# SEKE '96

# The 8th International Conference on Software Engineering and Knowledge Engineering

## PROCEEDINGS

**Sponsored by:**

Knowledge Systems Institute (Founder and Organizer)

**Co-Sponsored by:**

Arizona State University
Envision Technologies Corporation
Pan American Center for Earth and Environmental Studies
University of Pittsburgh
University of Texas at El Paso

**In Cooperation with:**

ACM (SIGSOFT)
IEEE Computer Society (TC on Software)

*Technical Program*, June 10-12, 1996

Hyatt Regency, Lake Tahoe, Nevada, USA

# The Theory of Massive Cross-Referencing

Dong-Keun Shin

Samsung Electronics Co., Ltd.
8-2, Karak-Dong, Songpa-Ku
Seoul, Korea

## Abstract

The problems associated with cross-referencing of massive amounts of items from one list to another are that most operations to date have been time-consuming and ineffective. This paper suggests Shin's algorithm as the solution to the problem of massive cross-referencing. Shin's algorithm's ability to divide items into a fixed number of buckets to allow processing elements of each bucket to filter unnecessary data in parallel (what is called the divide and conquer strategy) and its inherent characteristic of parallel processing makes it highly adaptable for massive cross-referencing operations. This paper will illustrate several parallel processing methods of this algorithm at the software level, the processor level, the memory level, and the auxiliary storage level, and show how the inherent qualities of this algorithm accelerate the massive cross-references from one list to another. The paper will also explain how the algorithm is applied to solve the problem of redundancy checking.

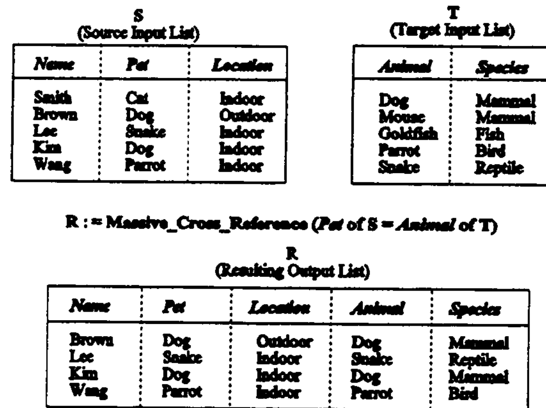
## 1 Introduction

Algorithms for sorting, searching, hashing, and redundancy checking are often used when there are multiple items in a single input list. There are many efficient algorithms already in these areas, but when there is a single input list (i.e., two identical input lists) or multiple input lists, and each list has multiple items, an efficient algorithm is needed for massive cross-referencing. Massive cross-referencing, also known as join or more specifically equi-join database management operation, is frequently used but it is too time-consuming.

An example of massive cross-referencing is shown in Figure 1. In this example, there are two input lists: the source list (S) and the target list (T). For the output list, there is the resulting list (R). Each list has items (i.e., rows) and columns, and each item is composed of one or more components. Each column can be specified by a title (e.g., ***Name***, ***Pet***, ***Location***, ***Animal***, and ***Species*** as shown in the example).

Copyright © 1996 by Shin, Dong-Keun. All rights reserved.

A component or a group of components can be used as a key (which can be called either a key component or key components) for massive cross-referencing. In the example, ***Pet*** is used as a key for the source list and ***Animal*** is used as a key for the target list. If a key of the source list is equal to a key of the target list (e.g., ***Pet*** of **S** = ***Animal*** of **T**), matched items in the **S** and **T** lists are merged to produce an item in the resulting list.



S (Source Input List)

| *Name* | *Pet* | *Location* |
|---|---|---|
| Smith | Cat | Indoor |
| Brown | Dog | Outdoor |
| Lee | Snake | Indoor |
| Kim | Dog | Indoor |
| Wang | Parrot | Indoor |

T (Target Input List)

| *Animal* | *Species* |
|---|---|
| Dog | Mammal |
| Mouse | Mammal |
| Goldfish | Fish |
| Parrot | Bird |
| Snake | Reptile |

**R : = Massive_Cross_Reference (*Pet* of S = *Animal* of T)**

R (Resulting Output List)

| *Name* | *Pet* | *Location* | *Animal* | *Species* |
|---|---|---|---|---|
| Brown | Dog | Outdoor | Dog | Mammal |
| Lee | Snake | Indoor | Snake | Reptile |
| Kim | Dog | Indoor | Dog | Mammal |
| Wang | Parrot | Indoor | Parrot | Bird |

Figure 1: An Example of Massive Cross-Referencing

The author's algorithmic solution to the massive cross-referencing and several implementation groupings of the Shin's algorithm are illustrated in this paper.

