Algorithms for PCM-Database Operators

A Thesis

Submitted

For the Degree of

Master of Engineering

in

Computer Science and Engineering

by

Abhimanyu Singh

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

July 2013

TO

My Family and Friends

Acknowledgements

I would like to thank my advisor Prof. Jayant R. Haritsa for his encouragement and unmatched guidance throughout my project work. I have been extremely fortunate to work with him.

I would like to thank Anshuman Dutt and all my fellow lab-mates for their help and suggestions. Also I would like to thank all my CSA friends who have made my stay at IISc fun and memorable.

Finally, I am indebted with gratitude to my parents for their love and inspiration that no amount of thanks can suffice.

Abstract

Phase change memory (PCM) is an emerging memory technology. It has many interesting properties. It is random access, byte-addressable and non-volatile memory. PCM has 2-4$\times$ density than DRAM. Its idle power consumption is less than that of DRAM. PCM read latency is comparable to DRAM. But its write latency is high and we can perform only limited number of writes to a PCM cell ($10^6 - 10^8$). If we compare PCM with NAND-Flash then PCM has much better endurance and read/write latency than NAND-Flash. Because of all these characteristics it is expected that PCM will become one of the main components in the memory hierarchy in computer systems. A design goal of databases for such computer systems would be to reduce writes to PCM without compromising much on running time. We should reduce writes because if writes to PCM will be more then it will wear out quickly and the system will not be usable at all. Running time is also important because user can't wait forever for the result of any processing task. We did background study of work done for systems having PCM and designed new PCM-aware algorithms for two database operators, sort and join.

Contents

List of Tables

List of Figures

Keywords

Phase change memory, PCM, Database, Database Operators, Sort, Join.

Chapter 1

Introduction

Storage Class Memory (SCM) is a solid-state memory that blurs the boundaries between storage and memory by being low-cost, fast and non-volatile [16] . There are various candidates for SCM but PCM is the best candidate for SCM. PCM memory is an emerging non-volatile memory technology. PCM is byte addressable and solid state memory. It is expected that PCM would play very important role in future because of various reasons. As explained in [13] that memory capacity is not increasing at the same rate as the number of cores per chip area. Hence future systems are likely to be performance-limited by inadequate memory capacity. If we will use large amount of DRAM it will occupy more area. PCM has 2-4$\times$ density in comparison to that of DRAM hence PCM is good in this sense. Similarly when PCM is idle it consumes order of magnitude less power than DRAM hence systems having PCM as main memory would be more energy efficient.

But having many good characteristics there are some drawbacks with PCM. PCM consumes more energy when we write data to it and its write speed is slow as compared to its read speed. PCM also has write endurance problem, we can write only $10^6 - 10^8$ times to a cell.

So given PCM with these characteristics now the question is that where we should put PCM in the memory hierarchy and what change we should make in algorithms or do we really need to change the existing algorithms. Since our concern is mainly on database so we will talk only about algorithms used in database engine.

Because of random access PCM can be used as main memory in the system. There are various proposals for how PCM can be used in memory hierarchy. It is shown in [20] that just using a small amount of DRAM with PCM can reduce write traffic from CPU to PCM significantly. So we should have both DRAM and PCM in the memory hierarchy. So there are various possible system model based on the choice of having DRAM in system and access control over DRAM e.g. as a cache or explicitly by user program.

Intuitively we can say that we need to change the algorithms because for example if we look at the quicksort algorithm used as subroutine in external merge sort in databases, quicksort does around $N \times log_2(N)$ writes to memory in worst case (it depends on implementation), while the write optimal sort algorithm (cycle sort [7]) can sort it doing just N writes. Here N is the number of element and we are counting writes as elements written to memory.

In this thesis first we describe PCM and its characteristics in detail. We also describe what the possible system models are. Then we focus about the challenges that we need to consider while designing algorithms for PCM systems. We also describe the background work in brief that has been done for PCM database systems. In particular we discuss sort algorithm described in [15] and join algorithm described in [5] .

Next we describe our contribution to the sort and join operator. In last we describe the performance of our algorithms experimentally. Next we describe few algorithms of theoretical interest. The thesis concludes by comparing our algorithms with other existing algorithms.

Chapter 2

Phase Change Memory

In this section we describe about the PCM memory and compare it with other memory technology. Then we discuss the possible system models and various challenge with the PCM.

2.1 PCM Technology

Phase change memory exploits the large resistance contrast between amorphous and crystalline state of phase change materials such as Chalcogenide glass. Difference between the resistances can be used to infer the logical state of binary data.

Writing or reading data to PCM need application of electric current. The application of current lead to either SET or RESET depending on the value of current as shown in Figure 2.1 .

It is clear from Figure 2.1 that writing data to PCM takes more time and energy as compared to reading data from PCM. At normal temperature PCM offers many years of data retention.

Some optimization has been made in hardware to reduce the writes to PCM. ***Read before write*** optimization is mentioned in [22] . In it during the write operation of PCM, we first read the data already present in the cell and if it is same, we don't perform any write to that PCM cell.

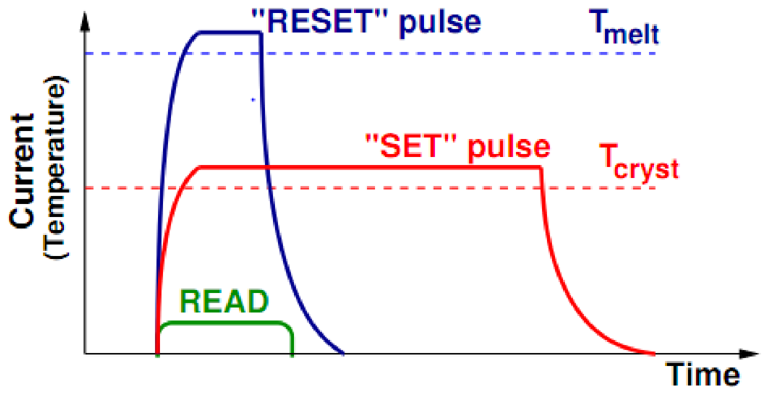

Figure 2.1: Currents and timings (not to scale) for SET, RESET and READ operation on a PCM cell. For phase change material $Ge_2Sb_2Te_5$, $T_{melt} \approx 610°$ C and $T_{cryst} \approx 350°$C. [5]

2.2 System Model

The location of PCM in memory hierarchy depends on the characteristics of PCM. Table 2.1 contains a good comparison of various memory technologies.

Table 2.1 Comparison of memory technologies[5]

	DRAM	**PCM**	**HDD**
Read energy	0.8 J/GB	1 J/GB	65 J/GB
Write energy	1.2 J/GB	6 J/GB	65 J/GB
Idle power	∼100 mW/GB	∼1 mW/GB	∼10 W/TB
Endurance	∞	$10^6 - 10^8$	∞
Page size	64B	64B	512B
Page read latency	20-50ns	$\sim 50ns$	$\sim 5ms$
Page write latency	20-50ns	$\sim 1\mu$s	$\sim 5ms$
Write bandwidth	∼GB/s per die	50-100 MB/s per die	∼200 MB/s per drive
Density	1×	2-4×	N/A

Now it is clear from Table 2.1 that the read latency of PCM is close to DRAM while its write latency is order of magnitude slower but PCM offers a density advantage over DRAM and more energy efficient than DRAM in idle mode. So based on these analysis system models shown in Figure 2.2 are the various possibilities.

The basic difference between model (b) and (c) is that in model (b) the software has explicit control over DRAM and PCM but in system model (c) the DRAM act as

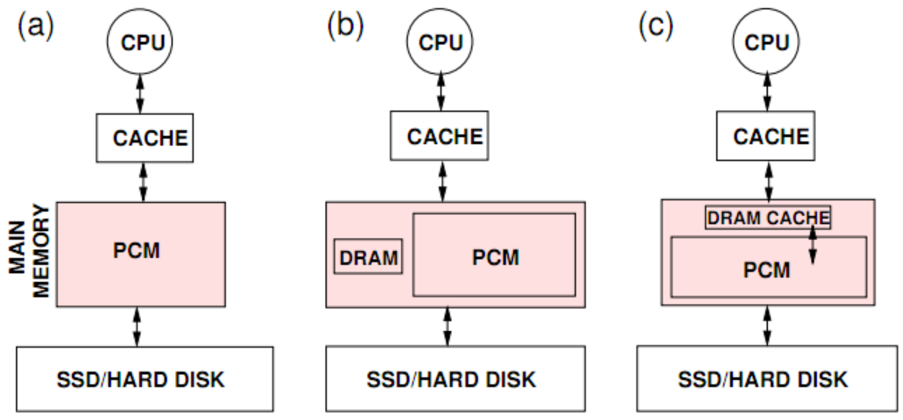

Figure 2.2: Candidate main memory organization with PCM.

a hardware cache of PCM. The difference between model (a) and (c) is that the size of DRAM cache is larger than available conventional last level cache.

It has been shown in [20] by experiments that use of small amount of DRAM with PCM will lead to improvement in the performance of application and will also extend the life of PCM significantly. Hence system model (b) and (c) are good candidates. We will call system model (b) as DRAM_EXPLICIT model and system model (c) as DRAM_CACHE model.

DRAM_EXPLICIT model is more flexible than DRAM_CACHE model in terms of writes to PCM because we can always force any data to be resident in DRAM. But in this model we must have some way to distinguish between DRAM and PCM. One possibility could be that we distinguish between them by address space range. That may lead to a lot of changes in the design of OS. And in this case any running application must be aware of this. Hence it may be hard to run older existing applications on this model without quick wear out damage to PCM.

In case of DRAM_CACHE model even if applications are unaware of existence of PCM as main memory, we can run them easily. So from functional point of view no change is needed in existing software to run on this model. So DRAM_CACHE system model is more feasible from practical point of view.

2.3 Challenges with PCM

Based on the discussion about the characteristics of PCM so far we should keep the following things in mind while designing algorithms for PCM systems:

1. High energy consumption during write than read.
2. High latency and low bandwidth of write operation.
3. Limited endurance of PCM cell.

Hence the main idea is that we have to reduce writes to PCM. But number of writes to PCM is not the only **performance metrics** of an algorithm. Running time is also an important factor. So we have to reduce writes to PCM without compromising much on running time. From one another point of view if a small part of PCM will wear out then whole PCM will be unusable because now we would be having more bad blocks and all extra blocks to take place of bad blocks will exhaust. So one more performance metrics is the distribution of writes to PCM. Ideally writes should be uniformly distributed.

2.4 Background and Related Work

Algorithm for sort operator was suggested in [15] . They have considered DRAM_EXPLICIT system model. They have considered that database is disk resident and can be much larger than available main memory. They have considered external merge sort algorithm and modified the subroutine algorithm to sort a chunk of size of available main memory size. We have given their algorithm in Chapter 6 .

Algorithm for join operator was suggested in [5] . Their assumption was that whole database is present in the PCM. For join they modified the hash-join algorithm. They proposed virtual partitioning technique. In virtual partitioning in each partition we will only record the tuple ID instead of complete tuple. Because of random access characteristic of PCM we can always get complete data of the tuple without degradation in performance. Since during partitioning we are writing only tuple ID instead of complete tuple hence it will reduce the writes to PCM.

2.5 Choice of operators and assumptions

We focus mainly on two operators, sort and join. We choose these two operators because these are most expensive operators in databases in terms of both running time and writes to memory.

By default we are assuming that database is **disk resident** and much larger than available main memory. We assume system model to be **DRAM_CACHE system model** because we have already explained the practical feasibility of DRAM_CACHE model over DRAM_EXPLICIT model in Section 2.2 . We also assume that the cache in system is a **write-back cache**. For write analysis we are assuming that only one bit is stored per cell of PCM.

For write analysis we will measure writes to PCM at the following granularity: *(a) bits* and *(b) words*. Choice *(a)* impacts PCM endurance. Choice *(b)* influences PCM write latency. Fortunately, there is often a simple relationship between *(a)* and *(b)*. Denote γ as the average number of modified bits per modified word. γ can be estimated for a given input.

Chapter 3

Sort Operator

Sorting is an important operation in database. We have to sort a table which is present on hard disk. Size of table is much larger than available main memory to the program. We will use word *memory* for main memory. Following are the other **notations**:

m : DRAM size (effective)

M : PCM size

F_1 : Input file

F_2 : Output file

N : size of F_1

C_i : Chunk of F_1 of size M

We have modified the conventional external merge sort algorithm. Algorithm 1 contains the pseudo code of conventional external merge sort.

Algorithm 1 External merge sort

```
   Local Sort Phase
1: for $i$=1 to $k$ $(=\frac{N}{M})$ do
2:    Read $i^{th}$ chunk $C_i$ of $F_1$ in memory;
3:    Sort $C_i$ in memory; {In-Place Internal Sort}
4:    Write sorted $C_i$ to temporary file $T$.
5: end for
   Merge Phase
6: Merge the $k$ chunks by first reading their parts in memory and writing answer to $F_2$.
```

Write Analysis: Suppose a chunk C_i contains n elements. Then there would be

$k \times n$ writes to PCM because of reading file F_1 in PCM during chunks sort phase. And $2 \times k \times n$ writes will be during merge phase. Now suppose we are using an algorithm A_j to sort chunks in PCM that does $x \times n$ writes to PCM to sort a chunk C_i then total writes to PCM would be $k \times n \times (3 + x)$. So if for an algorithm A_1 x is 2 and for anther algorithm A_2 x is 4 then A_1 will do 28.6 % less writes to PCM than A_2. Hence step 3 of Algorithm 1 is an important factor.

Generally the in-place internal sort in step 3 of Algorithm 1 is done by quicksort. We have designed a new PCM aware quicksort algorithm which does less writes than conventional quicksort.

3.1 1-Pivot PCM-aware partition

In quicksort we partition the unsorted array into two parts. We did this partition based on some randomly chosen pivot value from the array. After partition one sub-array contains elements less than or equal to pivot value and the other sub-array contains elements greater than pivot value. We will call an element as wrong element if it is not in its correct sub-array initially e.g. it is greater than pivot value but is present in the left sub-array or vice-versa. The red elements ({70, 80, 12, 20}) in figure 3.1 are wrong elements.

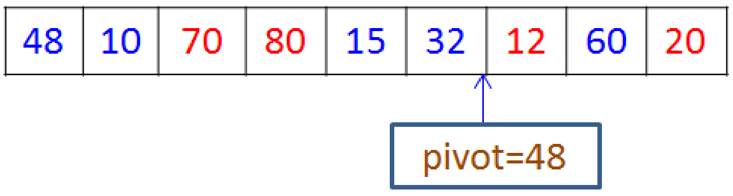

Figure 3.1: 1-Pivot wrong element example

There are various way to swap elements of array so that after partition both sub-arrays are consistent (sub-array having no wrong element). The number of wrong elements in both sub-array will be same. So to reduce writes we will swap one wrong element in one sub-array with the one wrong element in other sub-array. So the number of writes will be equal to number of wrong elements which is best with respect to a particular given pivot value. Based on this idea the pseudo code for 1-pivot PCM aware partition is given in Algorithm 2 .

Algorithm 2 1-Pivot PCM aware partition

INPUT: Type a[], int p, int r

```
1: $randIndex = p + rand()\%(r - p)$;
2: $pivot = a[randIndex]$;
3: $small = 0$; { $small$: # elements smaller than or equal to pivot}
4: for $i$=$p$ to $r$ do
5:    if $(a[i] \leq pivot)$ then
6:       $small++$;
7:    end if
8: end for
9: $pivotIndex = (p + (small - 1))$;
10: swap($a$, $pivotIndex$, $randIndex$); {#write=2}
11: $left = p$, $right = (p + small)$;
12: while $(left \neq pivotIndex)$ do
13:    while $((left \neq pivotIndex)$ AND $(a[left] \leq pivot))$ do
14:       $left++$;
15:    end while
16:    if $(left == pivotIndex)$ then
17:       break;
18:    end if
19:    while $(a[right] > pivot)$ do
20:       $right--$;
21:    end while
22:    swap($a$, $left$, $right$); {#write=2}
23:    $left++$, $right--$;
24: end while
25: return $pivotIndex$;
```

Note that hoare-partition ([6]) algorithm can also be written carefully so that it will lead to writes equal to number of wrong elements but we have knowingly written 1-pivot PCM aware partition like this because it can be extended to multi-pivot case easily while hoare-partition logic would not.

3.2 1-Pivot PCM aware quicksort

In 1-pivot PCM aware quicksort we will use 1-pivot PCM aware partition in Algorithm 2 as partition subroutine. The other idea is same as conventional quicksort with the modification that we will implement an explicit stack instead of recursion and will sort the smaller sub-array first so that stack size is $O(log_2 n)$ which is constant for all practical sorting.

