15. KAIST Presentation Viewgraphs Jul. 1, 1995

A New Join Algorithm

Dong Keun Shin and Arnold Charles Meltzer

Samsung Electronics Co., Ltd.
Communication Systems R&D Center
Songpa P.O.Box 117, Seoul, Korea

Department of Electrical Engineering and Computer Science
The School of Engineering and Applied Science
The George Washington University
Washington, D.C. 20052

Prepared for the Presentation in KAIST on July the 1st, 1995

OBJECTIVES IN THE RESEARCH

- Find a new efficient join algorithm (i.e., an optimal solution).
- Design an effective database computer which uses the new join algorithm.

RESULTS

- Shin's join algorithm is discovered.
- HIMOD-a database computer is designed.

WHY OUR RESEARCH ON THE JOIN WAS EXCITING?

- The join is a very useful operation in RDBMS.
- A frequently used operation.
- One of the most time consuming operations (i.e., Major bottleneck in RDBMS).
- The join operation might be included in future DBMS.
- Join theory can be used to solve cross referencing type of problem.

RELATIONAL OPERATIONS

- Project
- Select
- Join
- Union
- Intersect
- Difference
- Cartesian_Product

JOIN

- The join operation concatenates a tuple of the source relation (S) with a tuple of the target relation (T) if the value(s) of the join attribute(s) in this pair of tuples satisfy a pre-specified join condition, and it produces a tuple for the resulting relation (R).

Relation S

A	B	C
d	e	f
b	d	g
h	d	b

Relation T

D	E	F
d	g	a
a	d	c

```
SELECT  S.*  T.*
FROM    S, T
WHERE   S.B = T.E
```

```
JOIN  S, T (S.B = T.E)
```

Relation R

A	B	C	D	E	F
b	d	g	a	d	c
h	d	b	a	d	c

FOUR MAJOR JOIN ALGORITHMS

- The Nested-Loop Join Algorithm
- The Sort-Merge Join Algorithm
- The Hash Join Algorithm
- The Shin's Join Algorithm

• The Nested-Loop Join Algorithm

The two relations involved in the join operation are called the outer relation (or source relation) S and the inner relation (or target relation) T, respectively. Each tuple of the outer relation S is compared with tuples of the inner relation T over one or more join attributes. If the join condition is satisfied, a tuple of S is concatenated with a tuple of T to produce a tuple for the resulting relation R.

Relation S

A	B	C
d	e	f
b	d	g
h	d	b

Relation T

D	E	F
d	g	a
a	d	c

```
SELECT  S.*  T.*
FROM    S, T
WHERE   S.B = T.E
```

```
JOIN  S, T (S.B = T.E)
```

Relation R

A	B	C	D	E	F
b	d	g	a	d	c
h	d	b	a	d	c

• The Sort-Merge Join Algorithm

Each of the source (S) and target (T) relation is retrieved, and their tuples are sorted over one or more join attributes in subsequent phases using one of many sorting algorithms (e.g., n-way merge). After the completion of the sorting operation, the two sorted streams of tuples are merged together. During the merge operation, if a tuple of the relation S and a tuple of the relation T satisfy the join condition, they are concatenated to form a resulting tuple.

Relation S

A	B	C
d	e	f
b	d	g
h	d	b

Relation T

D	E	F
d	g	a
a	d	c

Relation S' (Sorted)

A	B	C
b	d	g
h	d	b
d	e	f

Relation T' (Sorted)

D	E	F
a	d	c
d	g	a

```
SELECT  S.*  T.*
FROM    S, T
WHERE   S.B = T.E
```

JOIN S, T (S.B = T.E)

Relation R

A	B	C	D	E	F
b	d	g	a	d	c
h	d	b	a	d	c

• The Hash Join Algorithm

The join attributes of the source relation (S) are first hashed by a hash function. The hashed values are used to address entries of a hash table called buckets. The same hash function is used for the join attributes of the target relation (T). If the join attribute of a tuple is hashed to a non-empty bucket of the hash table and one of the join attributes stored in that bucket matches with the join attribute, the equi-join condition is satisfied. The corresponding tuples of the S and T relations are concatenated to form a tuple of the resulting relation (R). The process continues until all the tuples of the target relation have been processed.

Relation S

A	B	C
d	e	f
b	d	g
h	d	b

Relation T

D	E	F
a	d	c
d	g	a

When h('Key') = a hash address,

```
SELECT  S.* T.*
FROM    S, T
WHERE   S.B = T.E
```

h('e') = 0
h('d') = 12
h('g') = 2.