## 2 Shin's Algorithm for Massive Cross-Referencing

As shown in Figure 2, Shin's algorithm repeatedly divides the source and target lists by a maximum of five functionally different hash coders and filters unnecessary items whenever they are detected. After completing a hashing (or division) process, the algorithm checks whether or not the source items and the target items in a pair of source and target buckets have an identical key. If so, the source and target items in the pair of buckets are then merged in order to produce items for the resulting list. Otherwise, the return address of the current pair

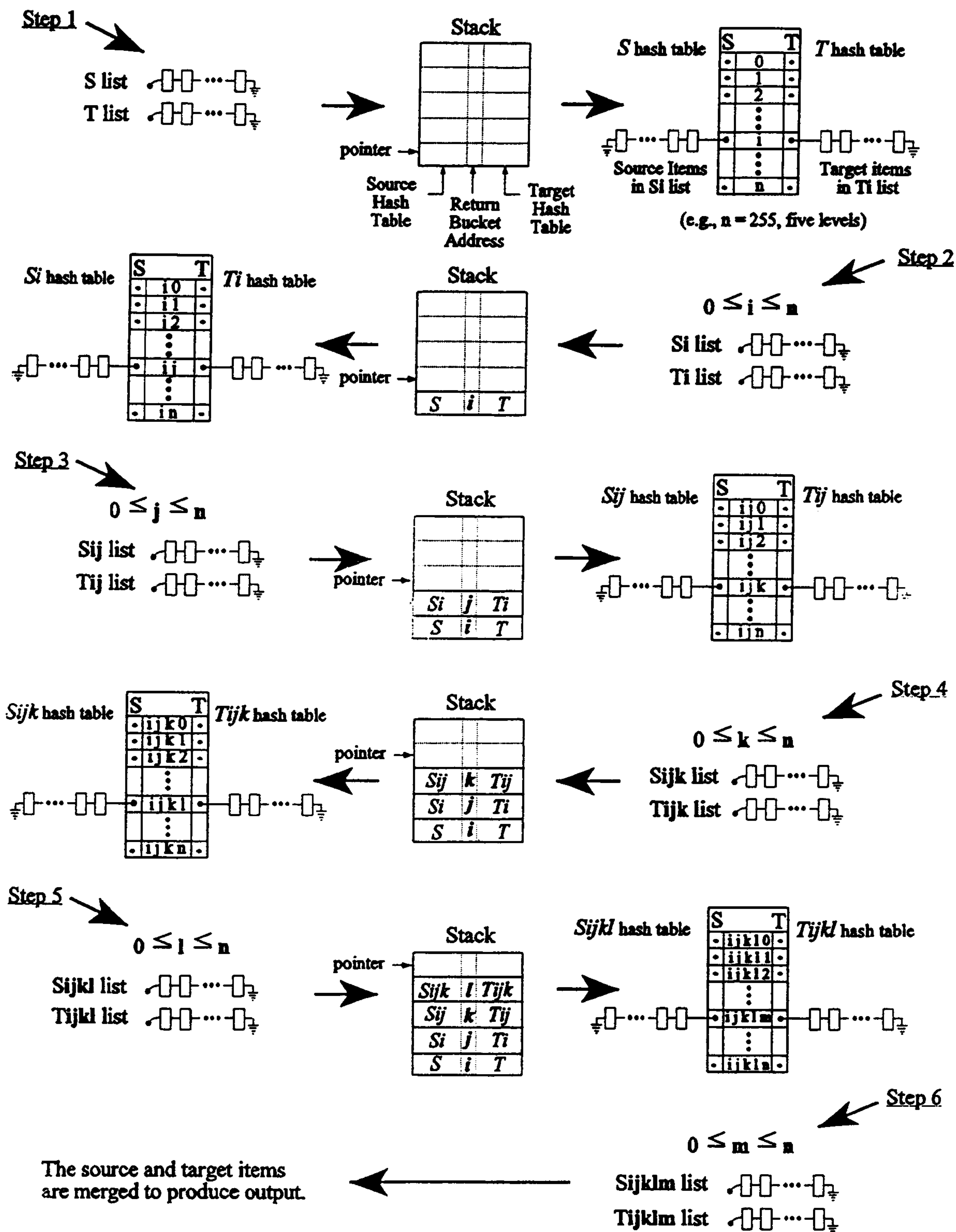



Figure 2: Shin's Algorithm for Massive Cross-Referencing

of source and target buckets is saved, and the source and target items in the pair of buckets are further divided by another functionally different hash coder. If a bucket is empty and the corresponding bucket in the pair is not, the items in the corresponding bucket are eliminated. The algorithm continues dividing the items in a pair of buckets, merging the items, or eliminating unnecessary items until every item in the buckets of created hash tables is either merged or eliminated.

The prime data structure used in Shin's algorithm is a stack. Each stack element consists of a return bucket address and a pair of two hash tables: one for the source items and the other for the target items. The stack pointer keeps track of the top element of the stack whenever a stack element is pushed into or popped up from the top of the stack. In the process of Shin's algorithm, several pairs of hash tables (e.g., a maximum of five) can be created. A source hash table includes a fixed number of bucket pointers (e.g., 256) for the linked lists of source items, while the target hash table includes the same amount (e.g., 256) of bucket pointers for the linked lists of target items.

One may choose proper numbers for the maximum level of the stack and size of the hash table. In this paper, five and 256 have been selected for the level of depth and table size respectively. As shown in step 1 in Figure 2, both source and target lists are divided into a maximum of 256 sublists for each list by the first hash coder. After the source and target items are hashed by the first hash coder, the items in the source bucket (Si) either match or don't match with only the items in the corresponding target bucket (Ti). If an empty bucket exists, all items in the corresponding bucket will be eliminated since they have no potential of being included in the resulting list.