Write Analysis: Suppose $W(n)$ is the number of writes to PCM to sort an array of size n elements and the effective DRAM cache size is m elements. When the sub-array size will be less than or equal to m there will be only n writes to PCM in worst case for writing answer to PCM because multiple writes will be absorbed by cache. The number of wrong elements in a partition can never be larger than size of sub-array hence maximum writes during a partition would be just twice of size of smaller sub-array. Hence we can define

$$W_{max}(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ n & \text{if } 1 < n \leq m \\ \max_{0 \leq s \leq \lfloor \frac{n-1}{2} \rfloor}(W_{max}(s) + W_{max}(n-s-1) + 2s + 2) & \text{if } n > m \end{cases}$$

Where $W_{max}(n)$ is the maximum possible writes to PCM to sort an array of size n and s is the size of smaller sub-array. We have proved that $W_{max}(n) < \lceil (i+2) \times n \rceil$ for all n, where $i = max\left(0, \log_2\left(\frac{n}{m}\right)\right)$. That means if we have 5% DRAM cache then the worst case writes to PCM will be $6.32 \times n$, regardless of quality of partition e.g. approximately balanced or unbalanced. A partition is called balanced if the size of sub-arrays are equal. Here the significance of $6.32 \times n$ is that it shows that if we write the quicksort code carefully then the worst case writes would not be quadratic in n and it is close to $2 \times n$ writes of selection sort.

3.3 Proof of maximum writes

Theorem 1: For all a, b such that $a > 0$, $b > 0$ and $a \leq b$

$(a+b)\log_2(a+b) \geq a\log_2 a + b\log_2 b + 2a$

Proof

$(a+b)\log_2(a+b) \ ? \ a\log_2 a + b\log_2 b + 2a$

$\equiv a\log_2(a+b) + b\log_2(a+b) \ ? \ a\log_2 a + b\log_2 b + 2a$

$\equiv a\log_2(\frac{a+b}{a}) + b\log_2(\frac{a+b}{b}) \ ? \ 2a$

$\equiv \frac{a}{a+b}\log_2(\frac{a+b}{a}) + \frac{b}{a+b}\log_2(\frac{a+b}{b}) \ ? \ 2\frac{a}{a+b}$ Divide by $(a+b)$

Define $x = \frac{a}{a+b}$ and $y = \frac{b}{a+b}$

Then $x + y = 1$, $x > 0$, $y > 0$, $x \leq y$, $0 < x \leq 0.5$ and $y = 1 - x$

$\equiv x\log_2(\frac{1}{x}) + y\log_2(\frac{1}{y}) \ ? \ 2x$

$\equiv -x\log_2 x - y\log_2 y \ ? \ 2x$

$\equiv -x\log_2 x - (1-x)\log_2(1-x) \ ? \ 2x$

Let $f_1 = (-x\log_2 x - (1-x)\log_2(1-x))$ and $f_2 = 2x$. It can be shown formally that for $0 < x \leq 0.5$ $f_1 \geq f_2$. The graph of f_1 and f_2 are shown in Figure 3.2 .

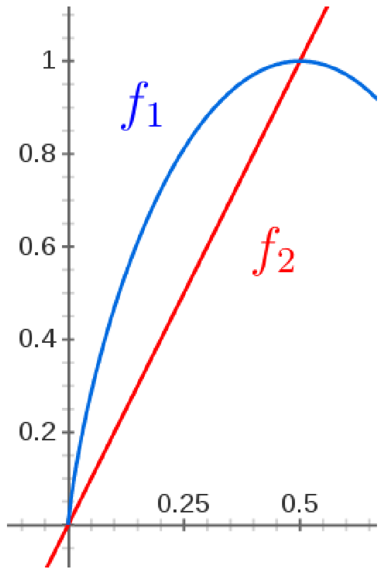

Figure 3.2: Comparison of function $f_1 = (-x\log_2 x - (1-x)\log_2(1-x))$ and $f_2 = 2x$

Hence

$\equiv -x\log_2 x - (1-x)\log_2(1-x) \geq 2x$

$\equiv (a+b)\log_2(a+b) \geq a\log_2 a + b\log_2 b + 2a$

Corollary 1: In theorem 1 if we take $(a+b)$ as n and s as a. Then we have

$s\log_2\frac{s}{m} + (n-s)\log_2\frac{n-s}{m} + 2s \leq n\log_2\frac{n}{m}$

where $1 \leq s \leq \frac{n}{2}, m \geq 2, n \geq 2$

Terms containing m will cancel out from both side.

Theorem 2: For all k, m such that $k \geq 1$, $m \geq 2$ and $k \leq m$

$k < (m+k)\log_2\frac{m+k}{m}$

Proof

$k \; ? \; (m+k)\log_2\frac{m+k}{m}$

We can write $k = x \times m$ where $0 < x \leq 1$

$\equiv x \times m \; ? \; (m + x \times m)\log_2\frac{m+x\times m}{m}$

$\equiv x \; ? \; (1+x)\log_2(1+x)$

Let $f_1 = (1+x)\log_2(1+x)$ and $f_2 = x$. It can be shown formally that for $0 < x \leq 1$ $f_2 < f_1$. The graph of f_1 and f_2 are shown in Figure 3.3 .

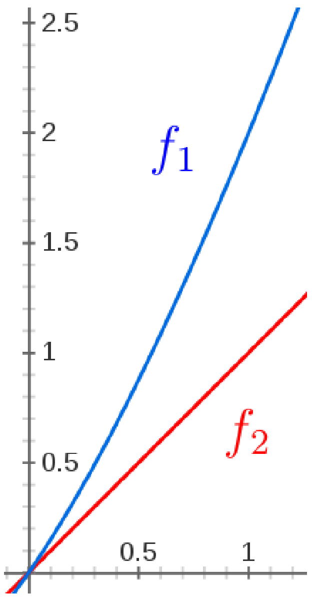

Figure 3.3: Comparison of function $f_1 = (1+x)\log_2(1+x)$ and $f_2 = x$

Hence

$\equiv x < (1+x)\log_2(1+x)$

$\equiv k < (m+k)\log_2 \frac{m+k}{m}$

Theorem 3:

If $T(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ n & \text{if } 1 < n \leq m \quad \text{where } m \geq 2 \\ \max_{1 \leq s \leq \lfloor \frac{n}{2} \rfloor}(T(s) + T(n-s) + 2s) & \text{if } n > m \end{cases}$

Then $T(n) < (i+2)n$

Where $i = max\left(0, \log_2 \frac{n}{m}\right)$

Proof

The main idea of proof is that first we prove the theorem for smaller value of n on case basis. After that for larger value of n we will apply Corollary 1 directly.

Case1: $1 \leq n \leq m$

$T(1) = 0$ which is less than $(i+2)n = 2$

$T(n) = n$ which is less than $(i+2)n = 2n$

Case2: $m < n \leq 2m$

In this case $T(n) = \max_{1 \leq s \leq \lfloor \frac{n}{2} \rfloor}(T(s) + T(n-s) + 2s)$. Let's assume that $n = m + k$ where $1 \leq k \leq m$. Now we have following possibilities:

Case2A: $s \geq k$

Hence $T(n) = \max_{k \leq s \leq \lfloor \frac{m+k}{2} \rfloor}(T(s) + T(n-s) + 2s)$

$= \max_{k \leq s \leq \lfloor \frac{m+k}{2} \rfloor}(s + (n-s) + 2s)$

$= \max_{k \leq s \leq \lfloor \frac{m+k}{2} \rfloor}(n + 2s)$

$= (n + 2\frac{m+k}{2}) = 2n < (i+2)n$

Case2B: $s < k$

Hence $T(n) = \max_{1 \leq s \leq k-1}(T(s) + T(n-s) + 2s)$

$= \max_{1 \leq s \leq k-1}(s + T(m+k-s) + 2s)$

$T(m+1) = 2(m+1) = 0 + 2(m+1)$

$T(m+2) = max \begin{cases} 2(m+2) \\ 2*1 + T(1) + T(m+1) \end{cases}$

$$= max \begin{cases} 2(m+2) \\ 2*1+0+0+2(m+1) \end{cases} = 0+2(m+2)$$

$$T(m+3) = max \begin{cases} 2(m+3) \\ 2*1+T(1)+T(m+2) \\ 2*2+T(2)+T(m+1) \end{cases}$$

$$= max \begin{cases} 2(m+3) \\ 2*1+0+0+2(m+2) \\ 2*2+2+0+2(m+1) \end{cases} = 2+2(m+3)$$

$$T(m+4) = max \begin{cases} 2(m+4) \\ 2*1+T(1)+T(m+3) \\ 2*2+T(2)+T(m+2) \\ 2*3+T(3)+T(m+1) \end{cases}$$

$$= max \begin{cases} 2(m+4) \\ 2*1+0+2+2(m+3) \\ 2*2+2+0+2(m+2) \\ 2*3+3+0+2(m+1) \end{cases} = 3+2(m+3)$$

$T(m+k) = g(k) + 2(m+k)$ where $g(1) = 0$, $g(2) = 0$, $g(3) = 2$, $g(4) = 3$

For $k > 4$

$$T(m+k) = max \begin{cases} 2(m+k) \\ 2*s+T(s)+T(m+k-s) \\ 1 \leq s \leq k-1 \end{cases}$$

$$= max \begin{cases} 2(m+k) \\ 2*1+0+g(k-1)+2(m+k-1) \\ 2*s+s+g(k-s)+2(m+k-s) \\ 2 \leq s \leq k-1 \end{cases} = g(k)+2(m+k)$$

Where $g(k) = max(g(k-1), max(s+g(k-s))$ for $2 \leq s \leq k-1)$

Hence $g(k) = k-1$ for $k > 2$ and $g(1) = g(2) = 0$

Hence $T(m+k) \leq (k-1) + 2(m+k)$ $1 \leq k \leq m$

Now by using Theorem 2

$T(m+k) < ((m+k)\log_2 \frac{m+k}{m} + 2(m+k))$

$\equiv T(n) < (i+2)n$

Case3: $2m < n \leq 3m$

In this case $T(n) = \max_{1 \leq s \leq \lfloor \frac{n}{2} \rfloor}(T(s) + T(n-s) + 2s)$. Now we will make use of mathematical induction [14] . Let's assume that $n = 2m + k$ where $1 \leq k \leq m$.

Case3A: $m < s \leq \frac{m+k}{2} \implies s > k$

By using mathematical induction [14] $T(s) < (i_s + 2)s$ and $T(n-s) < (i_{n-s} + 2)(n-s)$. Since both s and $(n-s)$ are greater than m hence i_s and i_{n-s} would be defined by logarithmic term. Hence

$(2s + T(s) + T(n-s)) < (2s + s\log_2 \frac{s}{m} + 2s + (n-s)\log_2 \frac{n-s}{m} + 2(n-s))$

$= 2n + (s\log_2 \frac{s}{m} + (n-s)\log_2 \frac{n-s}{m} + 2s)$

$< 2n + n\log_2 \frac{n}{m}$ By Corollary 1

$= (i_n + 2)n$

Hence $T(n) < (i_n + 2)n$

Case3B: $k < s \leq m$

In this case $m < (n-s) \leq 2m$, Hence

$T(n) = \max_{k+1 \leq s \leq m}(T(s) + T(n-s) + 2s)$

$= \max_{k+1 \leq s \leq m}(s + 2(n-s) + (n-s-m-1) + 2s)$ From Case 2

$= \max_{k+1 \leq s \leq m}(2n + (n-m-1))$

$= 2n + (n-m-1)$

$< 2n + n$ because $n > m$

$< 2n + n\log_2 \frac{n}{m}$ because $\log_2 \frac{n}{m} > 1$

$= (i_n + 2)n$

Hence $T(n) < (i_n + 2)n$

Case3C: $s \leq k \implies s < m$

In this case $(n-s) \geq 2m$ and $T(n) = \max_{1 \leq s \leq k}(s + T(n-s) + 2s)$

By using mathematical induction [14] $T(n-s) < (\log_2 \frac{n-s}{m} + 2)(n-s)$ because $(n-s) > m$ so its i_{n-s} would be defined by logarithmic value. Hence

$(s + T(n-s) + 2s) < (s + 2s + (\log_2 \frac{n-s}{m} + 2)(n-s))$

$= s + 2s + 2n - 2s + (n-s)\log_2 \frac{n-s}{m}$

$= 2n + n\log_2 \frac{n-s}{m} + s - s\log_2 \frac{n-s}{m}$

$= 2n + n\log_2 \frac{n-s}{m} + s - s\log_2 \frac{n-s}{m}$

$= 2n + n\log_2 \frac{n-s}{m} + s - s(value \geq 1)$

$\leq 2n + n\log_2 \frac{n-s}{m}$

$< 2n + n\log_2 \frac{n}{m}$ Because $\log_2 \frac{n}{m} > \log_2 \frac{n-s}{m}$

$= (i_n + 2)n$

Hence $T(n) < (i_n + 2)n$

Case4: $n > 3m$

In this case $T(n) = \max_{1 \leq s \leq \lfloor \frac{n}{2} \rfloor}(T(s) + T(n-s) + 2s)$.

Case4A: $s \leq m$

It is same as Case 3C because in this case $(n-s) \geq 2m$.

Case4B: $s > m$

It is same as Case 3A because in this case both s and $(n-s)$ are greater than m hence i_s and i_{n-s} would be defined by logarithmic term. So we can apply Corollary 1 directly. Hence $T(n) < (i_n + 2)n$

Theorem 4: If $W_{max}(n)$ is maximum number of writes by 1-pivot PCM aware quicksort then $W_{max}(n) \leq T(n)$.

Proof

It is trivial to proof. Actually the recurrence for $T(n)$ is for the 1-pivot quicksort in which we are using pivot value just to deciding the boundary of partitions.

3.4 Multi-Pivot PCM aware quicksort

To further make use of DRAM cache we will use multiple pivots to partition array into sub-arrays in a single pass. The benefit would be that as the total number of wrong elements cannot be larger than n (size of array) so number of writes would not be larger than n in a single partition but the expected size of sub-arrays would be less than effective DRAM cache size hence in next passes the sub-arrays will fit in DRAM cache which will lead to reduction in writes. The pseudo code of this partition method is given in Algorithm 3 .

Algorithm 3 Multi-Pivot PCM aware partition

```
n is the size of sub-array
m is the effective cache size ($n > m$)
c is a constant having value 2-3

 1: $k = c \times \frac{n}{m}$;
 2: randIndex[] = generate $k$ random indexes;
 3: $pivot[] = a[randIndex]$;
 4: $size[] = 0..0$; {size of sub-arrays}
    Read Phase
 5: for $i$=$p$ to $r$ do
 6:    Increment the size of sub-array corresponding to $a[i]$;
 7: end for{Time complexity=$n \times log_2 k$ }
 8: Calculate the boundary index of sub-arrays using their size.
    Swap Phase
 9: Swap wrong elements in sub-arrays in cycle like we do in Cycle sort so that each
    element is in its correct sub-array. {Time complexity=$n \times log_2 k$ }
10: return sub-arrays boundary indexes;
```

3.4.1 Multi-Pivot PCM aware quicksort version 1

In the Multi-Pivot PCM aware quicksort v1 we will use Multi-Pivot PCM aware partition as partition subroutine. When the size of sub-array is smaller than effective cache size then we will sort it using use hoare-partition based quicksort. And when sub-array size is larger than effective cache size we will use Muti-Pivot PCM aware partition as partition subroutine. Then we will recursively sort the sub-arrays whose boundary indexes were returned by partition function.

3.4.2 Multi-Pivot PCM aware quicksort version 2

Multi-Pivot PCM aware quicksort v2 is same as version 1 except that in the multi-pivot partition we will merge the contiguous partition before partitioning. Actually in multi-pivot partitioning we choose more pivots than needed ideally e.g. if effective cache size is

1 MB and we have to sort an array of 5 MB then instead of choosing 4 pivots (which would lead to 5 sub-array each of size 1 MB ideally) we choose 10-15 pivots so that resulting sub-arrays/partitions have size less than effective cache size with higher probability. But it would lead to contiguous sub-arrays whose total size is less than effective cache size. So in version 2 we will merge such partitions after step 7 of Algorithm 3. For example suppose we have an array of 50 elements and effective cache size is 20 and we choose pivots as {10, 20, 30, 40}. Corresponding to these pivots there will be 5 partitions: $\{p_1 \leq 10\}$,$\{10 < p_2 \leq 20\}$, $\{20 < p_3 \leq 30\}$,$\{30 < p_4 \leq 40\}$,$\{40 < p_5\}$. After counting the size of these partitions we found that size of partitions are {22, 5, 5, 12, 5}. Instead of applying swap phase on these 5 partitions we will logically merge partition p_2 & p_3 and p_4 & p_5. So after this the new pivots will be {10, 30} and we would be having only 3 partitions of size {22, 10, 17}. Note that after finding out these new pivots we don't need to recalculate the partitions size again by a read pass. We can get it directly by adding size of merged partitions. Algorithm to merge partitions is very simple and fast. Its pseudo code is given in Algorithm 4 .

Algorithm 4 Logical merging of partitions

```
size[] size of partitions
m is the effective cache size

1: $groupSize = size[1]$;
2: $startIndex = 1$;
3: $n = size.length$;
4: for $i$=2 to $n$ do
5:     if $(groupSize + size[i]) < m$ then
6:         This partition will be merged with group of previous partitions.
7:     else
8:         Merge partitions in range $[startIndex, i - 1]$;
9:         $groupSize = size[i]$;
10:        $startIndex = i$; {Start new group.}
11:    end if
12: end for
```

This merging of partitions is very important otherwise we would be having more #wrong elements in a single pass of multi-pivot quicksort.

