What can do Emerging Hardware for your DBMS Buffer?

Cheikh Salmi[1], Abdelhakim Nacef[2], Ladjel Bellatreche[1], and Jalil Boukhobza[3]

[1] LIAS/ISAE-ENSMA, Poitiers University
F-86961 Futuroscope, France
(salmi.cheikh, bellatreche)@ensma.fr
[2] National High School for Computer Science, Algiers, Algeria
a_nacef@esi.dz
[3] Univ. Bretagne Occidentale
UMR 6285, Lab-STICC, F-29200 Brest, France
boukhobza@univ-brest.fr

Abstract. The spectacular development of business intelligence applications (BIA) build around the data warehousing technology increases the demand on query performance of DBMS hosting extremely amount of data. In such context a high interaction among queries exists since they share a large number of intermediate results. This is due to the fact that BIA uses relational schemes such as a star schema in which each join passes through the fact table. The decision to cache these intermediate results in the traditional buffer becomes a crucial issue since it depends on the size of the buffer and the number of intermediate results candidate for caching. Optimizing queries requires new algorithms and emerging hardware. As flash memory is more and more adopted in mass storage systems thanks to its performance characteristic, we rely on it to become a part of the buffer and cache some intermediate results. In this paper, we first propose to couple the RAM and Solid State Drive to respond to the problem combing buffer management and query scheduling sub problems. Secondly, a cost model for evaluating the quality of buffering data and scheduling queries is given. Based on this cost model, an algorithm is given to solve our joint problem. Finally, we describe some results obtained thanks to a simulator we developed. It shows that our proposal enhances the performance of SQL queries by up to 85%.

1 Introduction

Big data brings the need for storing; managing and querying in an efficient way the huge amount of data issued from various heterogeneous data sources to increase the decisional power. Relational data warehousing is one of the most valuable technology to deal with this challenge. The analytical process is usually performed by the means of complex queries involving a large number of joins and aggregations. The particularity of these queries is their ability to interact with each others, since they share large number intermediate results such as joins and selections. This phenomena is well known by the multi-query optimization community [13]. This interaction has been largely exploited to define optimization structures such as materialized views [16]. Recently, some research efforts have been done toward studying the contribution of query interactions to manage the buffer of the DBMS. These efforts assume a traditional architecture of the DBMS in terms of storage: Hard Disk Drives (HDD) for storing the huge amount of data and the RAM for buffering data to speed up queries.

Nowadays, this assumption is no longer valid, since we are witnessing spectacular development of emerging hardware. As a consequence, efficiency dimensions have evolved from classical raw performance metrics such as IOPS (I/O per second) and throughput to energy consumption. One key cross-cutting technology allowing leveraging the big storage challenge is flash memory. Indeed, flash memory is considered as the most important technology change pertinent to data centric computing [12]. Flash memory market explosion is mainly due to consumer electronics democratization but is no more limited to this area. Indeed, it is flooding the mass storage system market such as cloud applications thanks to its performance and energy characteristics. Major companies such as Dropbox, Google, and Facebook have turned part of their storage systems to flash [4]. Flash memory has just celebrated its 25th birthday (1988-2013) after its creation in Toshiba labs and it is already shipped with almost 8 times more gigabytes than DRAM (in 2011). It became the process technology leader for memory fabrication and miniaturization. In fact, NAND flash memory revenues have hit a new record as of \$5.6 billion (a growth of 17%) in the fourth quarter (as compared to the third quarter of 2012). Flash memory cost, in terms of dollars per GByte, is higher than traditional HDD. This is the main reason why flash memory is not thought of as a replacement to HDDs at the mean term, even though many solutions have emerged containing exclusively flash memory based storage systems. Going from this observation, designers have tried to integrate efficiently flash memory in the memory hierarchy mainly in three ways [2]: (1) as an extension of the system memory (RAM), (2) as a cache between system memory and HDD and (3) at the same level as HDDs to constitute a hybrid storage system.

In this paper, from an architecture point of view, we use the flash memory as an intermediate buffer between the system memory and the HDD to extend the buffer size, so the traditional HDD becomes the last alternative when looking up for data. Based on this architecture, a buffer management technique at database operator level (intermediate query results) that enhances the RAM buffer by relying on flash memory has to be developed. Under this consideration, we consider one joint problem including two main sub problems related to database performance: buffer management and the query scheduling ($\mathcal{BMQSP}$). It has been studied in the context of data warehousing with traditional hardware including HDD and RAM [9]. To the best of our knowledge, none work dealing with the combined problem in the context of emerging hardware.

