initial runs are merged in two passes. A mergesort in RAM would require $\lceil \log_2 5 \rceil = 3$ passes.

Now we must ask how to do the first step of the mergesort, creating the initial runs. This is a multiway merge of the file with itself, called "replacement selection". It is best described by illustration: we make initial runs from the following data. `Ar, H, Go, S, K, D, Sh, T, Ti, W, De, R, Hu, B, L, Es, Le L, Lit, G, Se, Tr, Mo, A, Li, E, St, Br, Row, My, Sm, M, Wi`[2]

Figure 4 shows this input, a RAM buffer with a capacity of eight records, and the resulting output (which also requires further buffering in RAM, not shown). The first eight records are read into the buffer, and the smallest of these, `Ar`, is selected and written to the output buffer. It is replaced by the next record, `Ti`, in the input file. The smallest of the eight records now in the buffer, `D`, is selected, output, and replaced by `W`. The third record to be selected, `Go`, is replaced by `De`, a smaller value than the last output (`Go` itself), and so must be flagged (parenthesized in the figure) to be held in the buffer and not output until the current run is completed. That finally happens after `W`, the 11th record, is written. Then all records in the input buffer are flagged, so the flags are removed and we start again with a new run.

[1]Most of the subtlety of merge sorting, which we do not discuss here, lies in the arrangement of the initial runs on the various storage devices, and depends on whether we have four or more tape units, one or two disk drives, etc. Devising merge patterns which minimize tape rewinds or disk arm motion has produced the variety in external sorting methods ([1], sect. 5.4).

[2]This is the list of abbreviations for some of the participants in the Montréal Very Large Data Bases conference, without duplicates and in order of their appearance in the initial sessions. Session 0: Armstrong, Hsiao, Gotlieb. Session 1: Schmidt, Kent, Date, Shneiderman, Thomas, Ting, Walker, Demolombe, Rosenkrantz, Hunt. Session 2: Bochmann, Le Bihan, Esculier, Le Lann, Litwin, Gardarin, Sedillot, Treille, Mohan, Adiba, Lindsay, Epstein, Stonebraker, Brodie, Schmidt, Rowe, Mylopoulos, Smith, McLeod. Session 3: McLeod, Wilson.

Input	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Output**
Ar	Ar	H	Go	S	K	D	Sh	T	Ar
H	Ti	|	|	|	|	|	|	|	D
Go	|	|	|	|	|	W	|	|	Go
S	|	|	(De)	|	|	|	|	|	H
K	|	R	|	|	|	|	|	|	K
D	|	|	|	|	(Hu)	|	|	|	R
Sh	|	(B)	|	|	|	|	|	|	S
T	|	|	|	(L)	|	|	|	|	Sh
Ti	|	|	|	|	|	|	(Es)	|	T
W	|	|	|	|	|	|	|	(Le L)	Ti
De	(Lit)	|	|	|	|	|	|	|	W
R	Lit	B	De	L	Hu	G	Es	Le L	B
Hu	|	Se	|	|	|	|	|	|	De
B	|	|	Tr	|	|	|	|	|	Es
L	|	|	|	|	|	|	Mo	|	G
Es	|	|	|	|	|	(A)	|	|	Hu
Le L	|	|	|	|	Li	|	|	|	L
Lit	|	|	|	E	|	|	|	|	Le L
G	|	|	|	|	|	|	|	St	Li
Se	|	|	|	|	(Br)	|	|	|	Lit
Tr	Row	|	|	|	|	|	|	|	Mo
Mo	|	|	|	|	|	|	My	|	My
A	|	|	|	|	|	|	Sm	|	Row
Li	(M)	|	|	|	|	|	|	|	Se
E	|	Wi	|	|	|	|	|	|	Sm
St	|	|	|	|	|	|	∞	|	St
Br	|	|	|	|	|	|	|	∞	Tr
Row	|	|	∞	|	|	|	|	|	Wi
My	M	∞	∞	E	Br	A	∞	∞	A
Sm	|	|	|	|	|	∞	|	|	Br
M	|	|	|	|	∞	|	|	|	E
Wi	|	|	∞	|	|	|	|	|	M
	∞	|	|	|	|	|	|	|	