3.4.3 Worst case writes bound

The worst case number of writes of multi-pivot quicksort version 2 (PCM_QS_V2) is bounded by the worst case number of writes of 1-pivot quicksort (1P_PCM_QS). Every pass of PCM_QS_V2 in which we use muti-pivot partition can be simulated by multiple passes of 1P_PCM_QS. For example if we chose $\{p_1, p_2, p_3\ (p_1 < p_2 < p_3)\}$ as pivots in PCM_QS_V2 then in simulation first we will choose p_2 as pivot in 1P_PCM_QS and p_1 and p_3 as pivots in next level. Now it is easy to see that any wrong element will reach in its correct sub-array in multi-pivot case in fewer changes in its location (so fewer writes) than in 1-pivot case. **What elements were wrong in 1 pass of multi-pivot will become wrong either at the 1^{st} level (when p_2 will be chosen as pivot) or at 2^{nd} level (when p_1 or p_3 will be chosen as pivot) of 1P_PCM_QS so 1P_PCM_QS will have at least as many writes as PCM_QS_V2** . So writes in muti-pivot case is bound by the writes in 1-pivot case. Note that writes during 1 pass of muti-pivot case can be larger than only 1^{st} pass of 1-pivot case but will be lesser than total writes during all passes at all levels of 1-pivot that correspond to 1 pass of multi-pivot quicksort. **Merging of smaller partitions into one in PCM_QS_V2 is important** because presence of many contiguous smaller partition would create more wrong elements so more writes but these writes will not be covered in simulation because all these partitions will fit in cache collectively so no writes during swap phase.

Though worst case of PCM_QS_V2 is same as 1P_PCM_QS but experimentally we found that expected number of writes for PCM_QS_V2 is lower than 1P_PCM_QS. We found that number of writes is between $2n$ to $3n$.

3.5 Wear-leveling optimization during merge phase

In merge phase of conventional external merge sort algorithm (1) we divide memory into $k+1$ buffers and in k buffers we read data from corresponding sorted chunks from disk and merge them into the 1 remaining buffer (call it output buffer). In this case the number of writes to output buffer will be much more than others. So writes will not be distributed uniformly. For that we will divide memory into $k+2$ buffers of equal size and we will keep changing the buffer that we will use as output buffer. The implementation of rotation policy is easy. To implement it we keep counters for each $k+2$ buffer, which counts that how many times we have done writes to the buffer. Writes to buffer are counted based on how many times it has been used for input buffer of a chunk and as output buffer. The pseudo code of rotation policy is given in Algorithm 5.

We found that using this method the writes to PCM were almost uniformly distributed. Here actually we want to claim that no buffer will have more writes than $maxWriteCount$ which imply that difference in $writeCount$ of two buffers will not be more than 2. We can say this intuitively because of following observations:

1. $writeCount$ of a buffer will never exceed $maxWriteCount$ when it is assigned for input for a chunk. It is true at the time of initialization and later because of the condition in Statement 9 of Algorithm 5 during donation.

2. If $writeCount$ of a buffer exceed $maxWriteCount$ when it was assigned for output buffer then $writeCount$ of all free buffer will be more than $maxWriteCount$ because of the condition in Statement 3 of Algorithm 5 (last part) during gaining.

3. In start of algorithm buffers assigned for input already has accommodation of extra $writeCount$ caused to output buffer because of them.

4. Algorithm is greedy and does donations in best possible way.

We do not have formal proof that this algorithm will have distribution of writes uniform during merge phase. But there is possibility that we can show that just before the occurrence of the event in observation 2 there would be defiantly donation by some input buffer or some buffer assigned for input will become free.

Algorithm 5 Buffer Rotation in Merge Phase

chunkCount Number of chunks
chunkSize Size of a chunk
.
Initialization

```
 1: $bufferCount = chunkCount + 2$;
 2: $bufferSize = \frac{PCM\_SIZE}{bufferCount}$;
 3: $maxWriteCount = \lceil \frac{2 \times chunkSize}{bufferSize} \rceil + 1$;
 4: $writeCount[] = \{0, 0..0\}$;
 5: $SortedList\ freeBuffers = \{bufferCount\}$; {Sorted by writeCount of buffers}
    {Insert the last buffer initially which is free}
```

It will be called when a buffer(bufferIndex) assigned for input will become empty

```
 1: $writeCount[bufferIndex] + +$;
 2: $chunk = getChunkIndex(bufferIndex)$; {Get the chunk index to which this buffer
    is assigned}
 3: if There is no data left in chunk then
 4:   $freeBuffers.add(bufferIndex)$; {Insert in sorted order of writeCount}
 5: else
 6:   $remainingWrites = \frac{reamingChunkSize}{bufferSize}$; {reamingChunkSize is the size of data of
      chunk that has not been processed yet and present on hard-disk}
 7:   $freeBufferIndex = freeBuffers.end()$; {Traverse in the reverse order}
 8:   while $writeCount[bufferIndex] < writeCount[freeBufferIndex]$ do
 9:     if $(writeCount[freeBufferIndex] + remainigWrites) \leq maxWriteCount$
        then
10:       $setChunkBufferIndex(chunk, freeBufferIndex)$;
11:       $freeBuffers.remove(freeBufferIndex)$;
12:       $freeBuffers.add(bufferIndex)$; {Input buffers are like donors, because they
          have less writes than output buffer}
13:       Return; {Best possible donation is done}
14:     end if
15:     $freeBufferIndex = freeBuffers.next()$;
16:   end while
17: end if
```

It will be called when a buffer(bufferIndex) assigned for output will become full

```
 1: $writeCount[bufferIndex] + +$;
 2: $freeBufferIndex = freeBuffers.start()$; {Take the first free buffer which has
    minimum writes}
 3: if $writeCount[freeBufferIndex] < writeCount[bufferIndex]$ then
 4:   $setOutputBufferIndex(freeBufferIndex)$;
 5:   $freeBuffers.remove(freeBufferIndex)$;
 6:   $freeBuffers.add(bufferIndex)$; {Output buffer is like gainer, because it has
      more writes than input buffers}
 7: end if
```

3.6 Effect of DRAM cache model

When we say that the sub-array fits in cache and sorting it will lead to writes to PCM just because of writing answers, for this to hold following conditions must hold: Any memory location that is being accessed during sorting should not be evicted from the cache. During sorting we access three memory regions 1)Sub-array data 2)Stack data 3)Executable code. So if the cache is direct-mapped cache then in worst case accessing one region may lead to eviction of other region. So cache must be at least 3 ways set associative. So if we assume that in worst case 1 way of all the sets will be consumed by stack data and 1 ways will be consumed by executable code then effective cache size would be $\frac{\#ways-2}{\#ways} \times cacheSize$. Note that in this case $\frac{cacheSize}{\#ways}$ should be greater than stack data size and executable code size individually. Practically it is true because for example we have considered 16 ways 2MB DRAM cache. So in this case it would be 128 KB which is enough. If cache is fully associative then there would not be any problem except that we have to change our condition that we use to check when it is safe to use conventional quicksort. So instead of having (sub-array data size) < (effective cache size) it would be (sub-array data size + stack size + executable region size) < (total cache size).

When sub-array is smaller than cache we are assuming that every thing fits in cache so cache replacement policy does not matter if we consider only worst case writes claim but practically it does matter. For better performance it should be pseudo LRU because LRU saves few writes to PCM when we sort a larger sub-array (size > cache size) and then we sort its sub-sub-array (size < cache size) because few elements will stay in cache. When sub-array size is larger than cache then during the swap phase of multi-pivot partitioning we use a temporary location of size equal to tuple size. We assumes that this should always be in cache. We access this temporary location most frequently so replacement policy should not evict it from cache.

The granularity at which the data is transferred between DRAM cache and PCM does not matter much except if we say that two regions are aligned such that they share pages. But whatever is the case the worst case writes claim will hold always.

3.7 Other algorithms of theoretical interest

In this section we describe few algorithms that we have not implemented but may be helpful in some other situations.

3.7.1 Internal out of place merge sort

If we have to sort a table which is already present in PCM and there is extra memory of size equal to table size to write output then we can do sorting so that we have worst case $2 \times n$ writes to PCM. First we will divide the table into small chunks just based on index values so that each chunk fits in cache. Then we will sort each chunk locally by using any in-place sorting algorithm. After that we will merge the locally sorted chunks and write the answer to output buffer. So in this case the writes will be just $2 \times n$, one because of local sorting and one because of writing final sorted table to output buffer. The idea of this algorithm is shown in Figure 3.4.

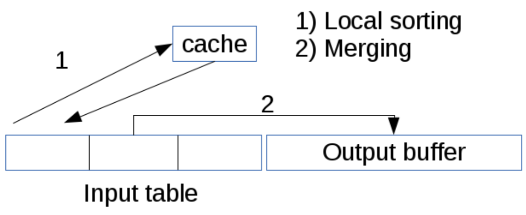

Figure 3.4: Merge sort when we have extra memory for writing answer

We can use this algorithm in Algorithm 1 as subroutine in two ways:

1. One way to use it is to have chunk size as half of available memory. In that case it will perform as good as multi-pivot PCM aware quicksort (theoretically it is better) in terms of only writes to PCM. But it will have more IO cost so its running time will be more. One problem with this is that we have assumed that there would be only 1 local-sort merge pass but by having the chunk size half may lead to 2 or more passes in the external merge sort for a relatively smaller table so more writes to PCM.

2. One other way is to have chunk size as (available memory size - 1-2 MB) and to sort a chunk (of external sort) we will do local sorting of sub-chunks and then during merging of sub-chunks we will write output to the remaining small space (1-2 MB) and when it will become full we will flush it to the disk. Now this algorithm will also have similar performance as the first case but its IO cost can be even more because of writing data to disk in small-2 parts. And in this case writes to PCM will not be uniformly distributed.

3.7.2 External sort having chunk size as cache size

We can sort a table on disk by doing less writes to PCM in comparison to any other algorithm in this thesis by keeping chunk size equal to effective cache size. In this case we will not be using PCM completely until the merge phase of external sort. This algorithm does only $4 \times sizeOfTable$ writes to PCM while all other algorithms in this thesis does around $5 \times sizeOfTable$ writes to PCM. But the problem with this algorithm is that it will have very high IO cost and in this case the external merge sort can have more than 1 pass with a relatively much smaller table. In that case there would not be any benefit of it.

Chapter 4

Join Operator

Join is one of the most expensive operator in database. We are considering only natural joins and equi-joins. We are also assuming the selection predicates in query has already been evaluated and we are considering the relation generated after this. There are various algorithms for join. If we have database to be main memory resident or relation size less than main memory size then nested loop join would be most write efficient though very slow. If relations size is less than available main memory than algorithm mentioned in [5] would be applicable. We are assuming that relation size is much larger than main memory size. In this case we have to read table into main memory multiple times. Hash-join is the most efficient join algorithm in such condition. Algorithm 6 contains the pseudo code of conventional grace hash join algorithm. Following are the **notations** used:

m : DRAM size

M : PCM size

R, S : Join relations

R_i, S_i : Partition of R, S having size $< M$

N_R, N_S : Number of tuples in R and S

L_R, L_S : Size of tuple of R and S

We have omitted the buffer management code in the pseudo code given in Algorithm 6 . When partitioning a relation we would have a buffer allocated for each partition and

Algorithm 6 Grace Hash Join

```
1: $htsize = N_R\times$ per_entry_meta_data_size;
2: $P = \frac{N_R\times L_R+htsize}{M}$;
   **Partition Phase**
3: for each tuple $t$ of $R$ do
4:    $p = hash_1(t)\%P$;
5:    Copy $t$ to partition $R_p$;
6: end for{Partition R}
7: for each tuple $t$ of $S$ do
8:    $p = hash_1(t)\%P$;
9:    Copy $t$ to partition $S_p$;
10: end for{Partition S}
    **Probing Phase**
11: for $p = 0$ to $P-1$ do
12:    Read $R_p$ in memory.
13:    Build hash table $HT_2$ on $R_p$ in memory using hash function $hash_2$. {**Reinitialize
       hash table**}
14:    for each tuple $t$ of $S$ do
15:       Probe $t$ in the hash table $HT_2$.
16:       If there are match(es) then either save them or send them to upper level operator.
17:    end for
18: end for
```

when this buffer will be full we will write it to the file corresponding to the partition. Similarly when creating hash table of a partition R_i in probing phase we would read R_i in main memory and for each tuple in R_i we will insert its join key and corresponding tuple memory address (or index from a base address) in the hash table.

4.1 Write analysis

During partition phase writes to PCM would be $2\times(N_R\times L_R+N_S\times L_S)$, first because of reading data from disk into a buffer and then copying it to corresponding partition buffer. During probing phase the writes to PCM would be $(N_R\times L_R+2\times P\times size_{HT_2}+N_S\times L_S)$. Here $N_R\times L_R$ writes because of reading all partitions of R in memory. And $2\times P\times size_{HT_2}$ writes because of clearing all pointer value in hash table to $NULL$ and then changing them during insertion. $N_S\times L_S$ because of reading all partitions of S into PCM. So the total writes would be around $3\times(N_R\times L_R+N_S\times L_S)+2\times P\times size_{HT_2}$.

Now it is clear that even simple hash join would be better than sort-merge join because just sorting relations would lead to around $4 \times (N_R \times L_R + N_S \times L_S)$ writes in best case. And choice of block nested loop join would be wrong in terms of both running time and writes to PCM. Note that we have already assumed that relations size are much larger than available memory.

4.2 Bitmap based virtual hash-table

In this section we describe a hash table which is compact and based on same idea used in virtual partitioning in [5] . In virtual partitioning we have list of indexes of tuples for each partition. In list of indexes we do optimization by storing the difference from the last index in that partition. So we use the same idea for hashing. We store only the index of tuples in hash table. We call it virtual hashing because key will never be inserted in the hash table. The general configuration of this hash table is shown in Figure 4.1 .

There are N buckets in hash table. In it we have a bitmap array which has 1 bit for each bucket in the hash table. It tells whether the first entry in the bucket is valid or not. All entries in a bucket are packed contiguously. **This hash table does not support deletion.** For each entry there is a bit which tell whether that entry is valid or not. If a bucket has k valid entries then valid bit of first k entries will be 1 and for $(k+1)^{th}$ entry it will be 0 and after $(k+1)^{th}$ entry it can be either 0 or 1. There is no dynamic allocation for the buckets in this hash table. Whole memory is allocated at once so we must know in advance that how many entries we are going to insert. There would be few overflow buckets in the end of hash table and the last entry in a bucket contains index of the overflow bucket. W_0, W_1 and W_2 are the number of bits used in the field for storing index, delta index and index of overflow bucket respectively. Algorithm 7 contains the pseudo code for the operations on this hash table. For simplicity we have omitted the code to handle overflow. For overflow we keep an index pointer to first unused overflow bucket. It is initialized to 0 and when ever we need an overflow bucket we just use the overflow bucket pointed by index pointer and increment index pointer.

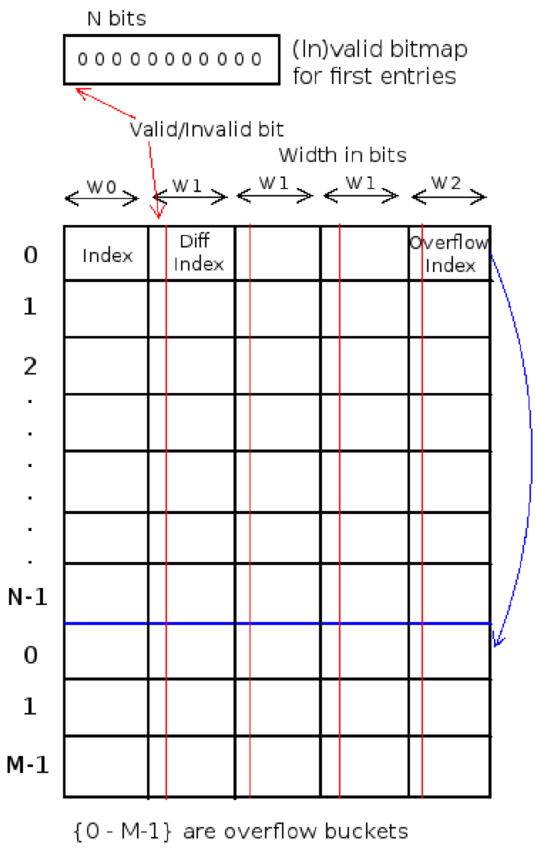

Figure 4.1: Bitmap based virtual hash-table

Advantages:

1. No memory wasted because of pointers.
2. No dynamic allocation for buckets so maximum usage of memory. The benefit of this is that we can keep load factor to be low (e.g. 2) to avoid overflows using same amount of memory that we need for a simple hash table of same size.
3. Logical resetting of hash table for reuse with fewer writes to PCM.