JOIN S, T (S.B = T.E)

Relation R

A	B	C	D	E	F
b	d	g	a	d	c
h	d	b	a	d	c

• **The Shin's Join Algorithm**

In Shin's join algorithm, the source and target relations are repeatedly divided (or rehashed) by a maximum of five statistically independent hash functions until a group of source tuples and target tuples are found to have an identical join attribute. The source and target tuples in the group are merged after a final screening in order to produce resulting tuples.

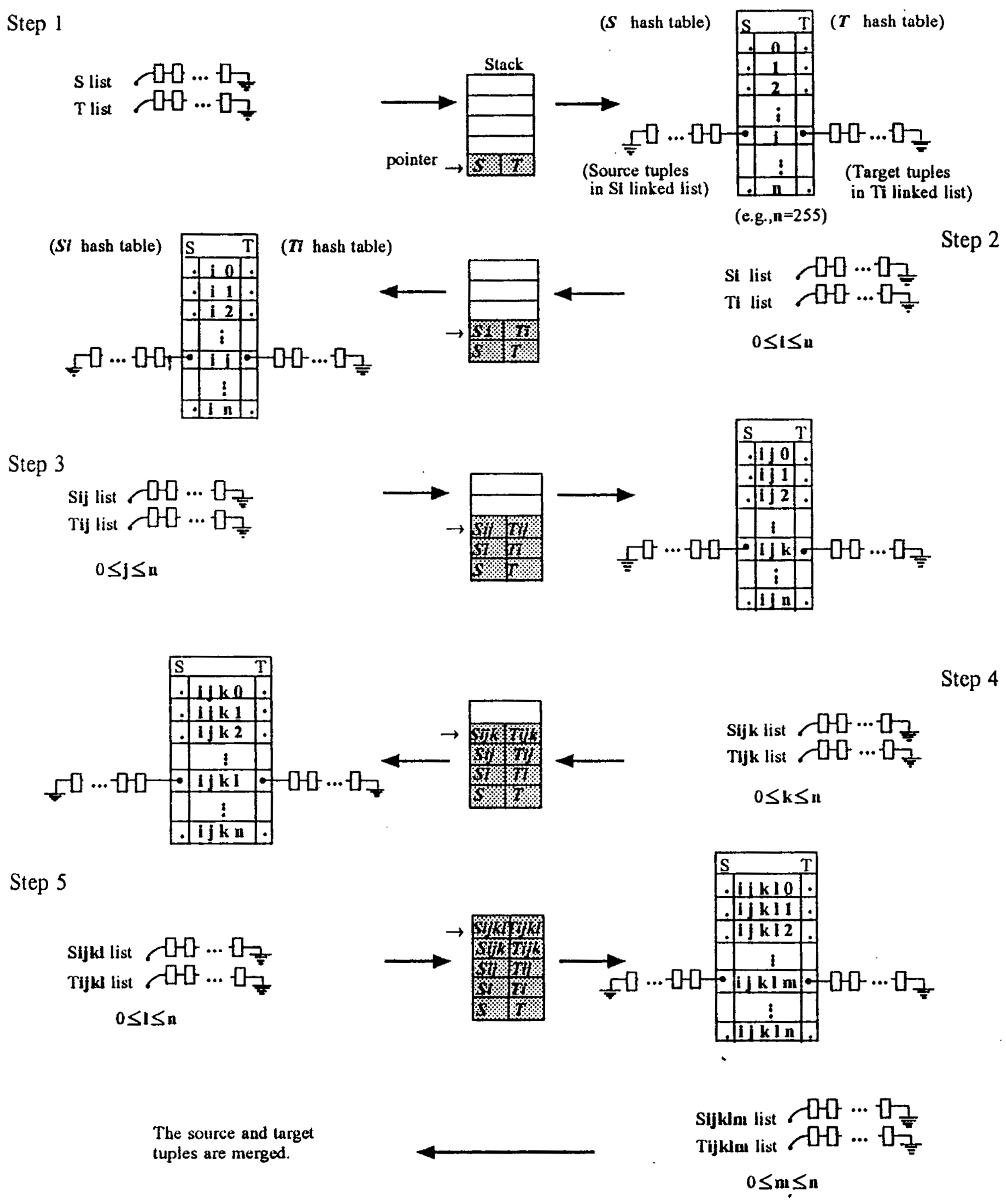

Figure 1. The SOFT and the Shin's Join Algorithm

```
begin
  Initialization;
  finish := false;
  repeat
    Hash_Source_And_Target_Relations;
    If Only_One_Hash_Address_Produced then
    begin
      Merge_Tuples_And_Output;
      If No_More_Next_Bucket_Addr then
      begin
        If Bottom_Of_Stack then
          finish := true
        else
        begin
          pop;
          if No_More_Next_Bucket_Addr then
          begin
            if Bottom_Of_Stack then
              finish := true
            else
            begin
              pop;
              if No_More_Next_Bucket_Addr then
              begin
                if Bottom_Of_Stack then
                  finish := true
                else
                begin
                  pop;
                  if No_More_Next_Bucket_Addr then
                  begin
                    if Bottom_Of_Stack then
                      finish := true
                    else
                    begin
                      pop;
                      if No_More_Next_Bucket_Addr then
                      begin
                        if Bottom_Of_Stack then
                          finish := true
                        else
                        begin
                          Assign_Source_And_Target;
                          Save_Next_Bucket_Addr;
                          push;
                        end;
                      end;
                    end;
                  end;
                  else
                  begin
                    Assign_Source_And_Target;
                    Save_Next_Bucket_Addr;
                    push;
                  end;
                end;
