Recent Advances in External Sorting

C. James[1]

Abstract

A recent advancement in external sorting is the radix hash sort. It combines radix sorting with perfect hashing as implemented in a single linked list. The performance of radix hash for disk to disk operations is 65% faster for any key size than the nearest published method which is the distribution counting sort named Algorithm 5.2.D of D. E. Knuth, as adapted here for external sorting. For radix hash, the hash size of 2-bytes is better for sorting less than about 2 million keys, and the

[1] Principal Scientist, CEC Services, LLC, 1613 Morning Dr, Loveland CO 80538-4410, Cell: 719.210.9534, Fax: 970.593.1350, RadixHashSort@CEC-Services.com

hash size of 3-bytes is better for sorting more. The source code for the sorting methods is included in True BASIC®.

Introduction

External sorting, also known as disk-to-disk sorting, is necessary when the keys to be sorted can not fit into memory. The usual approach is to modify some internal sort, or memory-to-memory sort, for external sorting. This paper describes a recent advance in external sorting named the radix hash sort. It combines the features of radix sorting with perfect hashing.

Radix sorts from right to left, from the least significant position to the most significant position within a key. An example of radix sorting uses the limited character set of the decimal numbers zero through nine to sort monetary amounts. Radix sorts the right most

decimals or cents positions then proceeds to sort the leftmost, non decimal or dollar positions.

Hash sorting maps a key into a location in a hash table where multiple keys may map to the same location and hence collide. Perfect hash sorting avoids the collision by mapping each possible key into a unique location in a hash table or file.

Problem Statement

If multiple records with the same sort key map to the same hash location, how are records tabulated as different records with the same key. A solution is to allow the size of the location to grow infinitely to accommodate an ever increasing number of records hashed there. For internal sorting, this may be implemented in computer memory by catenating the record numbers in order.

However, this requires rewriting the location of the memory string each time a record is added. This operation is resource costly and time consuming. A solution that avoids rewriting the same location is to expand new locations as needed into free space already reserved by using pointer links. This method is known as a linked list. A linked list may link backward or forward as a single linked list, or may link both backward and forward as a double linked list.

Approach and Techniques

To implement perfect hashing for radix sorting, a single linked list is chosen that links only forward. In order to locate where the next new record is added, the list may be read from beginning to end. A faster method is for any hash value to store a pointer to the next available

location for the link to a new record. This pointer is stored twice. It is stored in the hash table at the hash key index as the link from the last record accessed for that hash key. The pointer is also stored in another table as the last link associated with that hash key. This other table is named the next record update (NRU) table. What follows is how the NRU table and the hash table are logically constructed.

Given a hash key with a size of 2-bytes, the index range for hash values is 0 through 65535. For each of the respective hash values, the next record available is stored. The NRU table thus contains 65536 entries with pointers to the respective next available records in the hash table. The NRU table is initialized by hash index to the respective value of that hash index. Record 1 of the NRU table is indexed as 0 with a value of 1 to point to record 1

in the hash table. Record 65535 of the NRU table is indexed as 65535 with a value of 65536 to point to record 65536 in the hash table.

The hash table entry contains two data, the record number of the key sorted and a link in the hash table to the next available record location. This implies that the number of the entries in the hash table is the number of possible hash keys plus N entries for the N keys to be sorted. Therefore the number of keys to be sorted should be known, or determined, before the sorting begins.

Here is an example of how the NRU table and the hash table interact. When hash value 2 is encountered for the first time as the first key in the input file, the NRU table is accessed at index position 2 which points to record 3 in the hash table. An NRU counter beginning at 65536

is incremented to 65537. It is stored as the last updated record link in the NRU table at index position 2. In the hash table at record 3, the pointer is updated to the key index for the input file, and the associated link for the next free record in the hash table is also updated to 65537. Record 65537 in the hash table is a record containing values of zero. Therefore if the value of the link record is zero then that record terminates the hash chain. If the value of the link record is not zero then that record points to the next sorted key of the input file in the hash chain.

What remains is how to terminate the linked list when it is traversed. The answer is to rely on a blank record as the sentinel record. When a hash record links to a blank record, the last linked record was obtained. Therefore the hash table also contains a number of blank

sentinel records to equal the number of hash values to sort. This makes the total number of entries in the hash table as 65536 plus the N keys to sort. The hash table is initialized to zero.