Difference from virtual partitioning: This bitmap based hash table is different from virtual partitioning in the sense that in virtual partitioning the number of partition were less so the ending part of all the list were present in the cache. And we can implement virtual partitioning easily by having indexes for each partition which points to next empty slot in each partition. Updating these index value for every insertion would not

Algorithm 7 Bitmap based virtual hash table operations

Initialization/Reset

```
1: Set all bits to 0 in bitmap array bitmap[];
```

Insertion

```
1: $h = hash(tuple[index].key)\%N$;
2: if bitmap[h]==0 then
3:     bitmap[h]=1;
4:     table[h][0]=index;
5:     table[h][1].firstBit=0; {Mark next to last entry as invalid}
6: else
7:     $i = 1$;
8:     $lastIndex = table[h][0]$;
9:     while $table[h][i].firstBit! = 0$ do
10:        $lastIndex+ = table[h][i].diffIndex$;
11:        $i = i + 1$;
12:    end while
13:    $table[h][i].firstBit = 1$;
14:    $table[h][i].diffIndex = index - lastIndex$;
15:    $table[h][i + 1].firstBit = 1$; {Mark next to last entry as invalid}
16: end if
```

Deletion: Not supported.

lead to writes to PCM because all these indexes will be present in cache.

Writes to PCM in insertion of n tuples: Since the first index will be small so we can keep all W_is to be 16 bits (2 bytes). So in best case the number of writes to PCM would be $(16 + 1) \times n$ bits. Actually the size of W_i depends on the size of relation for which we have to build relation. If it is very large then we may need to use 3 bytes to store index or delta index.

Use in internal simple hash join: In [5] authors has considered main memory resident database. They have shown that simple hash join (building a big hash table for only one relation) would perform poor than virtual partitioning based hash join. But if we use bitmap based hash table than simple hash join will perform better than virtual partitioning based hash join in terms of writes to PCM because inserting all tuples of smaller relation in this hash table will lead to writes almost equal to writes because of partitioning of a single relation. While in virtual partitioning based hash join we are partitioning both relations. The running time of simple hash join would be more because of more cache misses during probing phase.

4.2.1 Improving search performance

Bitmap based hash table has poor search performance as we have to access every tuple in a bucket during search. So to improve it we can treat each bucket itself as a hash table. Since total number of entries in a bucket would be between 2-10 so if we use 1 byte hash value with each entry it would be quite good because 1 byte hash value can have 256 different values so it is like a hash table having 256 buckets and there are only 2-10 entries in it so load factor is very low. But for this idea to work we need to use a separate hash function than that we used to insert the tuple index in hash table. So with this the entries in hash table will be <index, 1 byte hash value>.

4.3 Optimization for PCM

There is one another hybrid hash join algorithm which is better than grace hash join in Algorithm 6 . In it we keep first partition of relation R in memory. For simplicity we have mentioned only grace hash join algorithm. In this section we mention few PCM aware optimizations which are applicable to both hash join algorithms.

Use of bitmap based virtual hash table: We can reduce writes by using bitmap based virtual hash table in the probing phase of Algorithm 6 . It will lead to fewer writes to PCM and it will also avoid write expensive resetting operation of hash table.
Nested block loop join between partitions: One other possibility for write optimization is to drop the hash table completely. During the probing phase for each tuple in S_i we will do linear search in R_i. It would be slow so to speed it up we will consider the blocks of R_i (completely present in memory) and blocks of part of S_i in small input buffer such that the size of two block is less than cache size. Then we will do join between these two blocks only. Note that if complete R_i or part of S_i fits in cache then we will consider other block to be of size of only one tuple.

We can improve the speed of nested block loop join by a factor of 2 by dividing each output buffer used for storing tuples of a partition in two parts during the partitioning phase in Algorithm 6 . To store the tuples in a buffer we will use one more hash function (or few bits of previous hash function will work) to divide the tuples into two groups and we will store the tuples in buffer starting from two end points and a buffer will become full when the two sequence will collide. In each buffer we will have a 16 bit counter which will count the number of tuples in one group in a buffer. It is shown in Figure 4.2 .

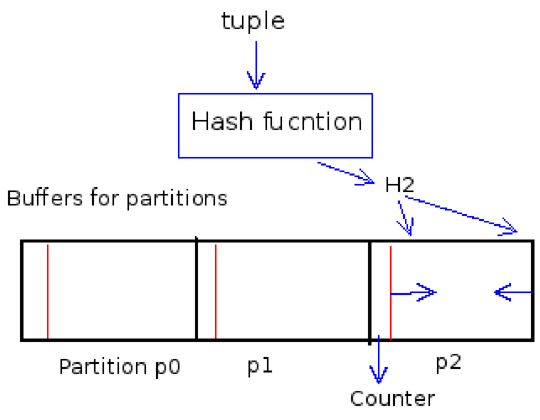

Figure 4.2: Optimization for nested block loop join during partitioning phase

During the probing phase (using search in this case) we will also calculate the hash value H_2 of a tuple in S_i and will search for match only in the part of buffers of partition R_i so it will reduce the search time by having very few writes to PCM.

Wear-leveling optimization during partition phase: The input buffer used in partition phase would have more writes than buffer allocated for partitions. So to make the distribution of writes uniform we can use the same technique that we used in section 3.5 for merge phase of external sort.

Chapter 5

Experimental Evaluation

We evaluated our proposed sorting algorithms through cycle-accurate simulator in this section. First we will describe the simulator used in the experiments.

5.1 PCM Simulator

We got the PCM simulator from the authors of [5] . They have extended the PTLsim [17] simulator. PTLsim simulator is a cycle accurate out-of-order simulator. PTLsim simulator models the details of a super-scalar out-of-order processor, including instructions decoding, micro-code, branch prediction, function units, speculation and a three-level cache hierarchy.

To count the writes to PCM they have extended it such that when data will be evicted from the L3 cache then we will compare the old data and the evicted data. So based on this we can count the number of modified bits, words and cache lines. They have also extended it to model 4 parallel PCM memory ranks. We will call extended simulator as PCM-PTLsim.

PCM-PTLsim does not model hard-disk and it just counts the number of total writes. There is no way to know writes to a particular word location. And when we do IO it does not count writes that happen to PCM because of data transfer from the DMA. So we have further extended the PCM-PTLsim to implement these things.

5.2 PCM Simulator Modification

We have done following modifications in the PCM-PTLsim:

5.2.1 Hard disk simulation

To calculate the IO cost we have traced all IO system calls. The traces have information which contains how much data has been read/write from/to which location of a file. Then using these traces we have implemented hard-disk simulator externally. We have simulated a very simple model of hard-disk in which we have assumed the location of files somewhere on hard-disk and then based on the current head position of hard-disk and read location in file we have calculated the total seek time. And based on the amount of data transferred we have calculated the data transfer time. We have assumed contiguous allocation for files.

$TotalSeekTime = SeekTime + RotationalLatency$
$DataTransferTime = TransferRate \times SizeOfData$
$IOtime = TotalSeekTime + DataTransferTime$
$IOcycles = IOTime \times CPUfrequency$

We have considered a hard disk which has only one platter having rotational speed of 7200 rpm and average seek time as 9ms. The data transfer rate is considered as 515 Mbits/sec. It has 36000 tracks each having 63 sectors. The size of each sector is 1024 bytes. Intially we assumed that the head location is at sector 0 (track 0). When we read/write data from/to a file, we first find out the starting point of data on hard disk e.g its track number and the sector number within track. If the head is not in the same track then we consider seek time to move head in the needed track in the total seek time otherwise we ignore it. We always consider rotational latency in total seek time even if the head is at the top of needed sector because it would has been moved (but within same track) from the ending position of last IO. Data transfer time is calculated as $TransferRate \times SizeOfData$. We have ignored the seek time to move head from one track to another during a large amount of data transfer because it would overlap with the data transfer from **disk-to-memory**, and **disk-to-buffer** data transfer rate is generally higher than what we have considered in our implementation as 515 Mbits/sec.

The hard disk simulation is not very precise but even with this if the IO cost of one process (P_1) is much higher than other process (P_2) then we are sure that P_1 will have more IO cost in real system too.

5.2.2 Wear leveling information

To get the wear-leveling information we have a counter for each 64 bit word of PCM. And we increment it for every write to that word. So using this we have calculated the standard deviation and maximum and minimum writes to any word. It was possible to implement this because for most of the experiments we have taken PCM size as 64 MB so we can allocate the memory for all counters for such PCM size.

5.2.3 PCM writes during write system call

When we write data to a file using write system call then we have assumed that DMA will transfer data from user buffer to hard-disk. Since data may not be updated in main memory so in the simulator first we write data to file and then based on how much data was actually written we flush that range of address in the cache hierarchy and count the writes because of flushing data.

5.2.4 PCM writes during read system call

When we read data from a file using read system call then we have assumed that DMA will transfer data from hard-disk to user buffer. In this case after data transfer the data in cache will not be valid so we have to invalidate that in the cache. Since we don't know in advance how much data will actually be read so we first read data in kernel buffer and then based on the actual amount of data read we invalidate that address range in the cache. Writes were counted to PCM by comparing the data present in the PCM user buffer and data in kernel buffer. In last we copy the data from kernel buffer to PCM user buffer.

5.3 Simulator Setup

1. Processor: Out-of-order, X86-64 core, 3GHz
2. CPU caches for sort:
    (a) L1D: 32 KB, 8-ways, 4-cycle latency
    (b) L2: 128 KB, 8-ways, 11-cycle latency
    (c) L3(DRAM): 2 MB, 16-ways, 39-cycle latency
3. CPU caches for join:
    (a) L1D: 32 KB, 8-ways, 4-cycle latency
    (b) L2: 256 KB, 8-ways, 11-cycle latency
    (c) L3(DRAM): 8 MB, 16-ways, 39-cycle latency
4. PCM:
    (a) 4 parallel ranks,
    (b) Read latency for a cache line: 230 cycles
    (c) Write latency per 8B modified word: 450 cycles

5.4 Sort

In this section we compare our multi-pivot PCM aware quicksort version 1 and version 2 with many existing conventional quicksort algorithms. Following are the algorithms that we used in the experiments:

1. Multi-Pivot PCM aware quicksort version 1 (pcm_qs_v1)

2. Multi-Pivot PCM aware quicksort version 2 (pcm_qs_v2)

3. Quicksort using cormen partition algorithm (cormen_qs) [6]

4. Quicksort using hoare partition algorithm (hoare_qs) [18]

5. Dual pivot quicksort (dual_pivot_qs) [8]

6. Quicksort in GNU standard C library (std_qs) [10]

Data set: We took TPCH lineitem table ([21]) as input data set. The width of each tuples of lineitem table is 141 bytes. We varied the table size as {4, 8, 16, 32, 48, 64} MB for internal in-place sort and {1×64, 2×64, ... , 6×64 } MB for external sort. Results of both internal and external sort are shown in same graphs. For internal sort table was already present in PCM and table was sorted in-place.

PCM size: PCM size was kept constant as 64 MB.

Effective cache size: Effective cache size was calculated as $\frac{\#ways-3}{\#ways} \times cacheSize$. Total $\#ways$ were 16 and we have taken $\#ways - 3$ instead of $\#ways - 2$ just to take care of the case that if we are accessing one more memory region other than executable code, stack and data.

In all the graphs pcm_qs_v1 and pcm_qs_v2 are almost overlapping. But in reality pcm_qs_v2 does little bit less writes than pcm_qs_v1 (around 0.9%) and pcm_qs_v2 takes very little more time to execute. Though the difference is not significant at all but still we have shown both of them. We will use the term pcm_qs to refer both of them.

Figure 5.1 shows the number of words modified by all the algorithm. pcm_qs is best and after this hoare_qs is best algorithm. For 64 MB internal sort pcm_qs provides 39% improvement and for 384 MB external sort pcm_qs provides 20% improvement over hoare_qs. But there is not that much space of improvement because if we have done chunk sorting using write optimal cycle sort then cycle sort would have provided 65%

and 34% improvement for the same. pcm_qs provides 47% and 26% improvement over std_qs for the same. Note that improvement of A over B is calculated as $100 \times \frac{B-A}{B}$ where $B > A$. Because otherwise hoare_qs does 63.9% and 24.4% more writes than pcm_qs.

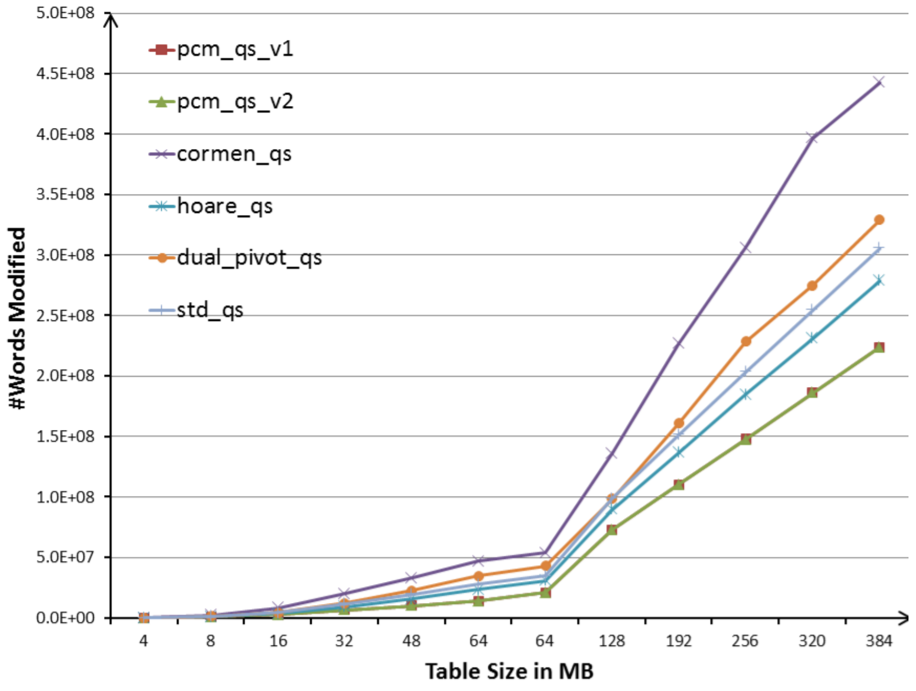

Figure 5.1: Modified Words

Figure 5.2 shows the bits modified in MB by all algorithms. The relative performance of all algorithm for modified bits is same as modified words. Because all algorithms do data manipulation at word level.

Figure 5.3 shows the average writes to a word. pcm_qs is best for this case also as it is obvious from total writes. The main purpose to show this graph is that if we kept decreasing the size of cache then for internal sort case practically pcm_qs have average writes around 2 because we kept increasing number of pivots chosen in multi-pivot partitioning while for other quicksort e.g. in case of hoare_qs it kept increasing. Though this thing is not visible properly in this graph. So if we have take DRAM cache as 1 MB instead of 2 MB then pcm_qs would have provided even more improvement over hoare_qs.

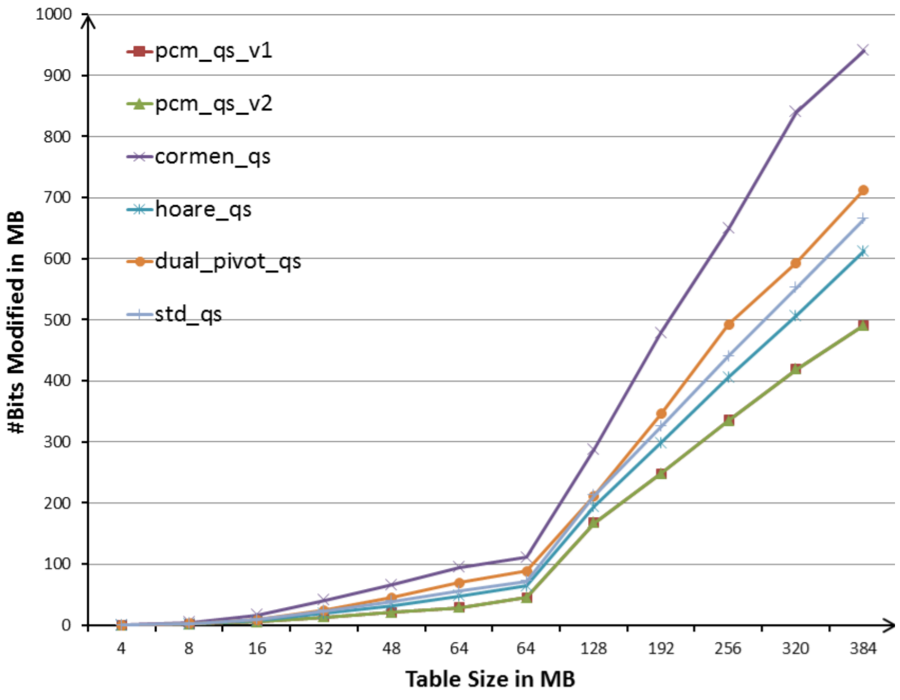

Figure 5.2: Modified bits in MB

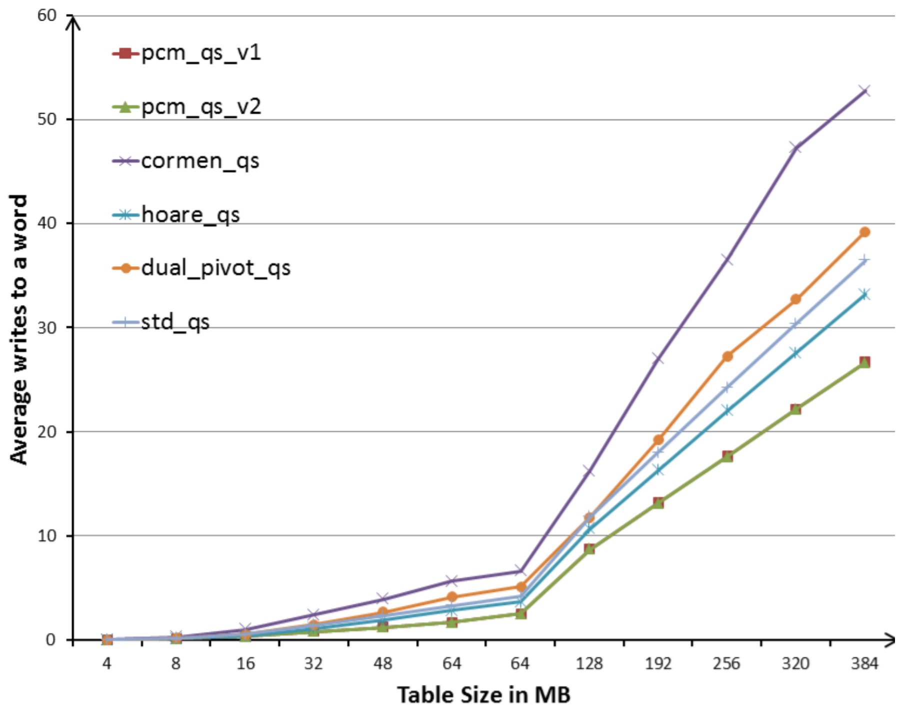

Figure 5.3: Average writes to a word

Figure 5.4 shows maximum writes to any word during sorting and Figure 5.5 shows writes standard deviation. In both of these graphs pcm_qs performs much better than other algorithms. For maximum writes pcm_qs provides 75% and 73% improvement over hoare_qs respectively for internal 64 MB and external 384 MB. For standard deviation pcm_qs provides 53% and 58% improvement over hoare_qs respectively for internal 64 MB and external 384 MB. Hence it is cleat that pcm_qs has much better distribution of writes than other algorithms. There is little drop in the maximum writes when we go from 64 MB internal case to 64 MB external case. Theoretically there should not be such drop but it could happen as the seed value was different and all algorithm are randomized algorithm except std_qs. And there is no such drop for std_qs. There is drop in standard deviation when we go from internal 32 MB case to 64 MB even if total writes are increasing. It could happen as standard depends on the variation in writes but not on total writes.