              end
              else
              begin
                Assign_Source_And_Target;
                Save_Next_Bucket_Addr;
                push;
              end;
            end;
          end
          else
          begin
            Assign_Source_And_Target;
            Save_Next_Bucket_Addr;
            push;
          end;
        end;
      end
      else
      begin
        Assign_Source_And_Target;
        Save_Next_Bucket_Addr;
        push;
      end;
    end
    else
    begin
      Assign_Source_And_Target;
      Save_Next_Bucket_Addr;
      push;
    end
  until finish;
end.
```

Figure 2. The Shin's Join Algorithm

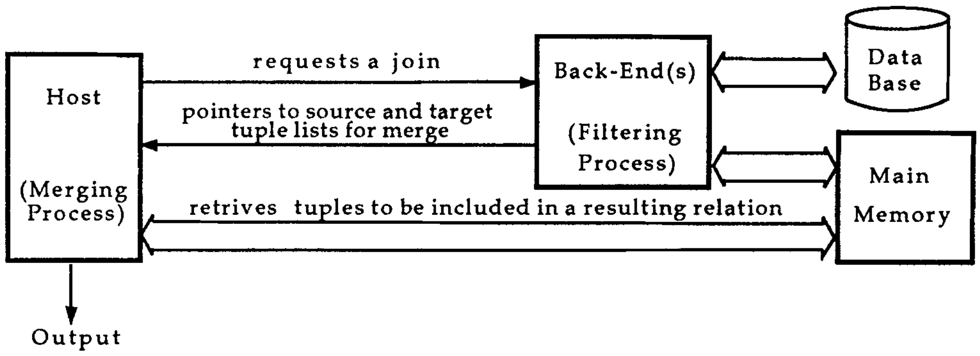
requests a join
Host
(Merging Process)
pointers to source and target tuple lists for merge
Back-End(s)
(Filtering Process)
Data Base
Main Memory
retrives tuples to be included in a resulting relation
Output

Figure 3. Execution of the Relational Join in HIMOD

TIME COMPLEXITY ANALYSIS

• The Nested-Loop Join Algorithm	O(N*N)
• The Sort-Merge Join Algorithm	O(N log N)
• The Hash Join Algorithm	O(N)
• The Shin's Join Algorithm	O(N)

PROBLEMS OF THE HASH JOIN ALGORITHM

- Hash Function Dependency
  The performance of the hash join algorithm is largely dependent on the distribution performance of a chosen hash function.

- Data-Size Dependency
  The performance is dependent on the ratio of hash table size to the number of source input tuples.

- Excessive Join Attribute Comparisons
  The join attribute comparisons include the comparisons for unnecessary tuples.

- No Filter Concept in the Algorithm.

- Insufficient Characteristic of Parallelism

ADVANTAGES OF THE SHIN'S JOIN ALGORITHM

- Less Hash Function Dependency than the Hash Join Algorithm

  My join algorithm uses several functionally different hash methods to reduce the dependency of a chosen hash function's distribution performance.

  Shin's mapping hash function is recommended.

- No Precalculation to Determine Hash Table Size

  Fixed size hash tables are used instead of a variable size hash table.

- Better in Hardware Implementation

  It is easy to implement in hardware since hash tables are fixed size.