[4] http://www.wired.com/wiredenterprise/2012/06/flash-data-centers/

In order to validate our proposition, we developed a tool to simulate the query processing in a hybrid storage environment coupling SSD and HDD. This simulator integrates a co-habitation strategy between these two devices, especially when one or both devices are full.

The paper is organized as follows. In Section 2, we review some related works on the joint problem combining buffer management and query scheduling. Section 3 presents fundamental concepts regarding SSD, multi query optimization and formalization of the joint problem. In Section 4, our proposal is given where cost model quantifying our solution is developed. Buffer management and query scheduling strategies are also detailed. Section 5 presents our experimental studies by the means of a simulator. Section 6 concludes our paper.

2 Related Work

The buffer management problem is one of the pioneer database problems [3, 4, 15]. It spans all database generation: traditional database, object oriented database, data warehousing, etc. With the development of emerging hardware, some research efforts have been proposed to either leverage the existing solutions or propose new buffering techniques. The query scheduling problem got less attention compared to buffer management problem. In this section, we overview the most important works related to the $\mathcal{BMQSP}$ and to the place of SDD in database storage systems. The $\mathcal{BMQSP}$ has been studied in the context of Relational Data Warehouse (RDW) [9, 14]. The authors in [14] identified the interdependency between buffer management and query scheduling and proposed several issues related to the combination of buffer and query scheduling problem by considering multi query optimization. In [9], a formalization of $\mathcal{BMQSP}$ is given and its complexity is studied. To find a best solution, three main algorithms were proposed in the context of HDD: hill climbing algorithm, genetic algorithm and bee-inspired algorithm. The basic idea behind these algorithms is to exploit the interaction between queries and consider them as candidate for the buffer. An intermediate node is cached if and only if it can be exploited to schedule queries.

Many research projects have been launched in response to the following question: how can flash memories be used in the context of database systems [6, 5, 8]? Three main directions have been followed: (i) a real integration place of flash in the storage memory hierarchy, (ii) to couple database buffer pool and flash and (iii) the identification of buffering candidates. The work of [6] proposes to use SSD as an extension to database buffer pool and exploit them to manage the dirty pages. Before being evicted from the buffer, the authors propose to first store the dirty pages on the SSD to avoid the communication with the HDD. Three eviction strategies were proposed. Clean-write: never write dirty pages to SSD, dual-write: write an evicted dirty page to both HDD and SSD and finally, lazy-cleaning in which dirty pages are first written to SSD and later copied from the SSD to HDD (lazy updates). Authors in [17] focused on flash memory as a write cache for databases stored on conventional hard disk. In this case, when dirty pages are evicted from the memory buffer pool, they are first written on the flash-based-cache and later propagated to hard disk, applying some replacement strategies to reduce the amount of data written to flash.

In [5], the problem of SDD buffer has been studied in the OLTP environment with an important issue which is the improvement of recovery time and data integrity after a crash or normal database restart. Metadata about the contents of the SSD buffer are stored on the so called: SSD buffer table. This table can be reconstructed using transactional log files and is periodically flushed in an asynchronous mode. A caching algorithm at the granularity of subtuples is proposed in [8]. The algorithm partitions a database table vertically and then caches the subtuples. Updates are gathered until a page eviction occurs, subtuples are then flushed to a single area on the flash memory (a page or more) which reduces the amount of data written to flash memory. In [10], a caching system using flash memory as a layer between the main memory buffer pool and the magnetic disk is proposed. A theoretical cost model and a data strategy were also proposed.

To summarize, the buffer management and the query scheduling problems have been studied in both environments with well-established objectives: (a) reduction of query processing cost, (b) reduction of communication between database buffer pool and the HDD and (c) reduction of recovery time. Note that the multiple query optimization is ignored when SSD and database buffer pool are combined.

3 Background

In this section, we present basic concepts related to flash memory based drives and the multi query optimization. A formalization of our joint problem in the context of SSD and HDD coupling environment is also described.

3.1 Flash Memory based Drives

Flash memory is a type of EEPROM (Electrically Erasable and Programmable Read only Memory) based on floating gate transistors. It is structured as follows: a chip is composed of one or more dies; each die is divided into multiple planes. A plane is composed of a fixed number of blocks, each of which encloses a fixed number of pages that is typically a multiple of 64. Current versions of flash memories have between 128 KB and 1024 KB blocks (with pages of 2, 4, or 8 KB). A page consists of a data space and a metadata Out-Of-Band (OOB) area containing page state, information on Error Correction Code (ECC), etc. Three operations can be carried out on flash memories: reads and writes, which are realized on pages, and erases, which are performed on blocks.