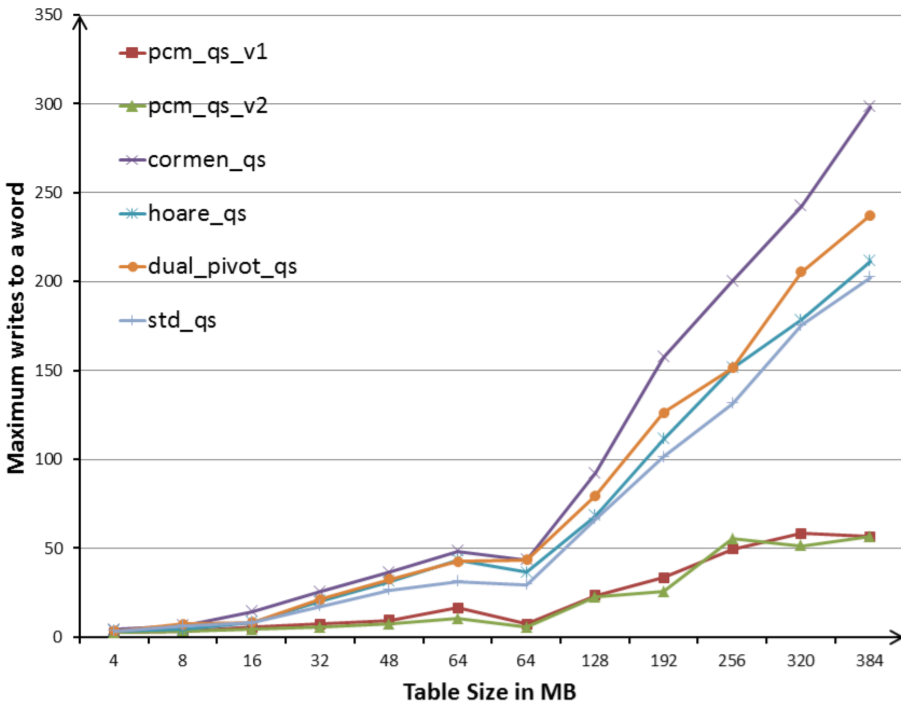

Figure 5.4: Maximum writes to any word

Next we compare the running time of all these algorithms. During the IO transfer (even for any system call) the simulator get paused so its simulated CPU clock, hence

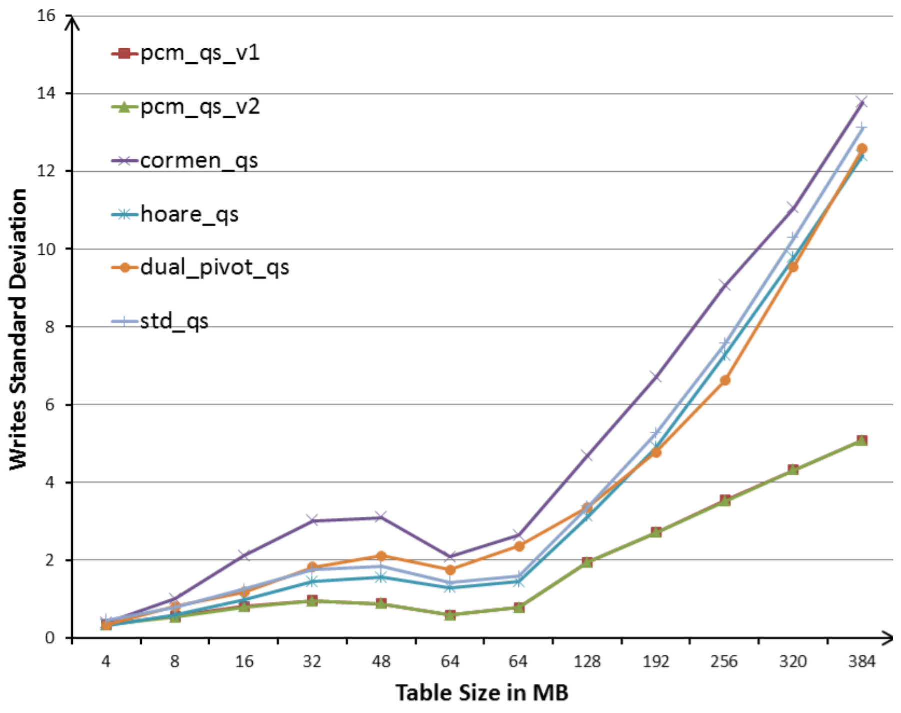

Figure 5.5: Writes Standard Deviation

execution time for programs doing IO is measured by number of cycles taken by CPU and the IO cycles estimated by the hard-disk simulator. Figure 5.6 contains the CPU cycles for all algorithms. The IO cycles are shown in Figure 5.7 . The total number of cycles (CPU+IO) are shown in Figure 5.8 . It is clear that pcm_qs is much faster than other algorithms. There is very little difference between the IO cycles of the algorithms. Actually pcm_qs take little more IO cycles than others because in pcm_qs we have done the wear-leveling optimization in which we have smaller buffer size. Both algorithm transfer same amount of data but pcm_qs takes little more time because of transfers in smaller sizes. Since for internal sort there was no IO so for that IO cycles of all algorithms is 0. pcm_qs_v1 is little bit faster than pcm_qs_v2 because by reducing the pivots by logical merging of partitions save writes but waste little work already done by CPU because now we would be having larger sub problems in comparison to pcm_qs_v1.

Figure 5.9 contains the instructions executed by CPU during the execution for all algorithms. pcm_qs have little bit more instructions than hoare_qs but difference is small. So even after having little more instructions (or even if we call equal) pcm_qs is much

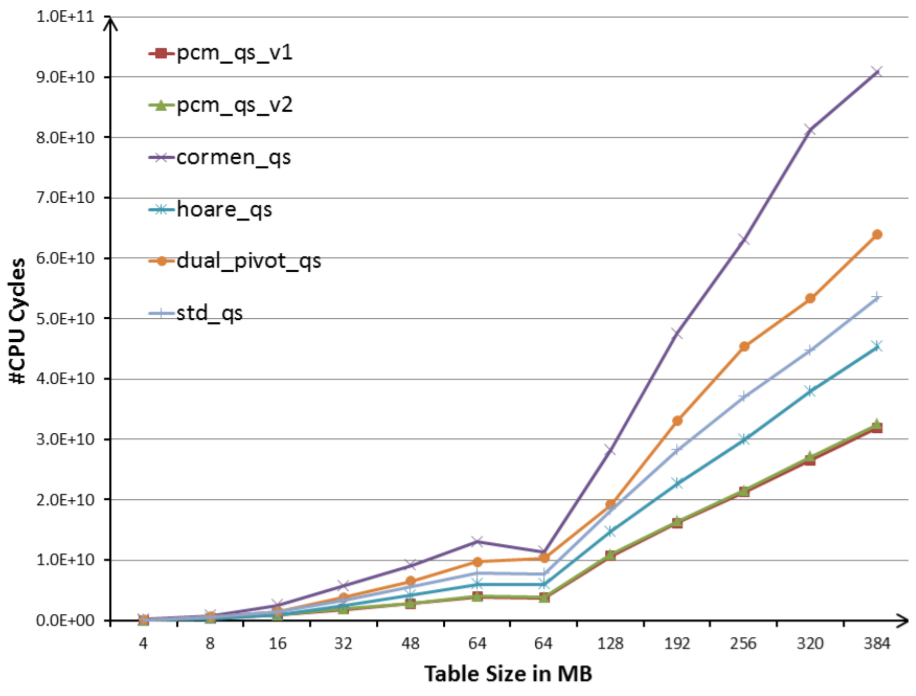

Figure 5.6: Execution time (CPU Cycles)

faster because it does less writes. Since writes to PCM are slow so in case of hoare_qs CPU would be waiting for memory so it is slow.

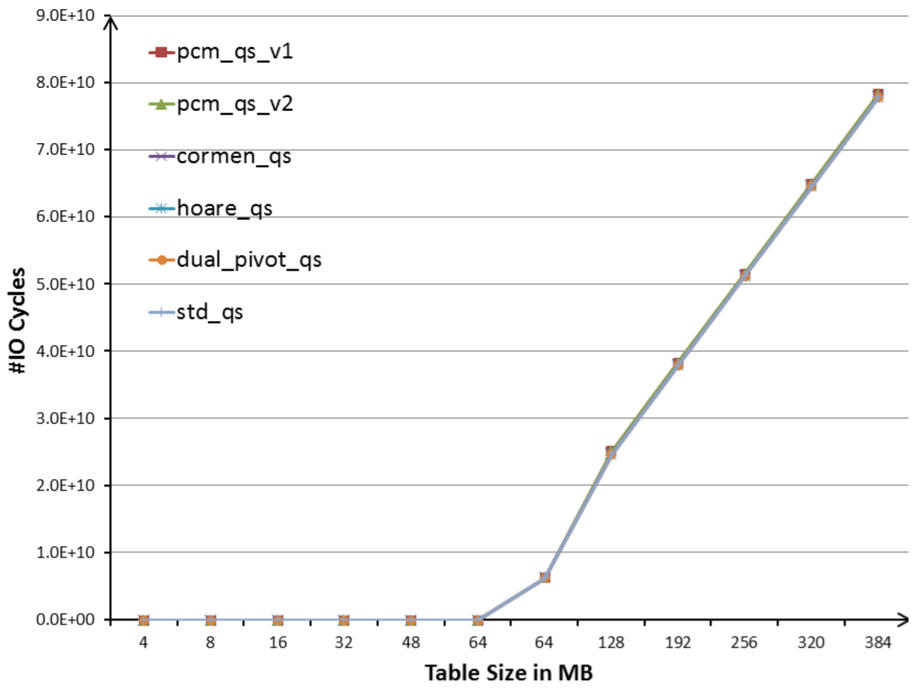

Figure 5.7: Execution time (IO Cycles)

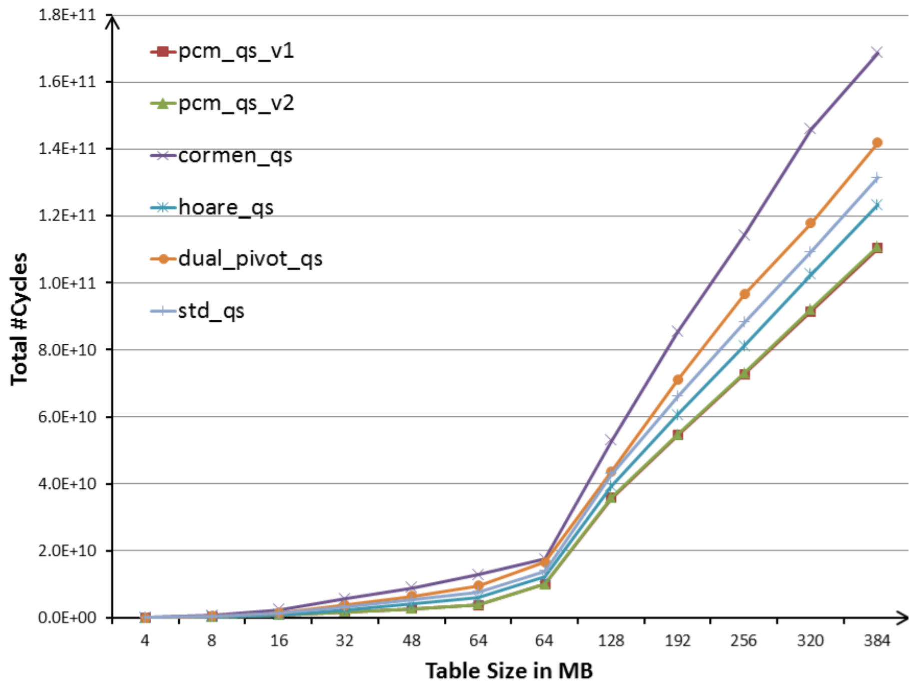

Figure 5.8: Execution time (Total Cycles)

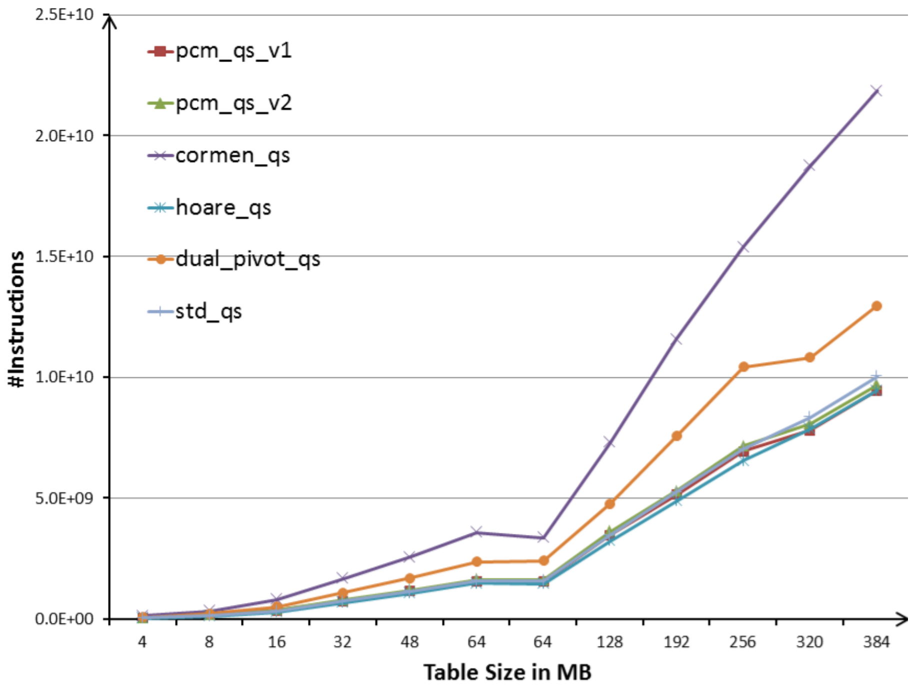

Figure 5.9: Instructions Executed

Effect of wear leveling optimization: To check the effect of wear leveling optimization we sort external 384 MB table using pcm_qs_v2 using conventional merge phase. Following are the results that we got:

pcm_qs_v2	with optimization	without optimization
Modified words	224532070	224263472
Standard deviation	5.1	12.1
Maximum writes	57	65

The writes to PCM is also most same in both cases as expected. Maximum writes increased little bit. But there is huge increase in standard deviation and it is same as it was for other algorithms. Hence wear leveling optimization effect the standard deviation a lot and it also shows that in other algorithms the main cause of increase in standard deviation was merge phase of external sort but not the internal sort subroutine used.

Effect of data distribution and tuple width: Data distribution does not have effect on performance as long as there are not too many key duplication. The problem with too many key duplication is that when we will choose many pivots randomly then some of them may be same so effectively when we will sort pivots and remove duplicates we will end up with fewer pivots so the probability that the sub-array size will be less than cache size will decrease.

We have did experiment by varying the width of tuple. We used the same TPCH lineitem table. We remove other attributes from tuple to reduce its size. Changing tuple width does not effect the performance. We got the same relative performance. The only difference was that the CPU cost got increased when tuple size was small because now we have more number of tuples.

