Hardware Assisted Sorting in IBM's DB2 DBMS

Balakrishna R Iyer

IBM Silicon Valley Lab
555 Bailey Avenue
San Jose, CA 95141
USA
balaiyer@us.ibm.com

Abstract

For over 20 years, researchers have experimented with hardware innovations to make RDBMS processing more efficient. Efforts to build database machines with special hardware may be contrasted with the introduction of assists for accelerating database processing in general purpose processors. Four CPU intensive DBMS functions are identified in this paper, and all four have been successfully offloaded to assists on processors by IBM's DB2 DBMS. One such assist, the sort assist, is examined in detail. Offset Value Codes as implemented by this assist and its exploitation in DB2 are described.

1. Introduction

Using hardware to improve RDBMS processing efficiency is an idea that has been explored since the early 80s [TERA83] [UBEL85] [IDM85] when multiple efforts began to build specialized machines for database processing. The field of database machines met the same fate as the field of specialized scientific supercomputers. Teams started building specialized computers based on the current best hardware technology. By the time the design was completed, the machines built, software written, hardware and software tested and deployed, and applications written for it, the special purpose machines, built from now slightly older hardware technology, did not have sufficient business advantage over database processing on the latest general purpose processors built from the latest hardware technology. Only one of the database machine projects from the early 80s [TERA83], the one that successfully changed design to exploit not special purpose but general purpose hardware [BALL94], has remained commercially viable.

International Conference on Management of Data
COMAD 2005b, Hyderabad, India, December 20-22, 2005.

At IBM's research and development labs, we have successfully pursued an alternate approach to leverage hardware technology for achieving RDBMS processing efficiency. We accepted the maxim that processing technology will continue to offer ever increasing efficiencies that would be hard to compete with. We decided to leverage processor advances by identifying several compute intensive functions in RDBMS processing, designed and implemented hardware assists within IBM's general purpose z/Architecture processor, re-architected IBM's DB2 DBMS to offload CPU processing to these hardware assists. Each assist mentioned in this paper, along with it's DB2 exploitation has already been implemented, commercialized, and is in active use world wide.

The four generic functions that consume significant CPU cycles in DBMS processing we identified and successfully accelerated with hardware processor assists are:

1) Sorting,
2) Compression [IYER94] [CHAN96] [BRUI98],
3) Encryption [HACI2002] [SLEG2004] [BERG2005],
4) Concurrency and Coherency control for shared everything DBMS clustering [DIAS87] [JOST97] [BOWE97].

In this paper we will describe the sort assist and it's exploitation in DB2. Note that, to simplify treatment, we will assume the input keys to be sorted are duplicate-free. The techniques described here extend in a straightforward way and correctly handle duplicates and it is this extended method that is supported by the DB2 DBMS. We argue the importance of sorting for DBMS processing in Section 2. To establish the base understanding needed to follow the rest of the paper, replacement/selection tournament tree based sorting is illustrated by an example in Section 3. Offset Value Codes are defined in Section 4 and two important properties of offset value codes stated and proved. The tournament tree sort example of Section 3 is

modified to incorporate offset value coding in Section 5. Section 6 contains the formal algorithm for Offset Value Code based Tournament Tree based replacement/selection sorting. IBM's sort assist is described in Section 7 and conclusions appear in Section 8.

2. Need for Sorting in RDBMS

There are many reasons why a relational database manager needs to sort data. For example, results may need to be ordered due to an SQL ORDERBY clause. Some SQL queries require results to be grouped per the semantics of the SQL GROUPBY operator and sorting is one way to implement grouping. The SQL DISTINCT operator, and the UNION operator, both may require duplicate elimination. Sorting is one method to detect and eliminate duplicates in DB2. DB2's query optimizer may choose the sort merge join method to join two tables. The sort merge join algorithm requires the two tables to be ordered by their join key. This is often done by sorting. Some query plans may call for the access of an index for a large set of rows. The index is efficiently accessed if the large set of rows is sorted first on the same key as the index and query optimizers may introduce a sort step for this purpose. DB2's Index AND/ORing and Hybrid Join methods [CHEN91] involve the extraction of row identifiers to qualifying rows from one or more indexes and sorting on these row identifiers. Pre-computing aggregates could also invoke sorting. A host of utility operations from index creation to tablespace reorganization (recovers clustering) invoke sort operations.

The time complexity of sorting is O(n log n) where n is the number of rows sorted. In a truly scalable DBMS, sorting is, perhaps, the only function whose CPU cost may theoretically scale faster than the number of rows. Over a wide spectrum of customer workloads, we found sort to be highly consumptive of the CPU and a prime candidate for acceleration via a hardware assist.

3. Tournament Tree Sort in DB2

Various commercial RDBMS have implemented various sorting algorithms. DB2 itself features more than one sort algorithm – an internal in-memory radix sort, an external tag sort and a tournament tree based selection/replacement sort. The tournament tree sort, used for query processing, leverages hardware and will be the focus of our discussion.