After the keys to be sorted are hashed, it is necessary to construct a table of sorted pointers. This table is initialized by index to the respective input record number to be sorted as 1 through n. For example, sort key index 1 is initialized to value 1, sort key index 2 is initialized to value 2, and sort key index N is initialized to value n. For the first hash pass, this table serves as the index to the input file of keys. After each hash pass, this table of sorted pointers is subsequently updated and serves also as the next index to the input file of keys to be sorted.

To evaluate the performance of radix hash sorting, the algorithm was programmed to make all table accesses to and from files on disk. The implementation was not programmed in MMIX assembly language so as to obtain live, empirical sorting results more easily from a higher level programming language that is dependent on the current load of computer hardware and operating systems as implemented or used by a casual tester. Programming was in True BASIC®. The nearest published algorithm to radix hash is the distribution counting sort known as Algorithm 5.2.D (Knuth 1998). That algorithm was also programmed to sort externally as disk-to-disk.

For both sorts, files were initialized or re-initialized as needed with one read from a clear file and one write to a sort file. The clear files were preloaded.

The time to build the clear files was not included in the tabulation of performance times.

Results

For about 16 million or 2 ^ 24 keys of any length with a hash size of 2-bytes, external radix hash performs 65% faster than external 5.2.D. Table 1 has the performance data in time units * 1000. The graph of performance in Figure 1 shows that radix hash is the lower curve. Both sorts are linear in time. The graph of logarithmic performance in Figure 2 shows both sorts are parallel. The sorts processed an input key of 4-bytes, meaning that two passes were required to sort with the 2-byte hash key. The source code for the two sorts are included as Program Listing 1 and 2.

What remains is to analyze the performance results from increasing the byte length of the keys to be sorted and from increasing the byte size of the hash key.

To test an increase in length of the key to be sorted, a hash size of 2-bytes and N = 2 ^ 19 or about 500, 000 records are chosen arbitrarily. From Table 1 with the length of the key of 4-bytes and from Table 2 with the length of the key of 6-bytes, radix hash and Algorithm 5.2.D perform about 50% slower with a 6-byte key than with a 4-byte key. This is to be expected because the graphs of both sorts appear linear.

To test an increase in length of the hash key, a hash size of 3-bytes was chosen because of testing limitations due to hard disk size. A three byte hash size contains values in the range of 0 to (256 ^ 3) – 1 or about 16

million. Each hash value indexes a hash record that contains two pointer links in the IEEE 8-byte numeric string format for a total of 16-bytes per hash value. For radix hash of N keys to be sorted, to initialize the hash file requires a clear file that is (N + (256 ^ 3) – 1) * 16 bytes. For N = 2 ^ 22 keys to be sorted, the clear file size is therefore about 320 MB. A 4-byte hash size contains values in the range of 0 to (256 ^ 4) – 1 or about 4 GB. The clear file size for N = 4 MB is about 64 GB. By contrast to the 3-byte hash size, the 4-byte hash size is thus impractical to test or to use.

Table 3 shows that with a 3-byte hash key, Algorithm 5.2.D performs worse than its 2-byte hash key for about N = 2 ^ 19 or 0.5 million keys. By contrast Table 3 shows that with a 3-byte hash key, radix hash

performs better than its 2-byte hash key at about N > 2 ^ 21. Therefore radix hash with a 2-byte hash key is better suited to sorting keys that number less than 2 million.

Future Directions

Planned enhancements to the radix hash sort are due to appear as a commercial product in 2006 under the product name of RadixHash™.

Acknowledgments

Thanks are due to Professor Emeritus Donald E. Knuth of Stanford University for pointing out that the original presentation of radix hash sorting (James 2005) was very similar in theory to that of Algorithm 5.2.D.

References

James 2005

James, C. Statistical Analysis of the Relative Strength of Chess Positions. Pattern Recognition and Image Analysis. 2005. Vol. 15. No. 3. pp. 609-613.

Knuth 1998

Knuth, D. E. The Art of Computer Programming. 2nd ed. Reading MA: Addison-Wesley. 1998. pp. 78-9, 176-177.