As shown in step 2, Figure 2, the key values of the source items are hashed by the second functionally different hash coder. As a result, the source items are stored in addressed buckets in the source hash table. Using the same hash coder, the target items are hashed and stored in the target hash table. While the items are being divided into a maximum of 256 groups, the first produced hash address is compared with the subsequently produced hash addresses to see if they are the same. If so, the source items and target items are merged. Otherwise, four kinds of pairs of buckets (ij) may be created. The pairs will appear in the following combinations:

(1) Neither the source bucket (Sij) nor the target bucket (Tij) is empty.
(2) Sij is not empty, but Tij is empty.
(3) Tij is not empty, but Sij is empty.
(4) Sij and Tij are both empty.

When one of the two buckets is empty as in cases (2) and (3), the items in the corresponding bucket are unnecessary and therefore filtered out. Shin's algorithm provides a termination condition that ends further division processes. Whenever the items in the pair of source bucket and target bucket are divided by a hash coder, the algorithm checks for the termination condition. If the produced hash addresses in a group of source and target items are identical, the termination condition is satisfied. Then the algorithm stops dividing the group and starts merging the source and target items.

In a parallel architecture, multiple functionally different hash coders (e.g., a maximum of five) may be employed in checking the termination condition. If their logical ANDed results show that only a single hash address is produced from each involved hash coder, the group of source and target items can be merged without final screening. In order to eliminate 100 percent (i.e., greater than 99.9999999999% which is equal to $1-(1/256^5)$) of the unnecessary data, keys have to be hashed by a maximum of five functionally different hash coders to make certain that all produced hash addresses are the same. Therefore, two kinds of software implementation of Shin's join algorithm is left to one's choice: multiple hashings for each key at a time or a single hashing in each reading of a key. If one uses the latter for his software implementation, the filtering effect reaches 99.609375% (i.e., greater than 255/256) while a final screening process is needed for the merge.