DB2's selection/replacement sort is implemented by means of a 'tournament tree' [KNUT73] also known as a 'loser tree'. The simple, yet elegant algorithm is comprised of three steps. They are: 1) Tree Initialization, 2) Run Formation, and 3) Tree Flush. We describe each step through example.

In the first step, a fixed number of rows are brought into pre-allocated memory. Sort keys are extracted from the rows and organized as a tournament tree illustrated in Figure 1a. A tournament tree or loser tree is a multi-level binary tree. Leaf nodes are labelled by the keys to be sorted. Keys in adjacent nodes (those with a common parent) are compared. The parent of two leaf nodes points to the loser of the comparison of the keys of it's two children. (Wlog we assume that we are sorting in ascending sequence and a key of lower value "wins" over a key of larger value.) Other ancestor nodes have two sub trees. Winning keys from each of the two sub trees are compared. The ancestor node points to leaf node supplying the losing key of this comparison. A fully annotated tournament tree is shown in Figure 1a. Note that the smallest key, key 44 is the overall winner.

In step 2 of the algorithm, Key 44, the winner is removed from the tree and written into a sorted run. It is replaced by the next key from the list to be sorted in this run, key 53.

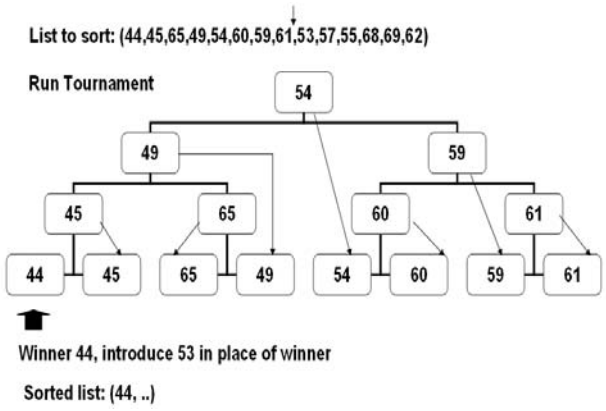

Figure 1a: Initial Population of Tournament Tree
Identification of first winner

A winner path is the set of ancestor nodes of the winning key starting from the bottom ending at the top. The winner path for key 44 in Figure 1a is shown in Figure 1b. A very important observation is that key 53 needs to be compared only with keys in the winner path of the key it replaced, key 44. That is because these nodes, at each level point to the smallest key contained in the sub tree not containing key 53. If key 53 loses at any node (along the winner path), that key, to which it loses, is carried up the winner path. In turn, if it loses at an ancestor node, the key it lost to is carried upwards along the winner path. It is critical to observe that the task of finding the next winner involves only comparisons of keys in the winner path. The result of the replacement of key 44 by key 53 and the key comparisons along the winner path is shown in Figure 2.

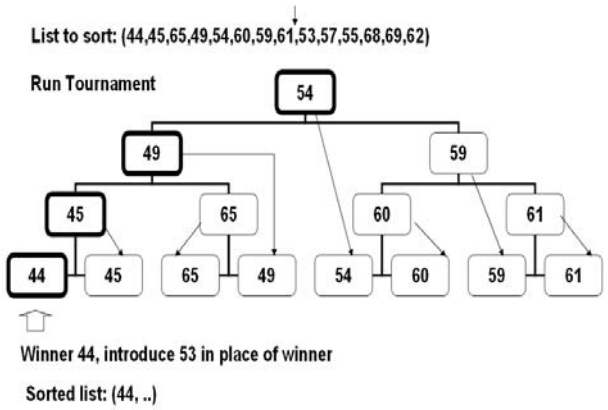

Figure 1b: Winner path
Highlighted Boxes - Winner Path for winning key 44

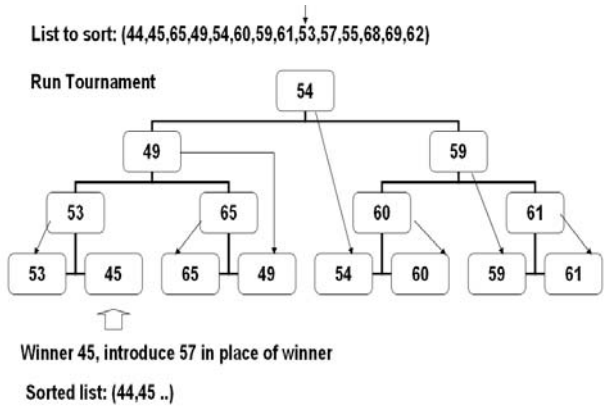

Figure 2: Tournament Tree after extraction of first winner
Identification of second winner