5.4.1 Effect of 1-pass multiple pivot quicksort

In our knowledge there is no known multiple pivot quicksort algorithm in which the number of pivots varies. There is a dual pivot quicksort. The design goal of dual pivot quicksort was to improve the running time of quicksort for the data sets for which 1 pivot quicksort performs poorly because of bad pivot choice. If we want to do partitioning for more than 1 pivot in 1 pass then it would lead to increase in writes to PCM because we don't have information of final sub-arrays boundaries in advance so we don't know where to put a wrong element when we found such element during partitioning. So generally in this case we have to keep shifting whole sub-arrays, like we do in cormen partitioning and dual-pivot partitioning. That is why these algorithms has much more writes than others. There is very little probability that there would exist a multiple pass multiple pivot quicksort algorithm because multiple pass algorithm has large constant factor and in past the main design goal was to reduce running time but not the writes.

5.5 Join

Though our main purpose was to design algorithm for disk resident case but the overall improvement by using bitmap based hash table for disk resident join would not be much because when we did write analysis then writes because of hash table was not the dominating factor. In this section we compare our bitmap hash table based simple hash join algorithm with the virtual partitioning based join algorithm given in [5]. We have called bitmap based hash table without 1 byte hash value as version 1 and bitmap based hash table with 1 byte hash value as version 2. We compared both version of bitmap hash table. We implemented internal join operation in which both relation are in main memory. We have considered join of relation R and S where joining attribute was the primary key of R and it is foreign key in relation S. The tuple size of both relation was taken to be same. For a given R and number of matches per tuple of R we have taken relation S such that it contains all the primary keys of R repeated as many times as number of matches per tuple of R. So if there are 100 tuples in R and #match per tuple of R is 2 then S will contain 200 tuples and each key in R will occur exactly twice in S. In the implementation of bitmap based hash table we have used 3 bytes to store indexes and have stored index directly instead of delta.

Data set: Both relations were randomly generated. Size of relation R was kept fixed as 50 MB. Then based on the tuple width we have calculated the number of tuples in R and then we generated these many random unique keys. Relation S was generated by repeating each key in R as #matchPerTuple of R times. The values of other attributes were generated randomly. Both relations were shuffled randomly.

PCM size: PCM size was taken to be large enough to hold both relations and other data. So we have taken PCM size as 1 GB.

Output tuples were not stored anywhere, we have assumed that upper layer operators will consume them so we have just incremented the counter for each output tuple. We have done experiments in two configurations. First we kept the size of tuple fix as 64 bytes and varied the #match per tuple of R as {1, 2, 4, 6, 8}. In second configuration we kept the #match fixed as 4 and varied the tuples size as {16, 40, 64, 88, 112} bytes. In all the figures we have used BITMAP_V1 and BITMAP_V2 for join algorithm using bitmap based hash table version 1 and version 2 respectively and VIRTUAL_PART for virtual partition based join algorithm.

First we describe the experimental results for configuration 1. Figure 5.10 show the number of words modified by the algorithms. Bitmap hash table based algorithms has much more number of modified words and it is constant as expected because number of tuples in R relation were fixed. But number of words modified is not right criteria to check the wear out of PCM because it is actually bits modified. And our algorithm operates on bits. More modified words just show that our algorithm has spent more time in writing data to PCM because write latency depends on word. But that is a problem only when CPU is waiting for memory. If we remove the cache completely still the writes to PCM for our algorithms will be same but the running time will increase a lot. So because of presence of cache running time is not slow as we will see next. Figure 5.11 show the bits modified by the algorithms so now it is clear that our algorithms does much less number of writes to PCM once the size of S is greater than twice of size of relation R. In [5] it was shown that simple has join does more writes than VIRTUAL_PART till the #match equals to 8 so bitmap based hash table provides good improvement.

Figure 5.12 shows the running time of all algorithm. So it is clear that even after having so many cache misses during probing phase our algorithm is little faster than VIRTUAL_PART because VIRTUAL_PART execute more instructions and does more writes to PCM. Figure 5.13 shows the instructions executed by all algorithms. It is clear that BITMAP_V2 is little bit fast and execute little bit more instructions than BITMAP_V1 hence 1 byte hash value improve the running time but it is not that much. So bitmap hash table without 1 byte hash value is good enough for internal join.

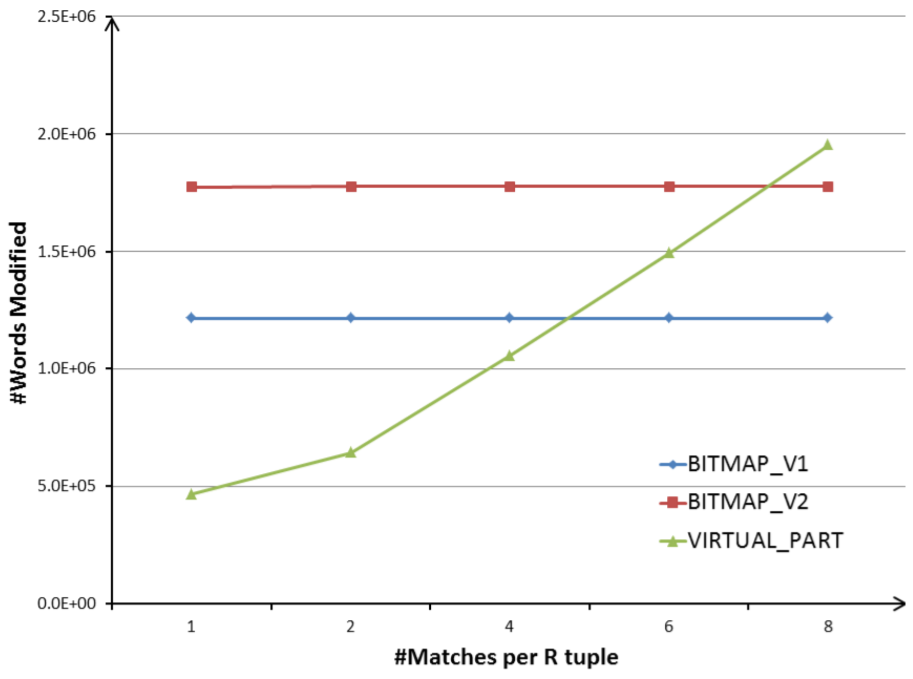

Figure 5.10: Modified words

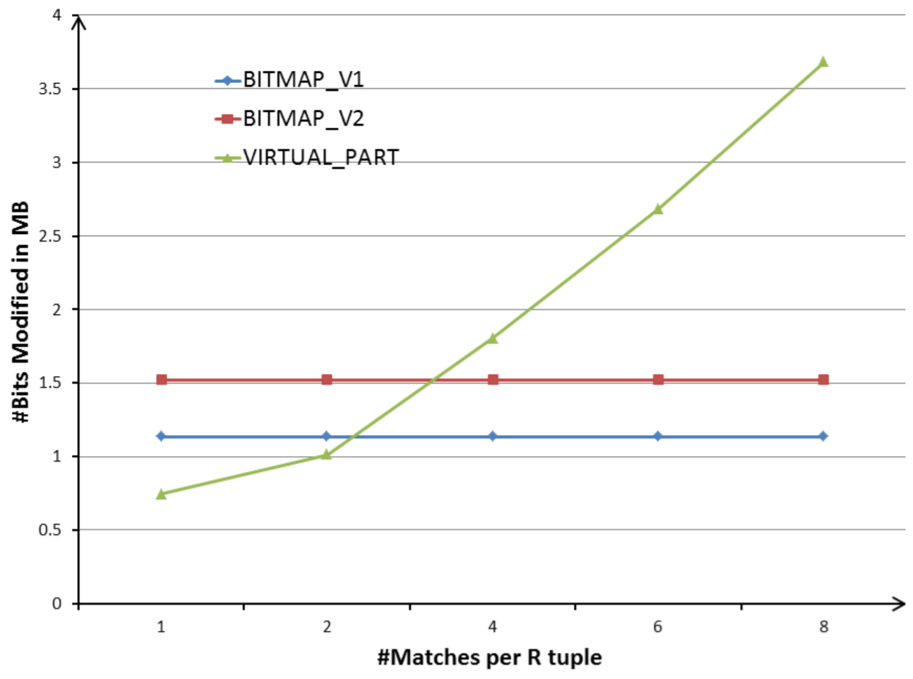

Figure 5.11: Modified bits in MB

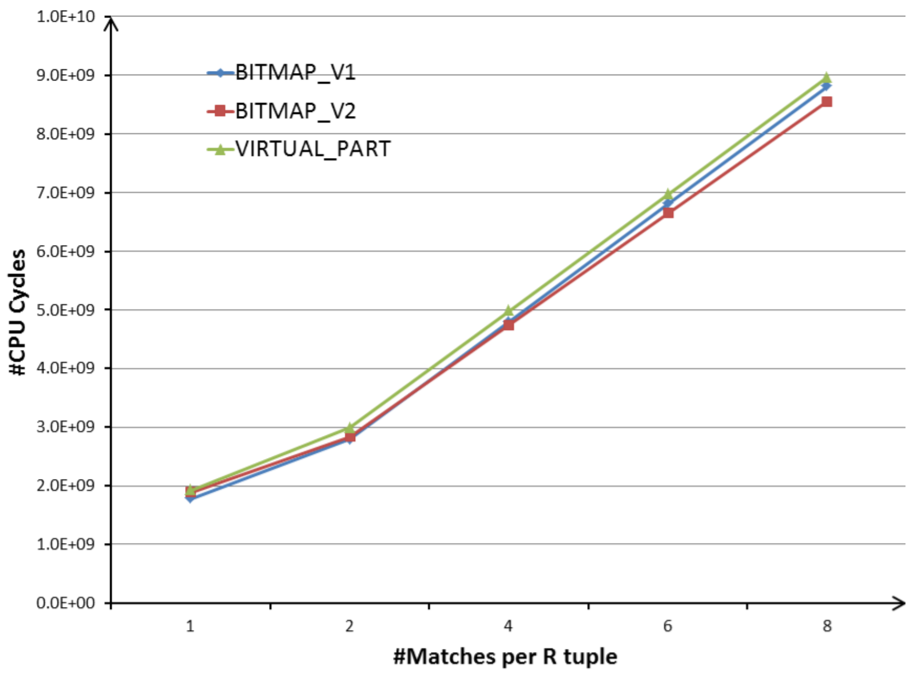

Figure 5.12: Execution time (CPU cycles)

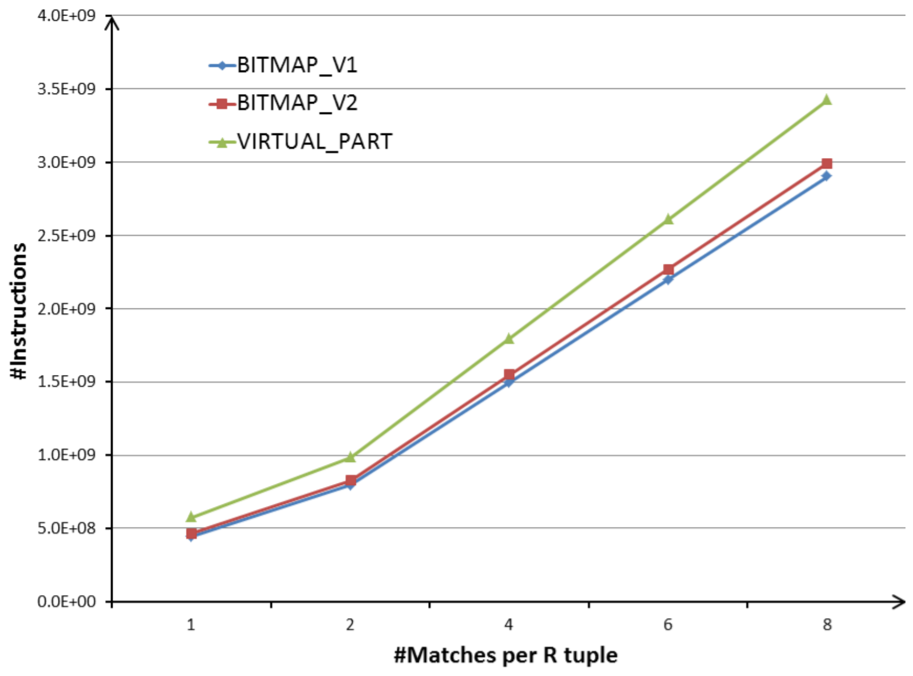

Figure 5.13: Instructions Executed

Next we show experimental results for configuration 2. Figure 5.14 shows the running time of all algorithms. Figure 5.15 shows the number of instructions executed by the algorithms. For these two parameters there is not much difference in the relative performance of algorithms as it was initially.

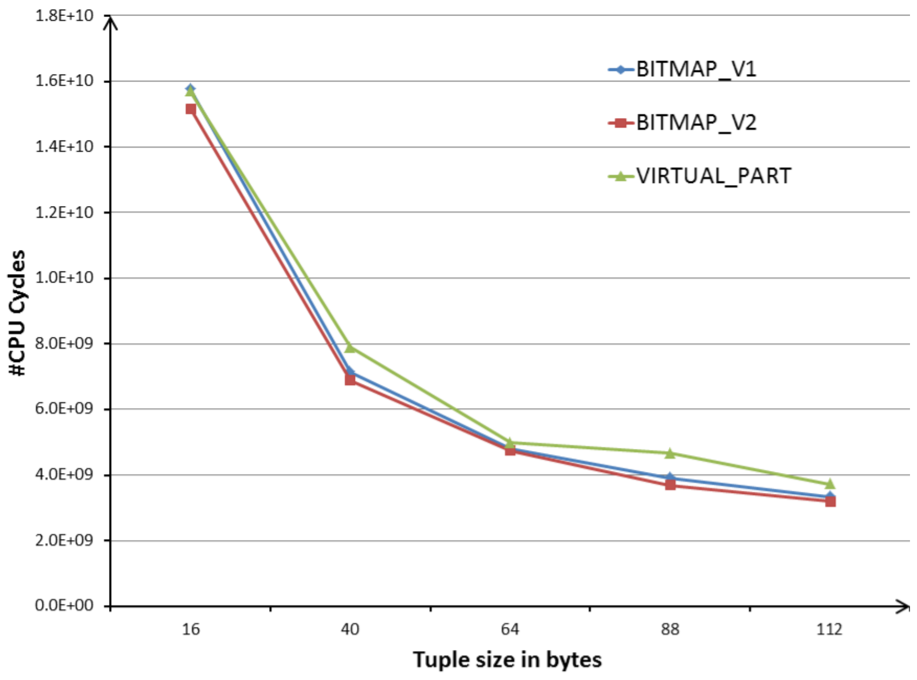

Figure 5.14: Execution time (CPU cycles)

Figure 5.16 and Figure 5.17 shows the number of words modified and number of bits modified in MB. So in this case also our algorithm has more number of words modified but less bits modified for smaller tuple size. But as we increase the tuple size the number of bits modified by all algorithms is almost same. It is because as the size of tuple is increasing the number of tuples is increasing. And in our implementation of bitmap hash table we are using 3 byte for each entry but in reality we could have used less bits as explained in the theory of bitmap based hash table. In that case our algorithm would have performed better than VIRTUAL_PART algorithm.

We haven't shown write distribution information for join because it is clear in theory that writes will be uniformly distributed and maximum writes to a word would not be more than 1-2.