Shin's algorithm proceeds from the first pair of buckets (e.g., addressed 0) to the last pair of buckets (e.g., addressed 255), checking that both source and target buckets are not empty. If neither buckets are empty, the next (return) bucket address (e.g., ***i***, ***ij***, ***ijk***, or ***ijkl*** in stacks of Figure 2) is saved and the items in the source bucket and the corresponding target bucket are rehashed (or divided) by the next functionally different hash coder. During the rehashing process, the algorithm compares the first produced hash address with the others. If the produced hash addresses are identical, the items are merged; otherwise, the items are further divided by another functionally different hash coder. Steps 3, 4, and 5 in Figure 2 can be explained similarly. In step 6, no available hash coder is left and all unnecessary data have been filtered; therefore, the source and target items have been merged without being rehashed.

Shin's algorithm requires a fixed number (e.g., a maximum of five) of readings for each key to determine whether the associate item is necessary or not. In Shin's algorithm, the number of visits to the buckets is proportional to the number of items even in the worst case; therefore, the time complexity of the algorithm is O(N) and traversing the buckets in the hash tables causes no problem. The time complexity of the algorithm for massive cross-referencing cannot be better than O(N) because the key in every item must be read at least once. The Shin's algorithm does not require another algorithm or a method to employ the divide and conquer strategy; thus, the algorithm is straightforward and

simple. A simulation program [12] for the algorithm was written by the author and was successfully executed. The following sections will explain where and how Shin's algorithm can be implemented.

## 3 Software Level

At the software level, parallel execution of Shin's algorithm can be performed by four types of processes: the hash process of source keys, hash process of target keys, filter processes, and merge process(es), as shown in Figure 3. The parallel execution of the algorithm at the software level can be divided into the hash phase, filter phase, and merge phase. The filter phase and the merge phase occur concurrently. In the hash phase, hash processes of source keys and target keys can be concurrently executed to complete a source hash table and a target hash table. After all keys in source and target lists are hashed in the hash phase, the filter phase starts. Every filter process within a group eliminates unnecessary items in a bucket and repeatedly rehashes their keys as necessary. Whenever a filter process finds a list of source items and a list of target items that are necessary, it sends the pair of lists to the merge process for final screening and merging. These processes must not interfere with each other. Thus, an interprocess communication mechanism (e.g., message queues or a shared memory) is needed.

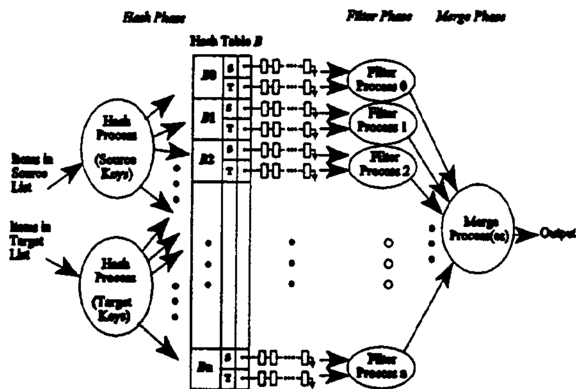



Figure 3: Parallel execution of Shin's algorithm at Software level

## 4 Processor Level

In the filter phase, unnecessary items in each pair of source and target buckets can be eliminated by identical filter processors. The filter processors can be either software or hardware filters. Figure 4 illustrates the general block diagram for processor configuration. In this structure, there are three functionally different processors: the hash processor, the filter processor, and the merge processor. The hash processors hash keys; the filter processors eliminate unnecessary items and pass necessary items to the merge processor; and the merge processor receives necessary items from the filter processors and merges them with final screening. Thus, the speed of output will not heavily depend on the bandwidth of the data path between the filter processors and the merge processor because almost all unnecessary items have already been eliminated.