Key 45 is the winner and joins the sorted run (44,45,..). Note, that if we continue this process to conclusion, the sorted run will have at least as many keys as the number of leaf nodes in the tree (which is the same as the number of rows in pre-allocated memory). A newly inserted key can not participate in the current run if it is greater than the winner it replaces. By pre-pending every key with a run number and incrementing the run number when a key belonging to the next run is inserted it is possible to continually process the input sequence. Winners emerging from the tree will be examined for their run number and appropriately separated. If the input is random, it has been shown that the average run length is approximately twice the number of leaf nodes in the tournament tree [KNUT73]. This is a critical observation because tree size is determined by the number of rows that can be fit into the space available in memory. Most sort algorithms produce a run of memory size. In a typical RDBMS, we worry about sorts that require runs to be merged. Memory is used to hold the merge data structures and enough pre-fetch buffers and deferred write buffers needed to support asynchronous bulk read of the input sort runs and asynchronous bulk write of the merged runs. Merge width is limited by the amount of memory one can dedicate to merging. Longer sort runs reduce the number of sort runs that need to be merged and may reduce the number of merge passes needed.

Step 3 begins when the last key to be sorted has been inserted into the tree, the tree still contains keys that need to be written out into the sorted run. These keys need to be flushed out. We insert a sequence of "infinity" keys, i.e., keys that sort higher than any key in the input. Infinity keys can be simulated by pre-pending the key with a run number greater than any run number that could occur in the sorted run. Insertion of the infinity keys effectively flushes out the remaining keys in the tournament tree. We examine each winner from the tournament tree and terminate the algorithm when the first infinity key emerges as winner.

4. Offset Value Coding

Most research in sorting has focused on the time complexity of sorting measured as a function of the number of key comparisons involved in sorting. While in scientific applications, sort keys may be integers or floating point numbers, in real database applications, especially for supporting query processing, sort keys upwards of 15 bytes in length are quite common. Database sorting time complexity is O(kn log n) where k is the average length of the keys being compared. There is relatively little research addressing the cost of key comparison. Offset Value Coding is a technique that was invented (CONN77) to attack key comparison cost.

Using offset value coding, it is possible to use cheaper comparisons, like integer compares rather than full key comparisons of long keys to determine the winner. Most computers have fast integer units. In fact, many may have more than one integer unit. By reducing the need to access the full key, the technique also makes better use of the processor cache and indirectly increases the effectiveness of the processor. Offset value coding, between two keys, codes the position at which the two keys differ and the value of the larger key at that position. A more formal definition follows.

Let key A(n) be a concatenation of characters A(1), A(2), …, A(n) where 1 denotes the high order or most significant position. Let B(n) be a second key. An offset value code is defined for non-equal keys . The offset value code of B wrt A, OVC(B,A) is the concatenation of the pair, offset O(B,A) and value code VC(B,A), where offset O(B,A) is p the smallest integer not less than 1 for which A(p) does not equal B(p) and VC(p) is the complement of B(p).

For example, OVC(421,336) = (1,5), and OVC(421,423) = (3,8), assuming that the occupier of each position is drawn from the domain of single digit non-

negative integers. All the examples in this paper will assume this domain. OVCs, however, may also be defined over bytes, double bytes or any character set for which order and complement are well defined.

Next, we give two theorems first stated by Conner (CONN77). Proofs for the two theorems, which are not found in prior literature, are included below.

Unequal Code Theorem for Offset Value Codes

When B > A, C > A, and OVC(B,A) > OVC(C, A)
then B < C and OVC(C,B) = OVC(C, A)

Proof:

We are given that OVC(B,A) > OVC(C,A)
Writing out the two components of OVC,
((O(B,A),VC(B,A)) > ((O(C,A),VC(C,A))

This can happen in only two ways as follows:
i) O(B,A) > O(C,A)
ii) O(B,A) = O(C,A) and VC(B,A) > VC(C,A)

We examine these cases separately,

Case i): O(B,A) > O(C,A)
In positions 1 through O(C,A) - 1 A, B, and C are identical.
In position O(C,A) A and B are identical.
In position O(C,A) A and C are different.
C > A is given, hence in position O(C,A) C has a higher value than A as well as B
Hence C > B, which is one of the parts to be proved.

B and C are different at position O(C,A)
Hence OVC(C, B) = (O(C,A),comp(C(O(C,A)))
= OVC(C,A), which is the other part to be proved for this case.

Case ii): O(B,A) = O(C,A) and VC(B,A) > VC(C, A)
A, B and C are identical in positions 1 through O(B,A)-1 (also O(C,A)-1)

VC(B,A) = comp(B(O(B,A))
VC(C,A) = comp(C(O(C,A)) = comp(C(O(B,A))
VC(B,A) > VC(C,A) … given
Hence, comp(B(O(B,A))) > comp(C(O(B,A)))
Thus, B(O(B,A)) < C(O(B,A)).