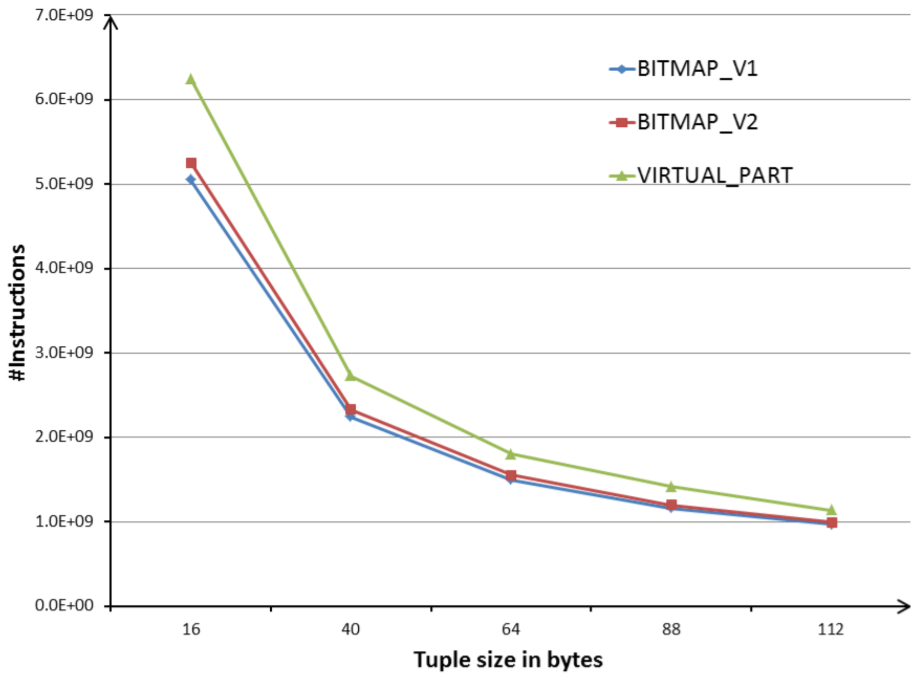

Figure 5.15: Instructions Executed

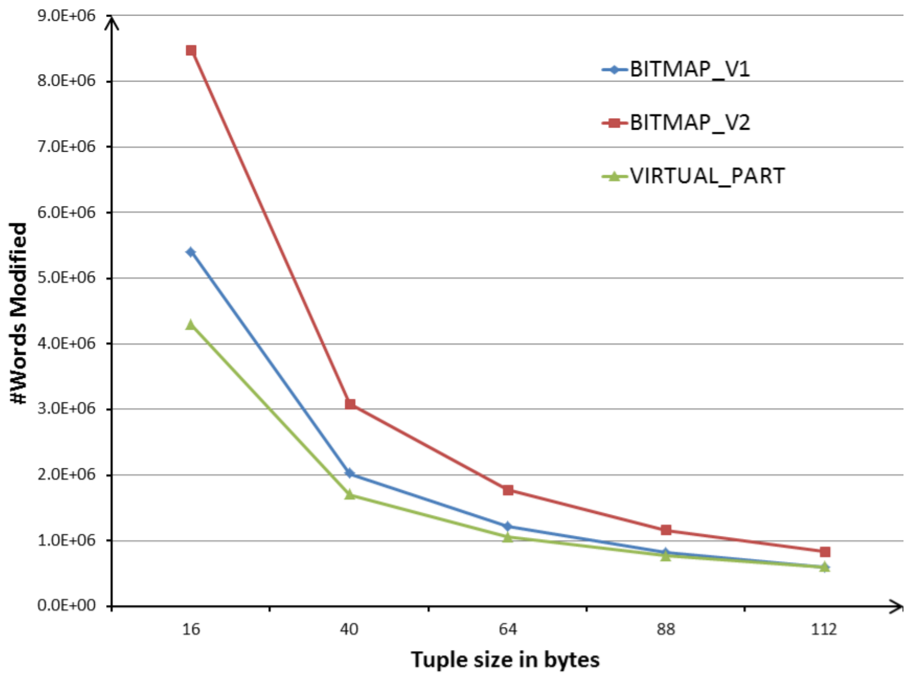

Figure 5.16: Modified words

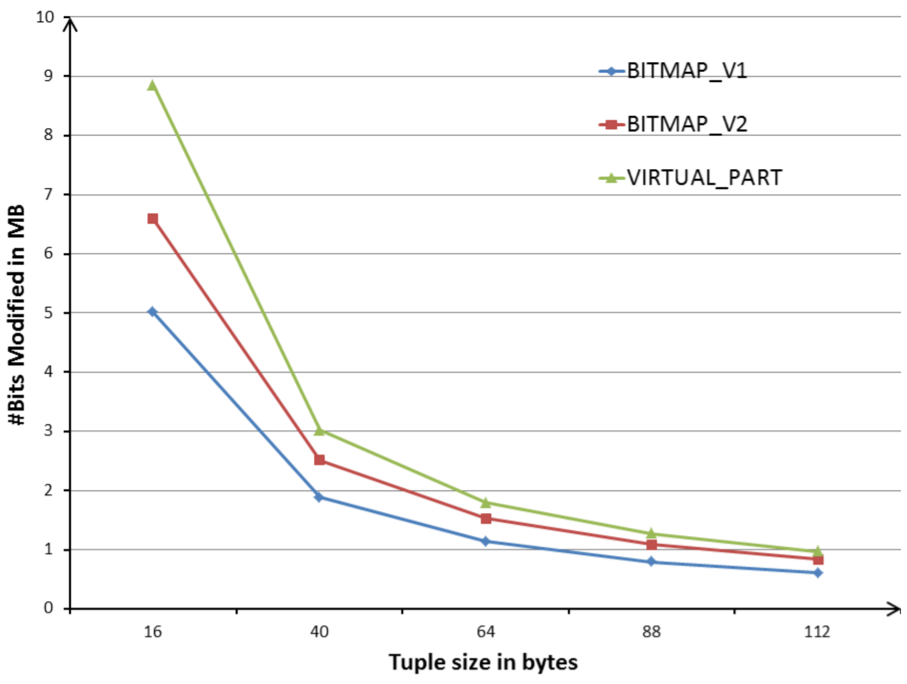

Figure 5.17: Modified bits in MB

Chapter 6

Sorting in DRAM_EXPLICIT system model

A sorting algorithm was suggested in [15] for system model having explicit control over DRAM. They have considered external merge sort algorithm. They have focused on the algorithm used to sort a chunk of size of available memory size in step 3 of Algorithm 1 . Pseudo code for their algorithm to sort a chunk C_i is given in Algorithm 8 .

We designed an algorithm for this system model which is better than algorithm in 8 in terms of both running time and writes to PCM. We have also considered the algorithm used to sort a chunk C_i in step 3 of Algorithm 1 . The main idea of our algorithm is to use external merge sort logic between PCM and DRAM. Algorithm 9 contains the pseudo code for this algorithm.

For this system model during merge phase of external merge sort (1) we will use DRAM as output buffer.

6.1 Comparison

Since both Algorithm 8 and 9 are subroutine of Algorithm 1 hence whichever subroutine will be better the corresponding external merge sort algorithm will be better. So we will compare these two algorithms for running time and writes to PCM.

Algorithm 8 Advance PCM-Aware Sort

```
1: Take a sample of data from chunk $C_i$ on disk into DRAM;
2: Construct a histogram $h$ in DRAM;
3: Based on this histogram $h$ make $k$ buckets and find their boundary element value in DRAM; {Here $k = \frac{M}{m}$, hence ideally each bucket size= $m$}
4: Write the buckets boundary information in a small part of PCM;
5: Divide the PCM memory into $k$ parts, one for each buckets.
6: Read data from chunk $C_i$ on disk in sub-chunks of size $m$ in DRAM;
7: Process each memory sub-chunk $c_j$ in DRAM and write its elements to their corresponding buckets in PCM; {Partition Phase}
8: If there is an overflow in the buckets because of error in histogram $h$ then write overflow tuples to DRAM or to a temporary file $T_1$ depending on the availability;
9: Sort all overflow tuples in DRAM by their <bucket id, key> value.
10: Bring each bucket data (sub-chunk $c_j$) in DRAM and sort it in DRAM; {When we bring first bucket data in DRAM we will simultaneously copy the overflow tuples in space for first bucket in PCM.}
11: Sort each bucket by bringing it into DRAM;
12: If there are overflow tuples for this bucket in PCM then merge them with this sorted sub-chunk and write it to temporary file $T$;
```

6.1.1 Writes

Merge Based PCM-Aware sort does exactly $sizeof(C_i)$ writes to PCM. While Advance PCM-Aware sort does $sizeof(C_i) + sizeof(histogram) + \#overflowtuples \times sizeof(int)$ writes to PCM. Hence our algorithm does less writes to PCM.

6.1.2 Running time

In Merge Based PCM-Aware sort we sort k sub-chunks which takes $k \times m \times log_2 m$ time. And during multi-way merge phase we have implemented a min heap which contains $< key, sub - chunk\_id >$ as elements. The key is the join key in the first tuple of sub-chunk that has not been output yet. To chose the minimum tuple from the smallest tuple of remaining tuples in each sub-chunks we have to delete the top most entry of heap and then insert the next tuple in the heap from the same sub-chunk. We have mixed the delete and insert operation hence it will take only $log_2 k$ time to output one tuple. Hence total cost for this is $M \times log_2 k$. Hence total running time $= M \times log_2 m + M \times log_2 k$.

In Advance PCM-Aware ideally we sort k buckets which takes $k \times m \times log_2 m$ time.

Algorithm 9 Merge Based PCM-Aware Sort

```
  Local Sort Phase
1: $k = \frac{M}{m}$;
2: for $i = 1$ to $k$ do
3:     Read $i^{th}$ memory chunk of size $m$ from disk chunk $C_i$ into DRAM.;
4:     Sort this memory chunk in DRAM;
5:     Write it to PCM; {$m$ writes to PCM}
6: end for
   Multi-way Merge Phase
7: Simultaneously merge all $k$ sorted memory chunks into DRAM;
8: Whenever DRAM becomes full write it to temporary file $T$;
```

Then during partition we have to find bucket corresponding to each incoming tuple so we will perform a binary search in the histogram which will take $log_2 k$ time hence total cost for this is $M \times log_2 k$. During sampling we have to sort the sample data and then construct histogram. When there will be overflow, which would always (though very little) be the case, then we have to merge the sorted tuples in DRAM with overflow tuples in PCM. Note that there is no extra space to merge or we do in-place merge so it will take time and may even lead to writes to PCM. But author in [15] has ignored this fact and has used **fopen and fwrite functions** ([3]) and write one tuple at a time as output using **fwrite** . So either the IO cost will be very high or *fwrite* maintains an internal buffer (it is the case most of the times but may not true always). And if the number of overflow tuples is so large that DRAM can't handle them then it would be even more costly. So total running time $= M \times log_2 m + M \times log_2 k + samplingIOtime + histogramCreationTime$.

The IO cost in our algorithm will be lower than that in Advance PCM-Aware sort because we would be doing IO always in size of m while in Advance PCM-Aware IO may be in smaller sizes. So it is clear that our algorithm is faster and writes efficient than Advance PCM-Aware sort.

6.2 Experimental Evaluation

We have compared our merge based algorithm with the Advance PCM-aware sorting algorithm given in [15] . The input table was unsorted TPCH lineitem table. DRAM size was 2 MB and PCM size was 64 MB. We varied the size of input table from 1×64 MB to 6×64 MB by increasing the number of tuples to sort. While implementing the algorithm in [15] we give advantage to them that we already build the histogram using sorted answer and in the sampling phase we just read only $4 \times k$ random elements from the chunk C_i's file and spend no time in building histogram. And we have also given 10 KB extra buffer for output tuples when there is overflow tuples in PCM for a sorted sub-chunk in DRAM. We have introduced very little error in histogram. The error was introduced by a factor of 400 tuples which means that the actual size of bucket was randomly chosen to be $m + rand()\%400$. Note that this error is very small because 2MB DRAM can hold around $14K$ tuples. In this case the number of overflow tuples were bounded by 2000 tuples. But still our algorithm is better than Advance PCM-aware sort in all metrics. In all the figures Adv-PCM is used for external sort using Advance PCM-aware sort as subroutine and Merge-based sort is our algorithm.

The number of writes to PCM is shown in Figure 6.1 . Adv-PCM does little more writes than our algorithm but the difference is small and the two graphs are overlapping.

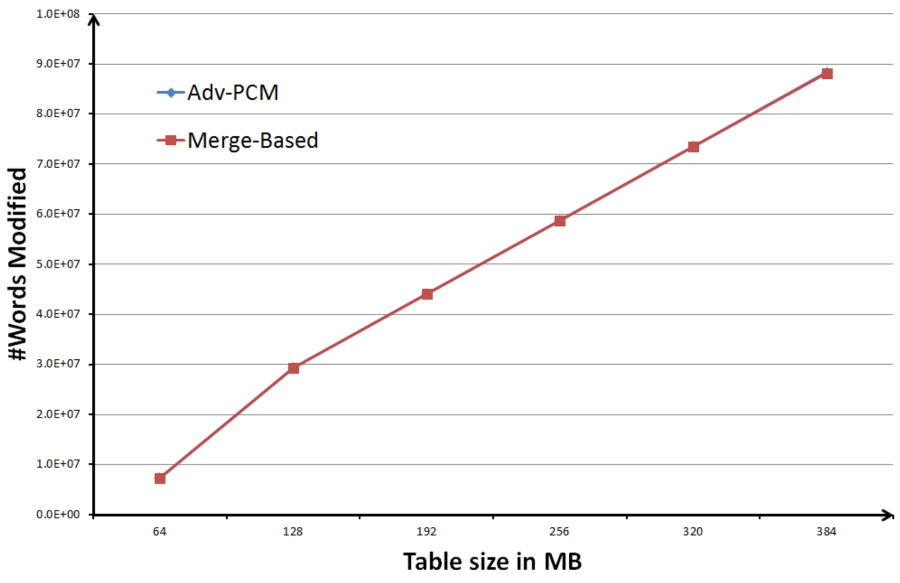

Figure 6.1: Modified words

The number of instructions executed is shown in Figure 6.2 . Adv-PCM executes more instructions because of sampling and overflow handling.

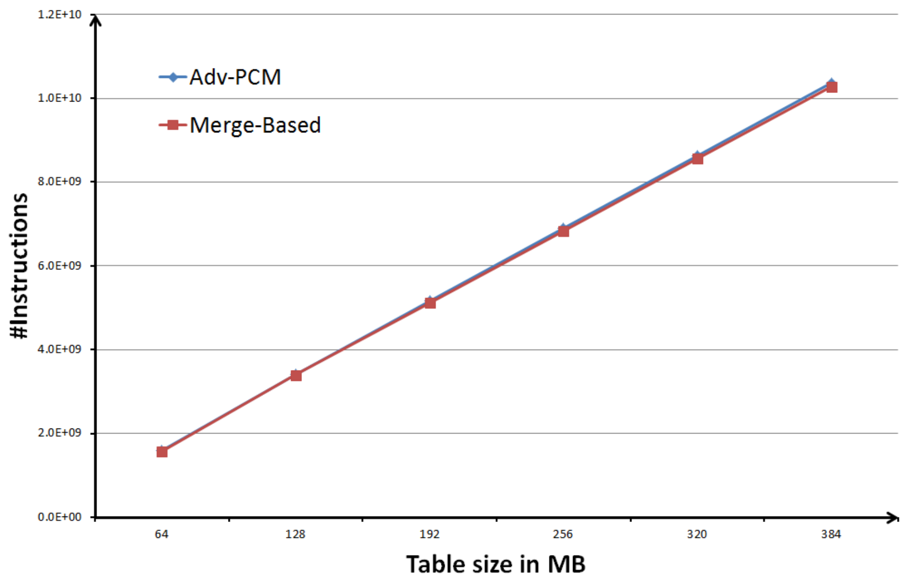

Figure 6.2: Instructions Executed

CPU cycles are shown in Figure 6.3 . There is not much difference between the CPU cycles though our Merge-based algorithm is little faster.

IO cycles are shown in Figure 6.4 . Adv-PCM takes much more time in IO than Merge-based algorithm. The total cycles are shown in Figure 6.5 . So our algorithm is better than Adv-PCM in terms of both writes and running time. Note that we have given advantage to Adv-PCM algorithm otherwise writes to PCM would be even more.

We haven't shown the write distribution information as it is clear in theory that both algorithm will have almost uniform distribution of writes to PCM.