Figure 4: Replacement Selection Giving Three Initial Runs: Eight-way Self-Merge

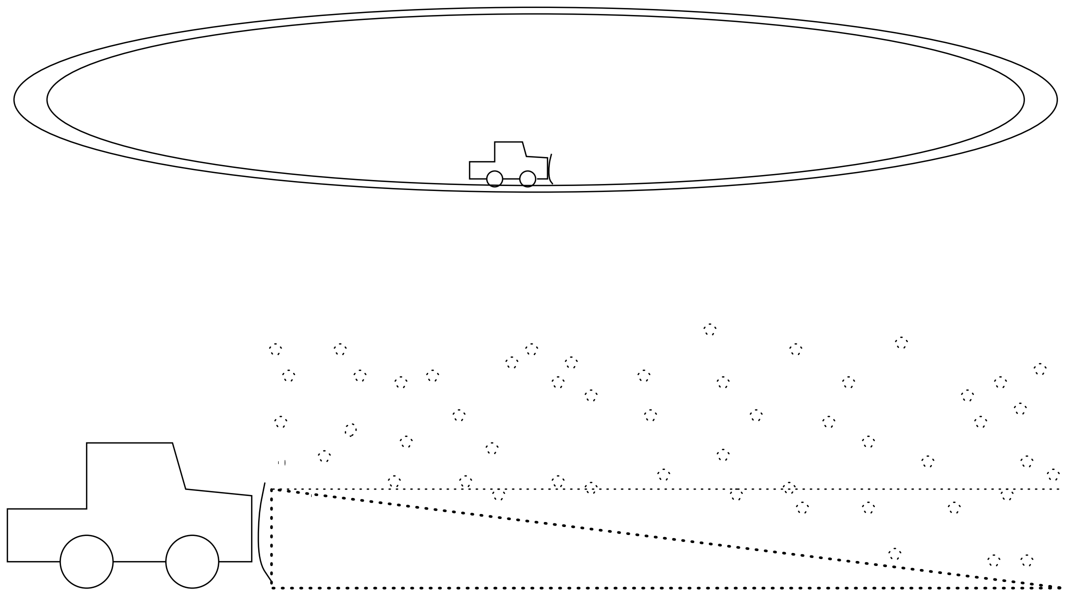

Figure 5: The Snowplough Argument

In figure 4, time is imagined to flow from top down in two different ways. For the input data, the records are read into the buffer from top to bottom. For the eight input buffers and the output buffer, each row of the input buffers is their state after the output of the row above has been written: no change is marked by | for clarity.

An important question is, how big are the resulting runs? We see that the first run is bigger than the input buffer capacity, 11 as opposed to 8. The second run, with 17 records, is more than twice as big. With this in mind, we can see why a more obvious way of generating initial runs would not be as good: read an input buffer full, sort it internally, and write out the sorted records as a run. This would never give runs larger than the buffer.

With replacement selection, all runs would be the same size as the input buffer only in the worst case that all the records were initially sorted in the opposite order. If all records were sorted correctly before input, replacement selection would produce a single run, the fully sorted file, in one pass.

A nice argument involving a snowplough on a circular road in a snowstorm shows that the expected length of the initial runs is exactly twice the RAM capacity of the input buffer ([1], sect. 5.4.1). The incoming records are the falling snowflakes, distributed uniformly over the road, with the coordinate of the snowflake along the road corresponding to the value of the sort field in the record. The smallest record currently in RAM (i.e., the output) is the position of the plough. The number of records in RAM is the amount of snow on the road, whose height decreases linearly from a maximum just in front of the plough to zero just behind it. The expected size of the initial run is the total snow removed by the plough in one trip around the circle—which is just twice the amount of snow on the road at any one time because the plough is always pushing away an accumulation of a fixed height (steady state).

References

[1] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume III. Addison-Wesley Publishing Co., Reading, Mass., 1973. 1st edition.