At the highest order position where B and C are different from A, the value in B is smaller than the value in C.

Hence, B < C, which is one of the parts to be proved for this case.

This also shows that B and C are different at the highest position where A and B are different from C.

In other words O(C,B) = O(C, A) = O(B,A)
OVC(C,A) = (O(C,A),comp(C(O(C,A))))
= (O(C,B),comp(C(O(C,B))))
= OVC(C,B)

The theorem shows that two keys compare inversely as their offset value codes against a common smaller key. This is how offset value codes are used to avoid key comparison. However, two keys could be different but their offset value codes may be the same. The theorem also states the conditions under which the offset value code of a larger valued key against a lesser valued key is the same as the offset value code of the larger valued key against an intermediate valued key. This property will be used to avoid evaluating the offset value code, whenever possible, during sorting. This is important because computing offset value codes between two keys involves key comparisons, and we would like to minimize full key comparisons.

Next, we give the Equal Code Theorem for Offset Value Codes.

Equal Code Theorem for Offset Value Codes

When B > A, C > A and OVC(B,A) = OVC(C,A) then O(B,C) > O(B,A).

Proof:

OVC(B,A) = OVC(C, A) … given
Therefore the offset components of the two codes are equal, O(B,A) = O(C,A).
Hence, A, B and C are identical from positions 1 through O(B,A)-1 (also O(C,A)-1).
The value components are also equal, VC(B,A)=VC(C,A).
Hence B and C are identical in position O(B,A) as well. If they differ they must differ first in a lower order position.

Thus O(B,C) > O(B,A)

The Theorem states that when offset value codes of two keys against a common smaller key are equal, we can not determine the result of the comparison of the two keys be comparing offset value codes. We need to examine less significant positions, positions beyond the offset value of the offset value code. Higher order positions of the two keys need not be examined since they are identical.

5. Tournament Tree sorting with OVC

Next, we explain how offset value codes can be incorporated in tournament tree based replacement/selection sorting. Before going further, we remind the reader of one important property of tournament tree sorting. A winner establishes a winner's path in the tournament tree. When a winner is replaced with a new key from the input sequence, the task to find the next winner only involves comparisons of the OVC of the new key computed against the most recent winner and OVCs of keys in the winner's path. We will show the OVC based tournament tree replacement/selection sort algorithm will leave the winner's path annotated with the OVC of the keys in the path against the most recent winner key. We illustrate that whenever two OVCs against the most recent winner are unequal, we don't have to compare their corresponding keys. When the two OVCs are equal, we need to compare the two keys and compute the OVC of the larger key against the smaller one. By using an example, we will next illustrate the algorithm in Figures 3a, 3b, 3c and 3d.

Tournament Tree Replacement/Selection Sort with Offset Value Coding has three steps analogous to the Tournament Tree Replacement/Selection Sort Algorithm we illustrated in Figures 1 and 2. The steps are: 1) Initialisation of the Tree with Key values and Offset Value Codes, 2) Run Formation, and 3) Tree Flush.

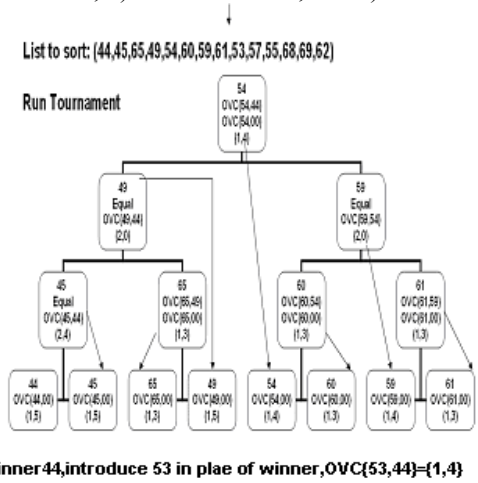

Figure 3a: Example of Tournament Sort with OVC Tree Initialisation

In the first step, the tournament tree is initialised with the same key values as the tree in Figure 1. In addition, the tree is annotated with the offset value code of each key against the 'min key'. The min key is a key that collates lower than any key in the tree. A min key is easily simulated by pre-pending any key with a run number lower than any valid run number produced by the tree. In this example, we are sorting positive two digit integers and we will assume 00 to be the min key. The tree initialisation algorithm proceeds as follows. For each key in the leaf of the tree, an offset value code is calculated for the key against the min key 00. At each parent of two leaf nodes, OVC's of it's two children are compared. If the two offset value codes are unequal, the Unequal Offset Value Code theorem is applied, the key with the higher offset value code is chosen as the winner, the key with the lower offset value code as the loser. A pointer to the leaf with the losing key is written into the parent node. The theorem further states that the offset value code of the loser against the winner is the same as the offset value code of the loser against the minkey. That OVC code is left in the parent node. If the two OVC codes of the two leaf keys are identical, then the Equal Offset Value Code theorem applies. OVC code comparison can not determine the winner. The two keys are examined starting from the position indicated in the OVC code to detect a difference. The lower key is declared the winner, an offset value code is computed for the loser against the winner and left in the parent node. The parent node is also made to point to the leaf with the loser key. For the other non-leaf nodes, when both children have been labelled with loser keys, their corresponding winner keys are similarly compared at the parent node through OVC comparison. A pointer to the leaf containing the losing key and the loser's OVC remains at the parent node, the process is repeated for each ancestor until we exhaust all parent nodes. At this time the smallest key in the tree has been identified as the winning key at the root node and step one is complete.