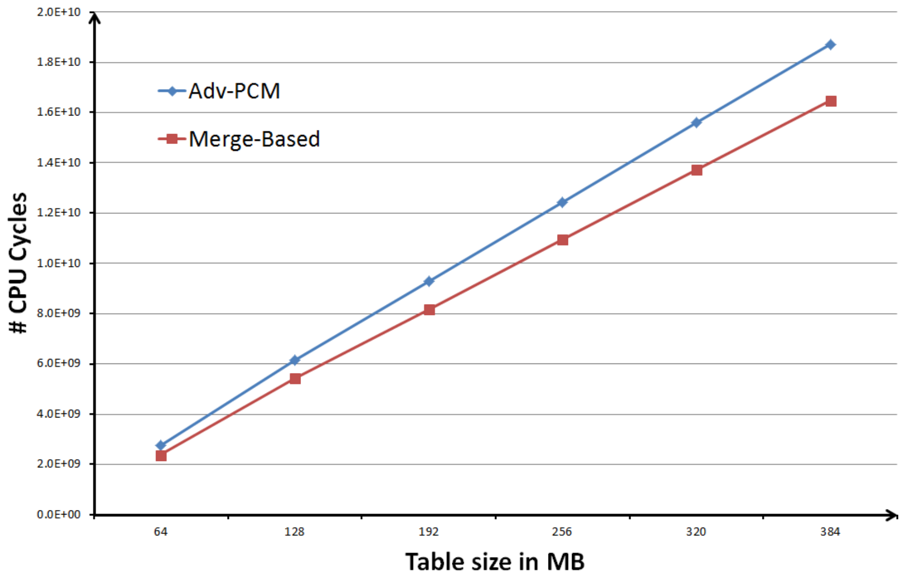

Figure 6.3: Execution time (CPU cycles)

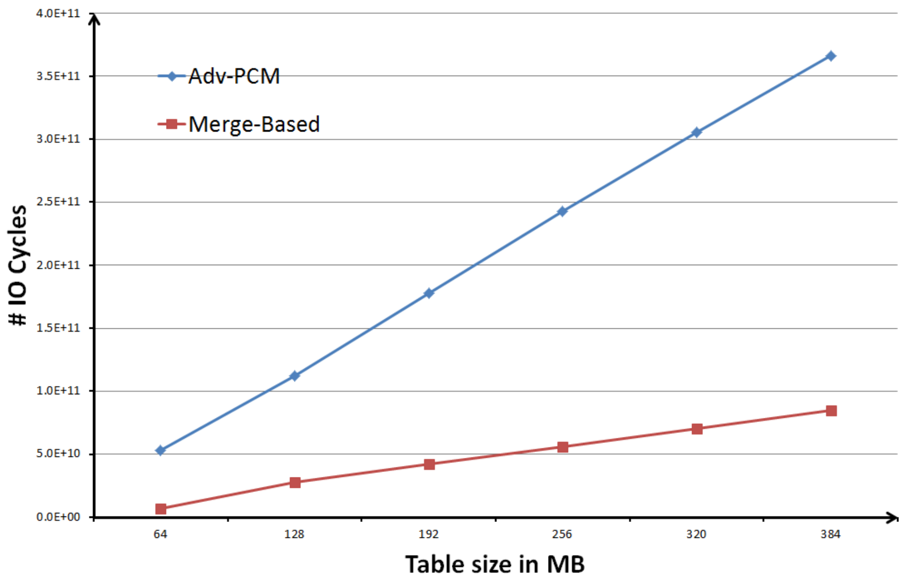

Figure 6.4: Execution time (IO cycles)

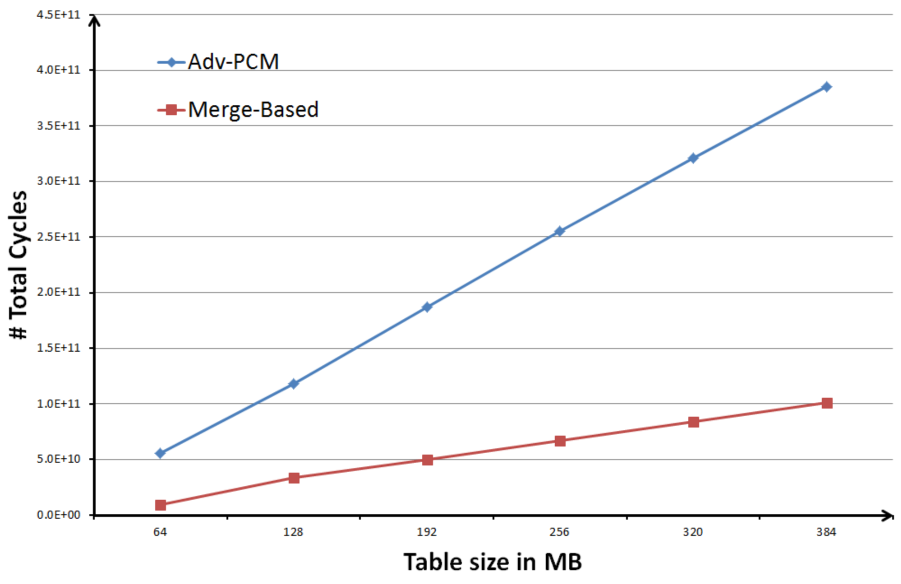

Figure 6.5: Execution time (Total cycles)

6.3 Other algorithms of theoretical interest

In this section we describe few algorithms that we have not implemented but may be helpful in some other situations. Note that all algorithm that were given for DRAM_CACHE system model are also applicable in this system model e.g. multi-pivot PCM aware quicksort. But these algorithms are not best for external sort as we can read data in DRAM directly. One can also suggest not to use PCM at all during external sort which would be the best algorithm in terms of the writes to PCM but such algorithm will be very slow as it will have very high IO cost.

6.3.1 Internal worst case $2 \times n$ writes algorithm

Suppose we are given an array in PCM and we don't have any other extra memory available and we have to do in place sorting. Then we can use multi-pivot PCM aware quicksort to sort data. In this section we describe an algorithm which choose pivot value such that the sub-arrays are guarantee to fit in cache. So it has worst case $2 \times n$ writes to PCM. The pseudo code is given in Algorithm 10 . In this algorithm we are assuming that all the elements are unique. The main idea of the algorithm is that we will do local sorting of chunks in DRAM and we will choose as many pivots from each chunk as the number of chunks. This will guarantee that the partitions size will be smaller than cache size.

Proof of claim that all count $\leq m_1$: For each chunk we can defined local partitions as the elements between its two consecutive pivots. For example as shown in Figure 6.6 we have 3 chunks (here k is 3). So we have local partitions in each chunk. The size of each local partition is $\frac{m_1}{k}$ as the size of each chunk is m_1. If we take all 12 pivots and sort them then in the partitions of the complete array due to these pivots there would never be elements from more than 1 local partition of same chunk. Means suppose we have two consecutive pivots p_1 and p_2 in 12 pivots then the elements between p_1 and p_2 will never be from more than 1 local partition of same chunk because if we have elements from more than 1 local partition of same chunk then these 2 or more partitions were

Algorithm 10 Worst case $2n$ writes algorithm

```
1: Calculate effective $k$ such that $k = \frac{n}{m-k\times(k+1)}$;
2: Calculate effective DRAM $m_1 = m - k \times (k+1)$; {Let's call part $m_1$ of DRAM as
   data part and part $k \times (k+1)$ as meta part}
3: for $i$=1 to $k$ do
4:     Read $i^{th}$ chunk of size $m_1$ in data part;
5:     Sort this chunk; {Time: $m_1 \times log_2(m_1)$}
6:     Write $1^{st}$, $1 \times \frac{m_1}{k}^{th}$, $2 \times \frac{m_1}{k}^{th}$, ... $(k-1) \times \frac{m_1}{k}^{th}$, ${m_1}^{th}$ element from the sorted chunk
       to meta part; {Total $(k+1)$ elements}
7: end for
8: Sort $k \times (k+1)$ elements in meta part; {Time: $k^2 \times log_2(k)$}
9: Treat these $k \times (k+1)$ elements as pivots;
10: Scan the array in PCM and count the number of elements in each partition; {Time:
    $n \times log_2(k^2) = n \times log_2(k)$}
11: Write array of pivots in DRAM as <pivot,count>; {Space required= $2 \times k \times (k+1)$}
    {Claim: All count $\leq m_1$}
12: By scanning the $k \times (k+1)$ pivot merge them such that summation of their count
    $\leq m_1$; {Claim: After this we can have maximum $2 \times k$ pivots}
13: Shuffle elements in array in PCM such that each element reach to its correct bucket,
    where these bucket are created by new $2 \times k$ pivots; {Time: $n \times log_2(k)$} {Claim:
    Maximum $n$ word writes to PCM}
14: Sort each bucket by bringing it into DRAM and then write answer back to PCM;
    {Claim: Maximum $n$ word writes to PCM}
```

separated by some pivot q in that chunk. Since we have elements greater than and lesser than q between p_1 and p_2 hence q will also be there between p_1 and p_2. Hence p_1 and p_2 can not be consecutive pivots as there is a pivot between them. Hence the maximum size of a partition because of all 12 pivots can be $\frac{k \times m_1}{k} = m_1$. So all partitions size(count) $\leq m$.

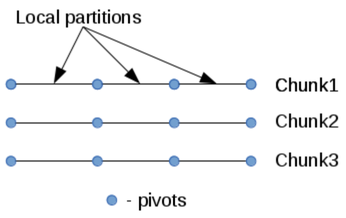

Figure 6.6: Local partitions. All chunks are sorted locally.

Duplicate keys case: The argument that all the partitions size will be less than or equal to m_1 just need that all selected pivots are unique. When the pivots will be repeated than the size of a partition can be large. But whenever the size of a partition will be larger than m_1 then the extra elements in that partition will be only the elements which has same key value as pivots. So If we drop all the elements from array which has key same as pivot value then partitions size will always be less than or equal to m_1. Hence to handle key repetition case we have to define partitioning in a different way. If pivots are $\{p_1, p_2, p_3\}$ then there would be 7 partitions {values $< p_1$, values $= p_1$, values $> p_1$ and $< p_2$, ... , values $= p_3$, values $> p_3$}. The size of partitions where values are equal to pivot value can be larger than DRAM but we don't need to sort them as all keys are equal and in other cases the partition size will be less than DRAM. Hence we can do sorting in worst case $2n$ writes even when there are duplicates.

Out of place write optimal sort: If we have to sort a table and we have extra memory to write answer then we can use the logic in Algorithm 10. After getting all the final pivots instead of having a swap phase we will read the elements of each partition in the DRAM. First we will read the elements of 1^{st} partition then 2^{nd} and so on. To read the elements of 1^{st} partition in DRAM without swap phase we will scan whole table and if an element of table is between the pivots of 1^{st} partition then we will keep it in DRAM otherwise we will drop it. So without any writes to PCM we can have all partitions individually in DRAM. After that we will sort the 1^{st} partition and write it to the output memory. Then we process the 2^{nd} partition and append it to the output. So we can do sorting by doing only n writes to PCM which is optimal. The only problem with this algorithm is that it has large constant factor.

Chapter 7

Conclusion

We designed algorithm for sort operator in PCM database which is much better in terms of writing to PCM then conventional algorithms. We found that reducing writes to PCM also reduce the running time of program because CPU would not be waiting for the memory. We found that our multi-pivot PCM aware quicksort is faster than conventional sort algorithm for PCM system. Our algorithm has better write distribution hence it is good for PCM wear-leveling also.

We proposed bitmap based virtual hash table which is good for PCM system because operations on this table are not write expensive. We found that using bitmap based hash table in simple hash join algorithm improve the write performance of join and it performs better than virtual partitioning based join algorithm in [5] when larger relation is larger than twice of the size of smaller relation. Bitmap based hash table can also be used in hash join for disk resident data case.

We also proposed sorting algorithm for system mode having explicit control over DRAM which is better than sorting algorithm in [15] in terms of running time and also does little bit fewer writes to PCM. The performance of our algorithm will always be same regardless of type of data but the worst case of the algorithm in [15] can be quite bad.

We also discover that to design algorithm for PCM system we can use following techniques:

1. Make use of multiple passes to reduce writes to PCM. Means to save writes we will pay by extra reading.

2. Do manipulation at bit level.

3. Make use of indexes for virtual clustering/partitioning/hashing and to avoid pointers.

4. Try to combine writes to a location that were distributed over long time interval to a short time window so that most writes got consumed by cache.

Chapter 8

Future Work

We have categories the future work into two part. One which need very little effort and time and other that need more. We have described both of these categories in following sections.

8.1 Minor work

In future we want to do following things:

1. PCM is implemented as having 4 parallel memory ranks. It has a queue for incoming read and write request. When CPU read/write data from PCM and it is not in cache we submit request and wait for it to complete (if CPU don't have any thing else to do). But when we do IO then for writes to PCM due to flushing of the cache and writes by DMA, we do not submit any request in the queue. We just do it quickly. We have assumed that the waiting time will overlap with the IO time. But this assumption may not lead to perfect timing. At least we should submit write request to PCM queue for flushing cache case. So in future we want to do so any want to analyze our overlapping assumption during IO.

2. When we read data from disk we read data first in kernel buffer and then invalidate the user buffer address range in cache. Next we calculate the difference in words

and bits between old data in user buffer and new data in kernel buffer. But actually the data in user buffer is not completely old. Because the caches are logical cache. We never copy data from PCM to cache, it is always in PCM. So we are comparing old updated data with new read data. For example suppose a word in PCM intially has value 10. Now we updated it to 20. So it will be updated in cache but not on PCM until there is eviction. Now suppose we read value 30 from disk into the word and there was no eviction in cache. So ideally we should compare 10 with 30 to get #bits modified because actual value that will be present in PCM in a real system will be 10 but not 20. But because the way we have implemented caches in simulator, PCM has value 20 and we are comparing 20 with 30. Interestingly it does not have any effect on the results in this thesis but in theory for other program it could. For example a program can be written such that it read 1 MB data from file (suppose file contains random data) in a buffer and then update all values to bits complement of them. But these updated values will be in cache (assume cache size $>$ 1MB). Now the program again read 1MB data from same file. So now in this the actual writes to PCM should be only $1\times(1024\times 128)$ 8 byte words writes (just because of reading data for the first time) but simulator will show $2\times(1024\times 128)$ words writes to PCM.

3. To show the superiority of multi-pivot PCM aware quicksort we would like to do experiments having even smaller cache size e.g. 1-2% of PCM size. And even we want to check the effect of change in the size of cache on the performance.

8.2 Major work

In future we want to do following things:

1. We want to prove expected number of writes for multi-pivot PCM aware quicksort formally.

2. We will improve the join operator further because there is still possibility of improvement in terms of writes to PCM.
3. We want to design algorithm for other operators and then will implement complete query execution plan.
4. We will do cost modeling of all these new algorithms so that query optimizer can decide which algorithm to choose for processing.
5. **System model:** We suggested that DRAM_CACHE system model is practically more feasible but still there are few problems with this assumption. Following are the problems with this assumption:
   (a) We says that existing software can be used as it is with this system model but still "can be used" does not mean that "it is good to use". Because if we look at the external sort, just because of writes by reading table twice and writing twice cause the improvement in performance of multi-pivot quicksort to be low. And in the simulation when we did IO the data transfer was always done directly to/form user buffer (we have used term kernel buffer for implementing functionality in simulator but in reality it has nothing to do with simulated system). But if we take any existing OS, it always implement buffer cache. Which actually read data in advance from files in kernel buffer and delay the writing of data to file by writing data in memory assigned for buffer cache. So in such scenario writes would amplify by a factor of 2 and there would not be any significant effect of the algorithms that we are using to sort chunks in memory.
   (b) In complete thesis we never talk about non-volatility of PCM. For sorting and join we do not need non-volatility but DRAM_CACHE model does not allow us to use non-volatility. Because for it a process must be able to access a region in PCM across multiple execution. And in this system model (by saying that we will use existing OS) we are mapping PCM to virtual address space of a

process as we does with DRAM main memory. Means which physical page will be mapped to which virtual page is not fix across multiple execution of a process. So how a process will use non-volatility. Hence there is no way to use non-volatility with DRAM_CACHE system model using existing software specially OS. We do not want to say that DRAM_EXPLICIT model is good. It also has similar problem.

There is actually one more possible system model. It is shown in Figure 8.1. In it we have PCM at the same level as DRAM. But it is not the actual main memory. Means for a program which is unaware of existence of PCM, the system is like having only DRAM as main memory. We have not shown the path of data transfer to/from PCM. But ideally it should be such that we can transfer data to/from PCM from/to DRAM without CPU interventions (but obviously controlled by CPU) and CPU can also access PCM directly. To access PCM a program could map a range of PCM, identified by physical address in PCM, at any where in its address space using some new system call. So this model allow to use the non-volatility of PCM and it is most flexible and feasible. Such configuration is possible because this is how memory of graphic card is accessed. We can map the graphics card memory to address space and can get C pointer to that memory location and use it normally to read write.

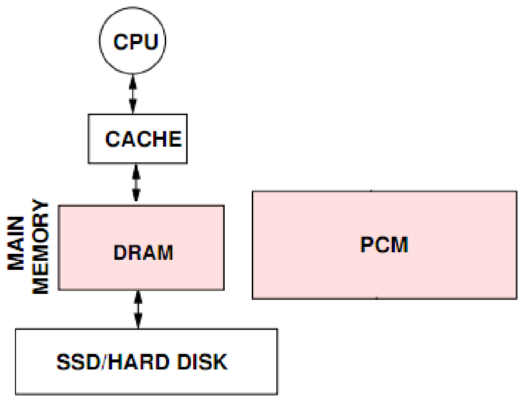

Figure 8.1: PCM Explicit System model

We can call this system model as PCM_EXPLICIT. Since we are not sure about the possibility of this system model so in future we would like to analyze this system model.

Bibliography

[1] D. Bausch, I. Petrov, and A. Buchmann. Making cost-based query optimization asymmetry-aware. *DaMon*, May 2012.

[2] P. Bonnet and L. Bouganim. Flash device support for database management. *CIDR*, January 2011.

[3] C file input output. `http://en.wikipedia.org/wiki/C_file_input/output`.

[4] J. Chen, R. C. Chiang, H. H. Huang, and G. Venkataramani. Energy-aware writes to non-volatile main memory. *HotPower*, October 2011.

[5] S. Chen, P. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. *CIDR*, January 2011.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, third edition, 2009.

[7] Cycle sort. `http://en.wikipedia.org/wiki/Cycle_sort`.

[8] Dual-Pivot-Quicksort.
`http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf`.

[9] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. PCMLogging: Reducing transaction logging overhead with PCM. *CIKM*, October 2011.

[10] GNU C Library. `http://www.gnu.org/software/libc/`.

[11] IEEE data engineering bulletin vol. 33 no. 4. December 2010.

[12] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. *ISCA*, June 2009.

[13] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *ISCA*, June 2009.

[14] Mathematical induction. `https://en.wikipedia.org/wiki/Mathematical_induction`.

[15] V. V. Meduri, Z. Su, and K. L. Tan. A write efficient PCM-aware sort. *DEXA*, September 2012.

[16] C. Mohan. Implications of storage class memories (SCM) on software architectures. *HPCA Workshop, Bangalore, India*, January 2010.

[17] PTLsim. `http://www.ptlsim.org/`.

[18] Quicksort. `http://en.wikipedia.org/wiki/Quicksort`.

[19] M. K. Qureshi, S. Gurumurthi, and B. Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool Publishers, first edition, 2012.

[20] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase change memory technology. *ISCA*, June 2009.

[21] TPCH. `http://www.tpc.org/tpch/`.

[22] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. *ISCA*, June 2009.