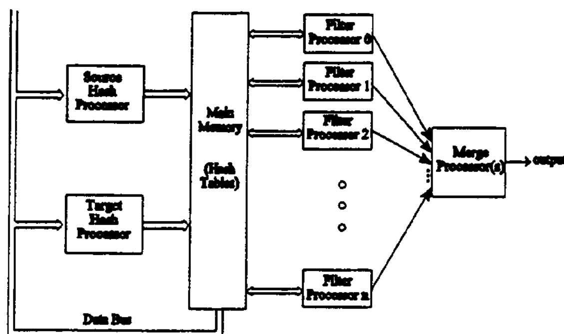



Figure 4: Processor Configuration

The general block diagram can stand for both the software and hardware filter processors. The next section will explain this in more detail.

### *4.1 Software Filter Processors*

Usually, general purpose processors are used for software filter processors. The programs that implement the Shin's algorithm are simply duplicated in all the filter processors, and they are executed according to command signals received by the processors. Each software filter processor (e.g., processor i) may be assigned to a bucket (e.g., Si and Ti) one at a time. The software filter may create another pair of hash tables (e.g., Sij and Tij) and rehash the keys of the items in the assigned bucket. Then, it eliminates unnecessary items in a bucket whenever there is no item in the corresponding bucket. It continues traversing buckets and creating a pair of hash tables as necessary until it finishes the traversal.

### *4.2 Hardware Filter Processors*

Hardware filter processors have progressive architectures which include multiple hardware hash coders such as Shin's (mapping) hash coder [12] and associated memory for hash tables. More than one hash coder will be used for subsequent rehashings. There is no relationship between the hash addresses of a key generated by functionally different Shin's hash coders. In this processor, effective