We will explain two of the comparisons shown in Figure 3a. As an example, let us consider the two sibling leaves labelled with keys 54 and 60 and their OVCs against min key 00. OVC(54,00) = (1,4), and OVC(60,00) = (1,3). If we set B=54, C=60, A=00, we realize OVC(B,A) > OVC(C, A) and the Unequal Offset Value Code theorem assures us that B < C and OVC(C,B) = OVC(C, A). Hence we have compared B against C just by comparing their offset value codes and without comparing the two keys B and C themselves. The theorem also tells us that OVC(60,54) is the same as OVC(60,00). We leave this OVC at the parent node along with a pointer to the leaf containing key 60. Observe there was no need to recompute the OVC.

As a second example of a comparison in Figure 3a, consider the node, left child of the root, labelled with key 49. At this node, key 44, winner from the left sub tree is compared with key 49, winner from the right sub tree. Both keys come with their OVC's against min key 00. OVC(44,00) = (1,5) and OVC(49,00)=(1,5). Identifying B as key 44, C as key 49, A as key 00, we can apply the Equal Code Offset Value Code theorem. To compare the two keys, we have to examine position 2 of the keys. We then find B to be the winner, and also

compute OVC(C,B) = (2,0). We leave this OVC and a pointer to the leaf containing key C, 49.

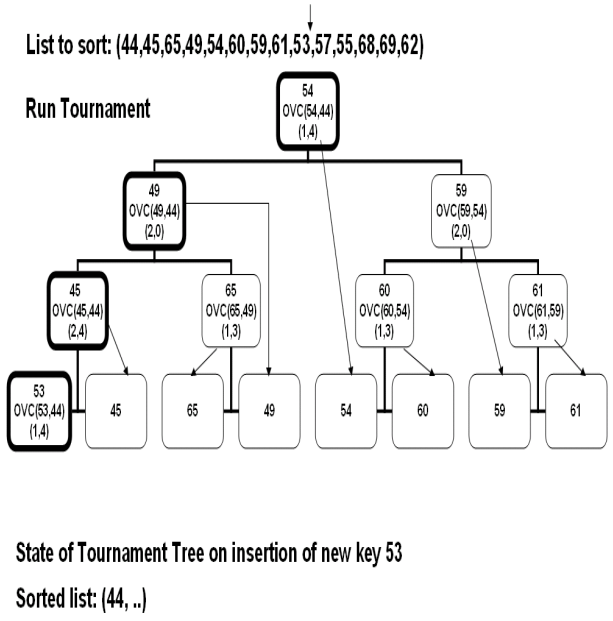

Figure 3b: Example of Tournament Sort with OVC Run Formation – First key replacement

Once the tournament tree is so initialised we go the second step, the run formation step. As illustrated in Figure 3a, the first step populated the tree and identified the first winner, key 44. Key 44 is written out as the first element of the first sorted run. The second step is an iterative step and repeats as long as there are input keys to be sorted. The winning key 44 has vacated a leaf node. In it's place the next key from the list to be sorted, key 53 is inserted as shown in Figure 3b. The OVC of 53 against the most recent winner, 44, is calculated and also placed in the leaf node. Notice that only keys in nodes along the winner path need to be compared to find the next winner. Note that every node contains the OVC of the most recent loser at that node against the most recent winner at that node. The winner path consists of nodes where the most recent winner was the most recent tournament tree winner. Hence all nodes along the winner path contain OVCs against the most recent tournament winner, key 44, and their OVC (all against key 44) in the nodes along the winner path (highlighted in Figure 3b) may be compared with each other to find the next winner. Each of the 4 OVC codes is unique and by application of the Unequal offset value code theorem, the outcome of key comparisons is determined simply by the outcome of the comparison of each key's OVC code(against the first winner, key 44). The Theorem also assures us that we may leave the OVC unaltered because the OVC of each key against the new winner, key 45 is the same as against key 44. So each node on the winner path not only contains a pointer to the key that lost at that node, but also the OVC of a node key against the second winner, key 45. No OVC recomputation is required. The process is illustrated in Figure 3c. The second winner key is replaced by key 57. Of course, had any two OVCs compared equal, the Equal Offset Value Code theorem would have applied, the two involved keys compared to determine the winner and offset value of the loser against the winner found. The computed OVC would have been left in the node.