- No Unnecessary Join Attribute Comparison

  It allows join attribute comparison after more than 99% of the unnecessary tuples are eliminated.
  Other join algorithms rely heavily on tedious join attribute comparisons.

- Inherent Characteristic of Parallelism

  The Shin's join has two major processes: the filter process and the merge process. These processes can be executed in parallel and they also can be divided into subprocesses to run concurrently.

  Hashing process can be divided into multiple subprocesses while the hash join algorithm cannot easily provide the parallelism in hashing due to frequent memory lockings.

CONCLUSION

Question: What is the optimal algorithmic solution for the join?

Answer: Need to conduct more experiments to compare the hash join to mine.

We believe the Shin's join algorithm is superior to the hash join algorithm not only because of the aforementioned problems that the hash join algorithm has but also because of the advantages of my join algorithm.

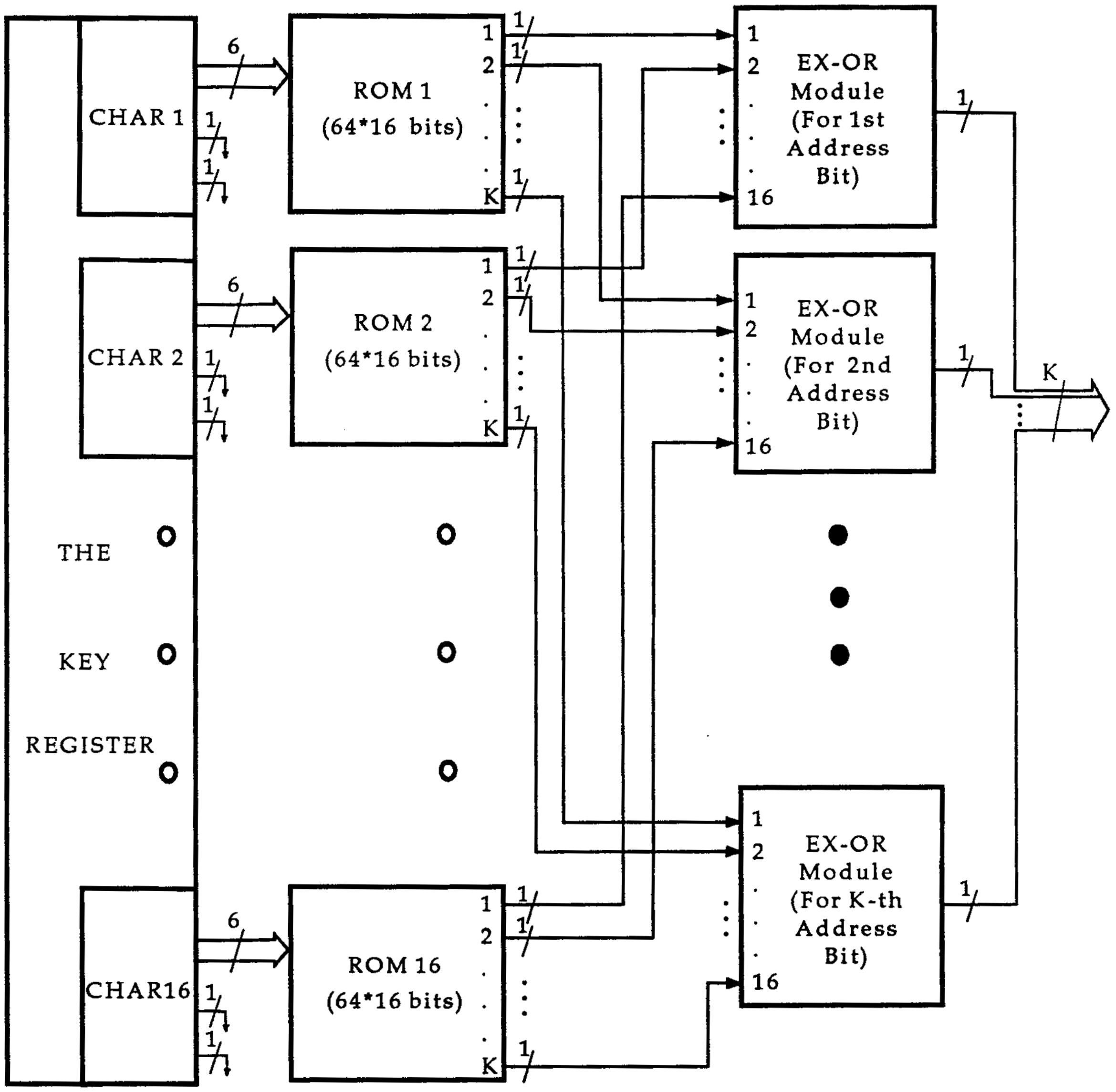

Figure 4. Hardware Mapping Hash Coder