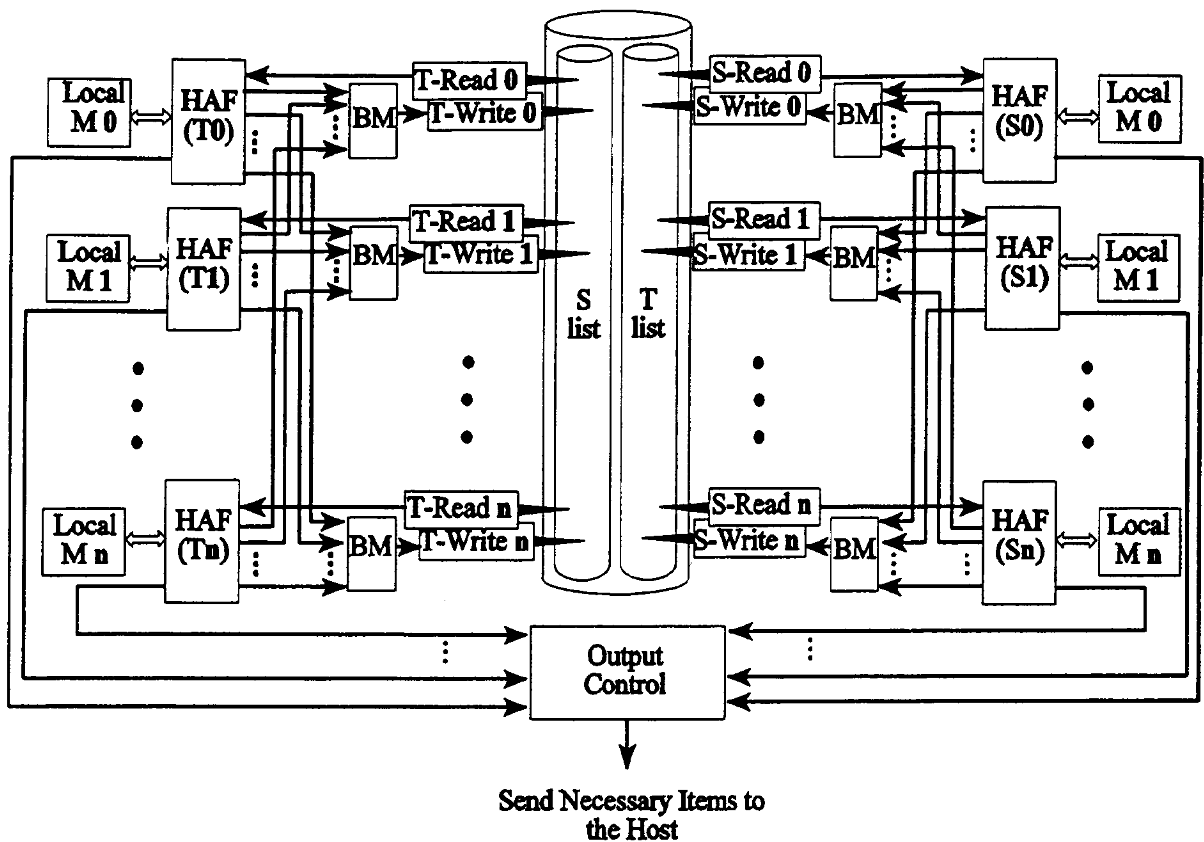



Figure 5: An Implementation of Shin's Algorithm at Auxiliary Storage Level for Maximum Parallel Execution

registers or RAMs are needed to implement indexes for hash tables. A special register is needed to load the first produced hash address and compare it with the subsequently produced hash addresses. The design details will be provided in a subsequent paper.

One may also consider associative memories for parallel filtering in Shin's algorithm. In this case, the architecture of every filter processor element is similar to the architecture of the hardware filter processor. The associative memory usually has a slow staging process; thus, a conventional memory scheme should be employed together with the associative memory to speed up the staging process.

## 5 Auxiliary Storage Level

Using the current algorithms such as nested-loop, sort-merge, and hash [6, 13], one may find it almost impossible to provide a solution for massive cross-referencing at the auxiliary storage level. Those algorithms require massive key comparisons, but unfortunately ordinary read/write functions in auxiliary storage devices cannot perform such complex operations. Figure 5 shows how Shin's algorithm can be implemented at the auxiliary storage level to solve this problem. First, the device should be equipped with intelligent hashing capabilities that produce a hash address from the reading of a key and the storing of items in buckets in either disk storage or local memory. The hash and filter processors (HAFs in Figure 5) perform hashing and staging items. When the local memory space is not sufficient to hold the items, the items are partitioned by their hash addresses and stored in subset files in the auxiliary storage. After the hashing and staging process, associated HAFs from each bucket in each subset file or local memory start filtering items in parallel. After the filtering process, only the wanted items are sent to the host processor for final screening and merging. The hashing capability may provide several hash addresses in each reading of a key by several functionally different hash coders. If subset files are being created in performing the Shin's algorithm, source and target files are being read by read headers (S-Reads and T-Reads in Figure 5) while the subset files are being written by write headers (W-Writes and T-Writes) with aid of buffer modules (BMs in Figure 5).

Multiple readers can read groups of items in a source file and a target file in parallel. More write headers are needed to write items in buckets in parallel.

Since other current algorithms have to compare source keys with target keys, they have to move every item into the main memory to perform massive cross-referencing. However, in Shin's algorithm, keys are compared to one another only after almost all unnecessary items are filtered out. In the filtering process, the algorithm repeatedly uses a simple hash operation without requiring complex operations. Thus, the parallel architecture of Shin's algorithm at the auxiliary storage level is simpler and more feasible than those of other current algorithms for massive cross-referencing. One may have noticed that the auxiliary filter device which implements the Shin's algorithm requires an architecture as complex as the architecture of a hardware filter processor and its operation isn't as fast as on-the-fly operations.

## 6 Relationship to Another Problem

When one needs to produce a list of duplicate items or a list of distinct items from an input list using a specified key component(s) in every item, the best solution to the problem of redundancy checking is the theory of redundancy checking. Shin's algorithm not only provides a basis for the theory of massive cross-referencing, but also provides a basis for the theory of redundancy checking. Although there is only a single input list in the problem of redundancy checking, Shin's algorithm can still be applied to provide a solution to the problem. Here, a single hash table (B), instead of a pair of hash tables (S and T), is created in each hashing process as shown in Figure 6. Thus, a stack element consists of a hash table (e.g., *B*, *Bi*, *Bij*, *Bijk*, and *Bijkl*) and a return bucket address (e.g., *i*, *ij*, *ijk*, and *ijkl*) as shown in Figure 6. A stack will contain a fixed number of stack elements. Thus, the traversal through buckets in the hash tables remains identical.