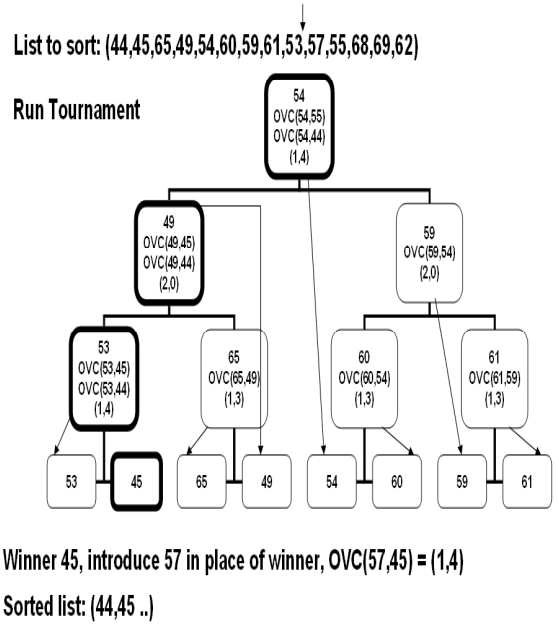

Figure 3c: Example of Tournament Sort with OVC Identification of the Second Winner

The result of replacing the OVC codes and insertion of the next input sort key, key 57 is shown in Figure 3d. The new key, key 53 is entered into the leaf vacated by the second winner. The OVC of this key against the most recent winner, 45, is computed and left at the same leaf tree node. By examining the state of the tree at this step as shown in Figure 3d, we conclude that we are ready to find the next winner. This iterative step repeats until there is no more elements in the input list to be sorted.

The final step is Tree Flush. 'Infinite' keys are inserted into the tournament tree and when the first infinite key emerges from the tree the algorithm is terminated.

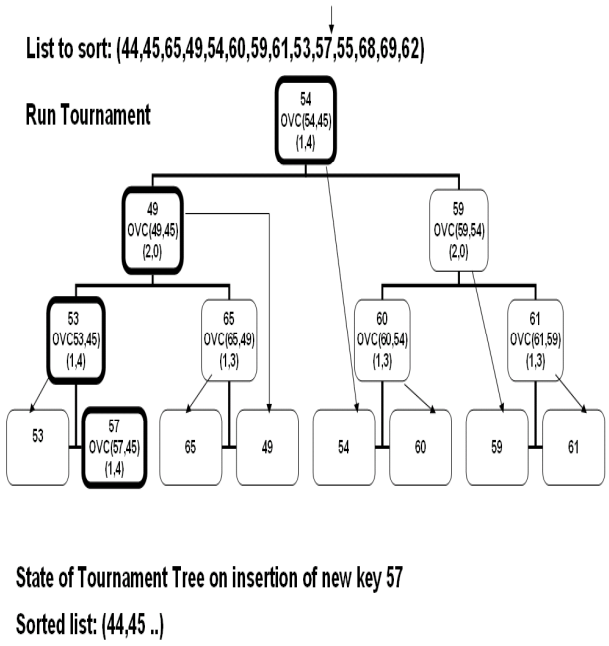

Figure 3d: Example of Tournament Sort with OVC Identification of the Second Winner

6. Formal Algorithm

In this section, we give the formal algorithm for Tournament Tree based Replacement/Selection Sorting using Offset Value Codes. Please note that we combine steps 2 and 3 of the algorithm by defining a getnextkey() function. The function returns the next key from the input list to be sorted. When the list is exhausted it returns the 'infinite' key. The transition from step 2 to step 3 occurs when the first 'infinite' key is returned. Also note that we have not included the run number generation steps here, which are straightforward to add. Instead, we assume that the next key from the list is always larger than the most recent winner.

Before we give the formal algorithm, we define the data structures used. The basic data structure is a binary tree with nodes. Except for leaf nodes, each node is associated with two nodes, a right child and a left child. Leaf nodes have no children. Each node is also associated with a parent which is also a node, except the root node. Each node is comprised of five elements, winner, loser, winner_key, loser_key, and ovc. The elements winner and loser are each identifications (may be implemented as a pointer) of a leaf node, and variables winnerkey and loserkey are valid key values. The final element, ovc, is an offset value code. Input_List is a list of keys to be sorted. Sorted_List is a list of sorted keys. (If we were to also consider run numbers, there would be one sorted list per run.) Getnext() is a function which, when invoked, removes and returns a key from the Input_List. Appendtorun(key) is a function which appends the parameter key to the end of the list, Sorted_list (To include run number processing, this function would append the key to its corresponding list.). The following functions are available for nodes. 1) leftchild(node) returns the left child of a node for a non-leaf node, NULL for a leaf node, 2) rightchild(node) returns the right child of a node for a non-leaf node, NULL for a leaf node, 2) parent(node) returns the parent of a node for all nodes except for the root node for which it returns NULL. The algorithm is given in two parts. The first part is step 1 which initialises the tournament tree. The second part combines steps 2 and 3, and it forms the sorted list and flushes the tournament tree when the Input_List is exhausted. The root node is referred to as root. We track the nodes supplying the winning key in the variable mostrecentwinner which has the same structure as the variable node.