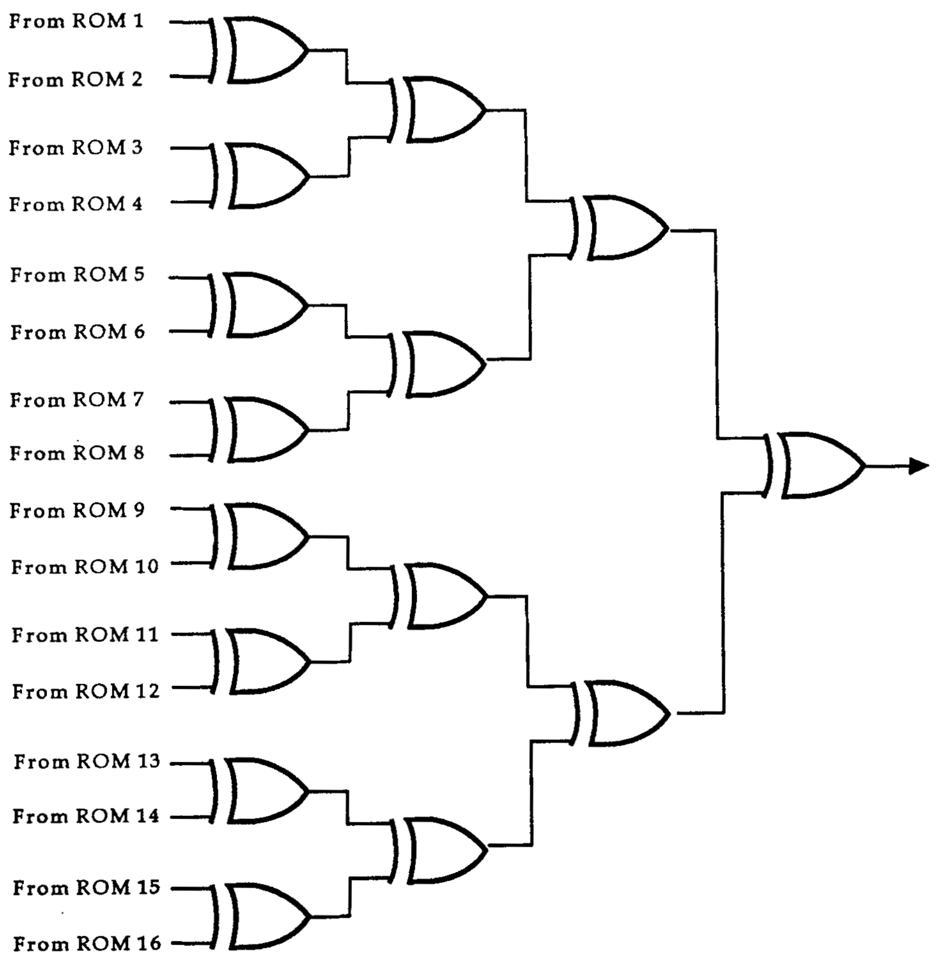

Figure 5. Exclusive-OR Moduel for a Hash Address Bit

```
const
        MAX_NO_CHARS_IN_KEY = 16; {number of characters in a key}
        MAX_NO_BUCKETS = 256; {number of buckets in the hash table}
        NO_PRIMES_IN_ROM = 64; {number of prime number in each ROM}

type
        {Type for the array of 16 characters key}
        Key_Array_Type = array [1..MAX_NO_CHARS_IN_KEY] of char;

var
        {Array table of 64 prime numbers for each ROM}
        Prime_Table : array [1..MAX_NO_CHARS_IN_KEY, 0..NO_PRIMES_IN_ROM-1]
                        of char;

function Mapping_Hash (Key : Key_Array_Type) : integer;
var
        Temp, Char_No, Index : integer;
begin
        Temp := 0;
        for Char_No := 1 to MAX_NO_CHARS_IN_KEY do
        begin
                Index := ord(Key[Char_No]);
                if Index >= NO_PRIMES_IN_ROM then
                        Index := Index - NO_PRIMES_IN_ROM;
                Temp := EX_OR(Prime_Table[Char_No,Index],Temp);
        end;
        Mapping_Hash := Temp mod MAX_NO_BUCKETS;
end;
```

Figure 6. Shin's Mapping Hash Function