In the solution, there are two major kinds of bucket configuration; the bucket is either empty or not empty. Assuming that a bucket is represented as Bi (e.g., $B_0$, $B_1$, $B_2$, ..., $B_{255}$ in a hash table B), the combinations are as follows:

(1) Bucket Bi is empty.
(2) Bucket Bi is not empty:
(2.1) Bucket Bi has a single item.
(2.2) Bucket Bi has more than one item.

If bucket Bi is empty as in case (1), the algorithm will pass the bucket and go to the next bucket $B_{i+1}$. If bucket Bi is not empty (2) and has a single item as in case (2.1), there is no duplicate for this item in the input list, therefore, the algorithm will keep the item in the resulting list. Finally, case (2.2) shows that bucket Bi includes items of which duplications are not verified. Thus, the keys of the items are rehashed by

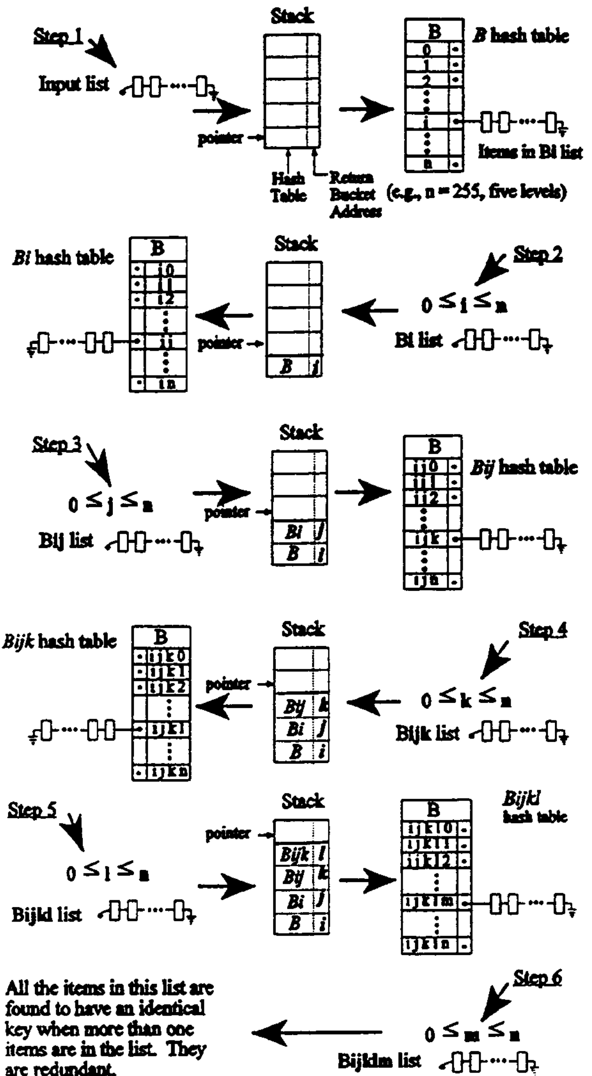



Figure 6: Shin's Algorithm for Redundancy Checking

another functionally different hash coder. If produced hash addresses are identical again, the items may be recorded as duplicate items in the redundancy list after a final screening. On the other hand, if the produced hash addresses are not identical, the algorithm will continue traversing buckets from the first bucket of a newly created hash table (*Bij*). If multiple functionally different hash coders produce multiple hash addresses, the algorithm may not need the final screening as explained in the second section of this paper.

The problem of redundancy checking can be converted into another problem which uses two identical input lists. The converted problem needs to produce a list of duplicate key items or a list of distinct key items as output. Then one can use the flow and principle of the Shin's algorithm, as shown in Figure 2, for the converted problem, adding an additional step to the final screening. In the final step, an item and its corresponding item in a pair of buckets are examined. If the corresponding item is the same item,

both are ignored. If different, their keys are compared to produce an output. Thus, the transformation that maps the problem of redundancy checking to the problem of massive cross-referencing provides the means for converting the Shin's algorithm that solves the problem of massive cross-referencing into a corresponding algorithm for solving the problem of redundancy checking. Therefore, the Shin's algorithm can be a solution for the problem of redundancy checking.