Tournament Tree Initialization Algorithm:
For all nodes belonging to the tree: set node=NULL;
For each leaf node:
{ node.winner_key = getnext()
node.winner = node;
node.ovc = OVC(node.winner_key, minkey);
}
For each non leaf node for which neither leftchild(node) nor rightchild(node) is NULL
{case 1: leftchild(node).ovc < rightchild(node).ovc
{node.winnerkey = rightchild(node).winnerkey
node.winner = rightchild(node).winner
node.loserkey = leftchild(node).winnerkey
node.loser = leftchild(node).winner
node.ovc = leftchild(node).ovc
}
case 2: leftchild(node).ovc > rightchild(node).ovc
{node.winnerkey = lefttchild(node).winnerkey
node.winner = leftchild(node).winner
node.loserkey = rightchild(node).winnerkey
node.loser = rightchild(node),winner
node.ovc = rightchild(node).ovc
}
case 3: leftchild(node).ovc = rightchild(node).ovc
{case 1: leftchild(node).winner<rightchild(node).winner
{node.winnerkey = leftchild(node).winnerkey
node.winner = leftchild(node).winner
node.loserkey = rightchild(node).winnerkey
node.loser = rightchild(node).winner
node.ovc=OVC(rightchild(node).key,
leftchild(node).key)
}
case2: leftchild(node).winner > rightchild(node).winner
{node.winnerkey = rightchild(node).winnerkey
node.winner = rightchild(node).winner
node.loserkey = leftchild(node).winnerkey
node.loser = leftchild(node).winner

node.ovc =

```
    OVC(leftchild(node).key, rightchild(node).key)
   }
  }
 }
 appendtorun(root.winnerkey)
 mostrecentwinner = root.winner
```

At the end, the first winning key would be found. We combine steps 2 and steps 3 into one next. temp is a variable with the same structure as a node. pathwinner is a pointer to a leaf node and identifies the key that is the winner in the tournament during the traversal from the leaf to the root. Therefore pathwinner.loserkey referes to the loserkey attribute of the leaf node pointed to by the variable pathwinner. compare2node is a pointer to a nonleaf node. It assumes as values nodes on the winner path. mostrecentwinner.loserkey contains the tournament tree winner at the end of a tournament round.

```
Algorithm FormRun()
do while mostrecentwinner.loserkey not equal to
INFINITY
{ temp=mostrecentwinner
  pathwinner=mostrecentwinner
  pathwinner.loserkey=nextkey()
  pathwinner.ovc=
    OVC(pathwinner.loserkey,temp.losrkey)
  compare2node=parent(pathwinner)
  do while compare2nod is not NULL
   {case1: pathwinner.ovc>compare2node.ovc
     {}
    case2: pathwinner.ovc<compare2node.ovc
     {temp=pathwinner
      pathwinner=compare2node.loser
      compare2node.loser=temp
     }
    case3: pathwinner.ovc=compare2node.ovc
     {case1: pathwinner.loserkey<compare2node.loserkey
        {compare2node.ovc=OVC(compare2node.loserkey,
           pathwinner.loserkey)
        }
      case2:pathwinner.loserkey>compare2node.loserkey
        {temp=pathwinner
         pathwinner=compare2node.loser
         compare2node.loser=temp
         compare2node.ovc=OVC(compare2node.loserkey,
           pathwinner.loserkey)
        }
     }
    compare2node=parent(compare2node)
   }
  appendtorun(pathwinner.loserkey)
  mostrecentwinner=pathwinner
}
```

7. Hardware Assist

IBM's z/Architecture [IBM2004] defines the offset value code as a 4 byte word. The first half word contains a byte offset into the key and the second half word contains the one's complement of the half word value of the key at the offset. A tournament tree is comprised of eight byte nodes. Each node contains two parts, a) an OVC for the key losing at that node against the key that won at that node, and b) a parameter usable to locate the key. IBM's z-series processors have implemented the Update Tree (UPT) and Compare and Form Codeword (CFC) instructions. The UPT instruction is given a pointer to a a key, its OVC and a pointer to a tree node and invoked. On invocation the UPT instruction compares the given OVC with the tree resident OVC, determines the winner, updates the tree node if necessary with a pointer to the losing key and repeats for the nodes parent by taking the winning key and OVC forward. The instruction terminates on equal comparison of two OVC codes or when the root of the tree is reached. If the UPT instructions terminates at node that is not the root, the CFC instruction is invoked. The CFC instruction compares two keys, determines the winner and computes the offset value code of the losing key against the winner and updates the tree node. The UPT instruction is once again invoked at this new node. Ultimately the UPT instruction will terminate at the root of the tree after finding the winner key. These two instructions are invoked in alternating sequence until a winner is found. Together, the UPT and CFC instructions do bulk of the sorting in IBM's commercial DBMS DB2 running on z/Architecture processors.