(The following Figures are for two columns only.)

Figure 1. Performance of Radix Hash and Algorithm 5.2.D.

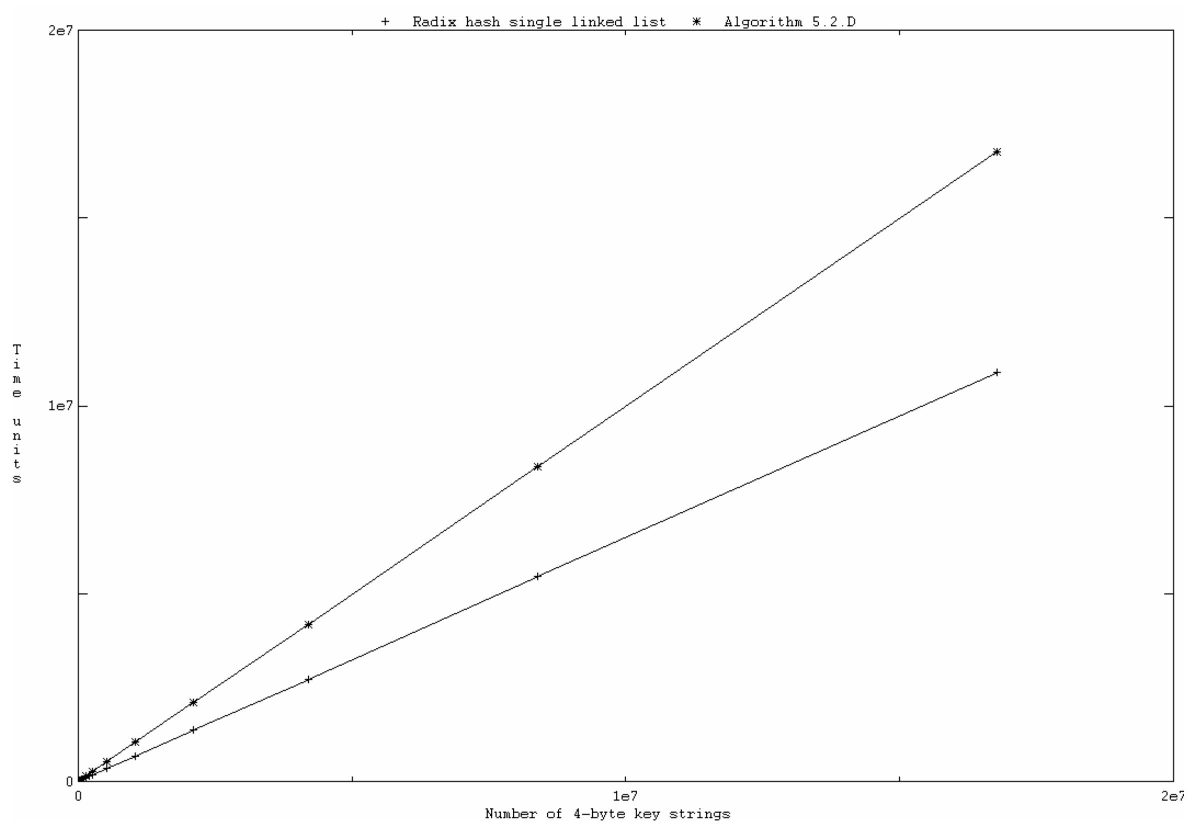
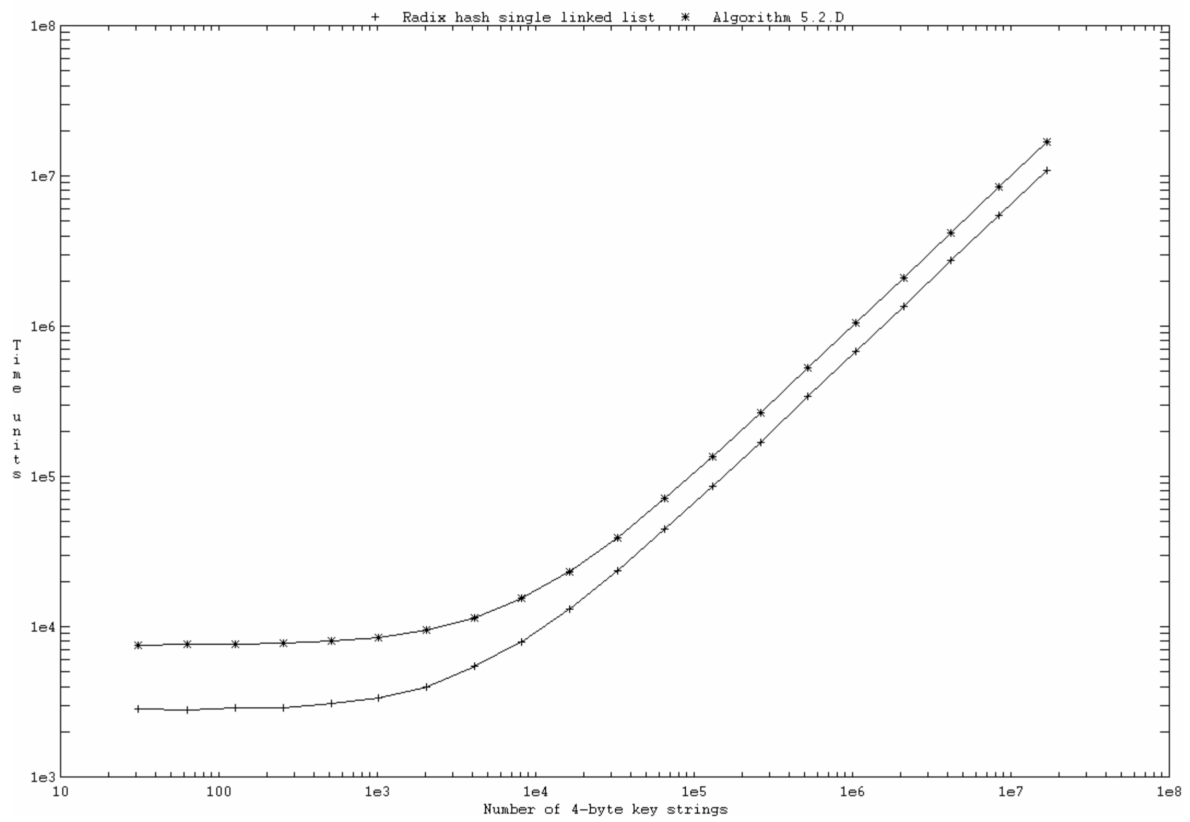