## 7 Conclusion

The Shin's algorithm for massive cross-referencing always uses a hash table with a fixed number of buckets (small in comparison to the hash algorithm). Thereby, the Shin's algorithm can divide items into a fixed number of buckets, allowing same amounts of processing elements to filter, in parallel, unnecessary items in the buckets. The fixed hash table size of the Shin's algorithm makes its parallel processing more efficient simplifying the software application and the hardware architecture. Furthermore, while the hash algorithm for massive cross-referencing cannot hash source and target keys in parallel, rather performing them serially one after another, Shin's algorithm can. Shin's algorithm also permits key comparisons only after almost all unnecessary items are filtered; therefore, the algorithm reduces the number of item transfers. The Shin's algorithm can also be carried out in a recursive routine. The characteristic of recursion in Shin's algorithm and the use of only a single algorithm in an implementation allow the filtering element to have a simpler and more adaptable architecture. Finally, to add to its desirability is the fact that it also can be applied to the problem of redundancy checking.

Shin's algorithm, because of its favorable inherent characteristic, can be broadly used for massive cross-referencing operations and join database operations. General design concepts for the application of Shin's algorithm for massive cross-referencing have been explained. Detail design and enhancement of the implementation of Shin's algorithm in various ways will be presented in the future.


## Acknowledgement

I would like to thank Chittoor Ramamoorthy, Arnold Meltzer, Domenico Ferrari, Diogenes Angelakos, Arie Segev, David Hodges,Steven Schwarz,Simon Berkovich, Beresford Parlett, David Messerschmitt, Manuel Blum, David Patterson, Michael Stonebraker, and Patrick Chang for their suggestions and feedbacks.


## References


[1] Abd-alla, A. M. and Meltzer, A. C. *Principles of Digital Computer Design. Vol. I*, Englewood Cliffs: Prentice Hall, 1976.

[2] Babb, E. "Implementing a Relational Database by Means of Specialized Hardware." *ACM Transactions on Database Systems*, Vol. 4, No. 1, Mar. 1979: 1-29.

[3] Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*, Vol. 13, No. 6, Jun. 1970: 377-87.

[4] Cooper, D. and Clancy, M. J. *Oh! Pascal! 2nd Ed.* New York: W. W. Norton, 1985.

[5] Date, C. J. *An Introduction to Database Systems*. Reading: Addison-Wesley, 1981.

[6] DeWitt, D. J. and Gerber, R. "Multiprocessor Hash-Based Join Algorithms." *Proceedings of the Eleventh International Conference on Very Large Data Bases*, Stockholm, 1985: 151-64.

[7] Horowitz, E. and Sahni, S. *Fundamentals of Computer Algorithms*. Rockville: Computer Science Press, Inc. 1978.

[8] Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*.San Francisco:W.H. Freeman and Company, 1986.

[9] Knuth, D. E. *The Art of Computer Programming Vol. III: Sorting and Searching*. Reading: Addison-Wesley, 1975.

[10] Patterson, D. A. and Hennessy, J. L. *Computer Organization and Design: The Hardware/ Software Interface*. San Francisco: M. Kaufmann, 1994.

[11] Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System." *Communications of the ACM*, Vol. 17, No. 7, Jul. 1974: 365-75.

[12] Shin, D. K. *A Comparative Study of Hash Functions for a New Hash-Based Relational Join Algorithm*. Pub. #91-23423, Ann Arbor: UMI Dissertation Information Service, 1991.

[13] Shin, D. K. and Meltzer, A. C. "A New Join Algorithm." *ACM SIGMOD RECORD*, Vol. 23, No. 4, Dec. 1994: 13-8.

[14] Stonebraker, M. R. "Operating System Support for Database Management." *Communications of the ACM*, Vol. 24, No.7, Jul. 1981: 412-18.

[15] Su, S. Y. W. *Database Computers*. New York: McGraw-Hill, 1988.

[16] Ullman, J. D. *Principles of Database Systems*. Rockville: Computer Science Press, 1982.