8. Conclusion

In this paper we examined the role of hardware in commercial RDBMS processing. Four often used CPU intensive functions in DB2, sorting, compression, encryption and concurrency/coherency control, were identified for offload into special hardware assists available on otherwise general purpose processors. The task of sorting was further explored. Offset Value Coding was introduced and a method to improve sorting based on offset value codes was given and illustrated. The formal algorithm for replacement/selection tournament tree sorting with offset value codes was given. Two hardware instructions to assist in the sorting were described. These two instructions are used to do sorting in DB2. All four functions have been successfully accelerated by hardware assists for DB2 running on IBM's z-Series processors. They are in active use by a multitude of large commercial enterprises, governments and academic institutions in every geography. Customer feedback has been extremely positive. Anecdotal evidence suggests that at some installations upwards of 40% of all CPU cycles have been offloaded to these hardware assists. We continue to look

for new opportunities and are working actively to build synergy between hardware and DBMS processing.

References

[BALL94] Ballinger, C., Evolving Teradata Decision Support for Massively Parallel processing with Unix, Proc. of SIGMOD Intnl. Conf. on Management of Data, Minneapolis, MN, p490, 1994.

[BERG2005] Berger, J., and Novak, J., How secure is your DB2 data?, z/OS Hot Topics Newsletter, Issue 13, Aug. 2005.

[BOWE97] Bowen, N. S, A Locking Facility for Parallel Systems, IBM Systems Journal, 1997.

[BRUI98] Buini, P., and Naidoo, R., DB2 for OS/390 and Data Compression, IBM Redbook SG24-5261-00, IBM Corp., San Jose, CA, Nov. 1998.

[CHAN96] Chang C.-C., Davoll G.L., El-Ruby M.H., Friske C.A., Iyer B.R., Lazarus J., Wilhite D., and Plambeck K.E., Method and system for adaptively building a static Ziv-Lempel dictionary for database compression, Journal of Cleaner Production, Volume 4, Number 3, 1996, pp. 255-255(1).

[CHEN91] Cheng, J. M., Haderle, D. H., Hedges, R., Iyer, B. R., Messinger, T., Mohan, C., Wang, Y., An Efficient Hybrid Join Algorithm: A DB2 Prototype, Proc. 7th Intl. Conf. on Data Eng., pp.171 - 180, 1991

[CONN77] Conner, W. M., Offset Value Coding, IBM Technical Disclosure Bulletin, 12-77, December, 1977, pp. 2832-37, IBM Corp.

[DIAS87] Dias, D. M., Iyer, B. R., Robinson, J. T., and Yu, P. S., Design and Analysis of Integrated Concurrency-Coherence Controls, Proc. 13th Intl. Conf. on VLDB, pp 463-471, 1987.

[HACI2002] Hacigumus, H., Iyer, B., and Mehrotra, S., Providing Database as a Service, Proc. of 18th Intl. Conf. on Data Engineering, San Jose, CA, 2002.

[IBM2004] IBM: z/Architecture. Principles of Operation. SA22-2832-03, May 2004, 4th edition.

[IDM85] The IDM Database Server, Britton-Lee Inc., 1985

[JOST97] Josten, J. W., Mohan, C., Narang, I., Teng, J. Z., DB2s Use of the Coupling Facility for Data Sharing, IBM Systems Journal, 1997.

[KNUT73] Knuth, D.E., The Art of Computer Programming, Vol. 3. Sorting and Searching , Addison-Wesley, Reading MA, 1973.

[SLEG2004] Slegel,T. J., Pfeffer,E.,and Magee, J. A., The IBM eServer z990 microprocessor, IBM Journal of Research and Development, Vol. 48, No. 3/4, 2004

[TERA83] Teradata: DBC/1012 Data Base Computer Concepts and Facilities, Teradata Corp. Document No. C02-0001-00, 1983.

[UBEL85] Ubell, M., The Intelligent Data Base Machine (IDM), in Query Processing in Database Systems, edited by Kim, W, Reiner, D., and Batory, D., Springer-Verlag, 1985.

Acknowledgements

The author would like to express his sincere thanks to all his IBM colleagues with whom he teamed together to complete the projects mentioned in this paper. Appreciation is due to many IBM database customers who shared their problems, workloads and feedback. Finally, the IBM managers of these projects deserve special mention for enabling their teams to execute successfully.