Figure 2. Logarithmic performance of Radix Hash and Algorithm 5.2.D.



(The following Tables are for one column only.)

Table 1. Sort performance for Radix Hash Algorithm 5.2.D with key length of 4-bytes and hash size of 2-bytes.

N = (2 ^ x) - 1	Radix hash	Algorithm 5.2.D.
5	2828	7562
6	2797	7610
7	2859	7687
8	2907	7781
9	3078	7985
10	3328	8468
11	4000	9438
12	5422	11391
13	7937	15375
14	13125	23250
15	23469	39109
16	44609	70984
17	85953	135813
18	169281	266344
19	343047	528859
20	683281	1050672
21	1364063	2095656
22	2725609	4186719
23	5449141	8388607
24	10897203	16772781

Table 2. Sort performance for Radix Hash and Algorithm 5.2.D with key length of 6-bytes for hash size of 2-bytes and of 3-bytes.

N = (2 ^ x) - 1	Hash size of 2-bytes		Hash size of 3-bytes	
	Radix hash	Algorithm 5.2.D	Radix hash	Algorithm 5.2.D
19	518859	812047	1065593	2471641
20	1032079	-	1390906	-
21	2058750	-	2079437	-
22	4112734	-	3449953	-

(The following Listings are for one column only.)

Program Listing 1. Radix hash with single linked list as external disk-to-disk.

Note: The acronym in the paper NRU (next record updated) means the same thing as the acronym LRU (last record updated) in the source code below.

[available on request to RadixHash@CEC-Services.com]

Progrmming Listing 2. Algorithm 5.2.D. as external disk-to-disk.
Note: The acronym in the paper NRU (next record updated) means the same thing as the acronym LRU (last record updated) in the source code below.

[available on request to RadixHash@CEC-Services.com]