Even though NAND flash memory presents some very interesting characteristics as discussed earlier, it also has some limitations caused by its internal intricacies. The main constraints are: 1) Write/Erase (W/E) asymmetry: writes are performed on pages whereas erasures are realized on blocks. 2) Erase-before-write limitation: a costly erase operation is necessary before data can be modified

[2]. 3) Limited number of W/E cycles: the average number is between 5000 and 10^5 depending on the technology used. After the maximum number of erase cycles is achieved, a given memory block becomes unusable. Finally, (4) the I/O performance for read and write (and erase) operations is asymmetric.

The Flash Translation Layer (FTL) is a hardware / software layer intended to overcome the aforementioned limitations: 1) The erase-before-write and the W/E granularity asymmetry constraints imply that data updates should be performed out-of-place. Hence, the logical-to-physical mapping scheme, which is a critical issue, is used to manage these updates. Mapping tables are stored in an embedded RAM and in some cases in the flash memory itself. 2) Out-of-place data updates require the use of a garbage collector to recycle blocks enclosing invalid pages in order to recover free space. 3) To minimize the limitation on the number of W/E cycles, FTLs try to evenly distribute the wear over the memory blocks. This wear leveling prevents some blocks wearing out more quickly than others.

Today's SSDs integrate many advanced techniques to achieve better performance in terms of lifetime and throughput. Among these techniques, one can mention: multi level parallelism (inter channel, chip, plane) to increase the throughput, deduplication, data compression, and flash-specific buffers to decrease the amount of written data thus increasing lifetime.

SSD performance depends highly on FTL efficiency and internal parallelism, but in general, the time taken for reading a flash page is in the order of microseconds, the writing time is in tens to hundreds of microseconds (us) and erasing time is from hundreds of microseconds to some milliseconds (ms) [7] .

3.2 Formalization of the joint Problem

The joint problem combining buffer management and query scheduling is formalized as follows in the context of coupling SSD and HDD: given a $\mathcal{RDW}$ including a fact table and a set of dimension tables, a query workload $\mathcal{Q} = \{Q_1, Q_2, ..., Q_m\}$, a limited buffer size including the $\mathcal{RDW}$ buffer pool and the SSD. The problem aims at providing: (i) a scheduled set of queries and (ii) a buffer management strategy, minimizing the overall processing cost of $\mathcal{Q}$.

In order to identify the candidates for buffering, we propose to use the multi-query optimization introduced by Sellis [13].

3.3 Multi-query Optimization

Note that each query Q_i of a given workload can be represented by its algebraic tree. Due to the strong interaction between queries, their plans may be merged to generate a graph structure called, *global query plan* (GQP). The leaf nodes of the GQP represent the tables of the $\mathcal{RDW}$. The root nodes represent the final query results and the intermediate nodes represent the common sub-expressions shared by the queries. Among N intermediate nodes , we distinguish: (i) unary nodes representing selection and projection operations and (ii) binary nodes representing join, union, intersection, etc.

Example 1. To illustrate the construction of our GQP, we consider an example a $\mathcal{RDW}$ representing the star schema benchmark (SSB)[5] (Figure 1). On the top of this schema, let us assume that the five following queries are executed:

```
SELECT sum(lo_revenue),d_year FROM lineorder,dwdate,part,supplier
  WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey AND lo_suppkey = s_suppkey
  AND p_brand1 = 'MFGR#2221' AND s_region = 'ASIA'
  GROUP BY d_year ORDER BY d_year;

SELECT sum(lo_revenue) FROM lineorder, part
  WHERE lo_partkey = p_partkey AND p_brand1 = 'MFGR#2221';

SELECT avg(lo_revenue),d_year FROM lineorder,dwdate,part,supplier
  WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey AND lo_suppkey = s_suppkey
  AND p_brand11 = 'MFGR#2221' AND s_region = 'ASIA'
  GROUP BY d_year ORDER BY d_year;

SELECT sum(lo_revenue),d_year FROM lineorder,dwdate,part,supplier
  WHERE lo_orderdate = d_datekey
  AND lo_partkey = p_partkey AND lo_suppkey = s_suppkey
  AND p_brand1 = 'MFGR#2221' AND s_region = 'EUROPE'
  GROUP BY d_year, p_brand1 ORDER BY d_year, p_brand1;

SELECT sum(lo_revenue) FROM lineorder,part,supplier
  WHERE lo_partkey = p_partkey AND lo_suppkey = s_suppkey
  AND p_brand1 = 'MFGR#2221' AND s_region = 'EUROPE';
```

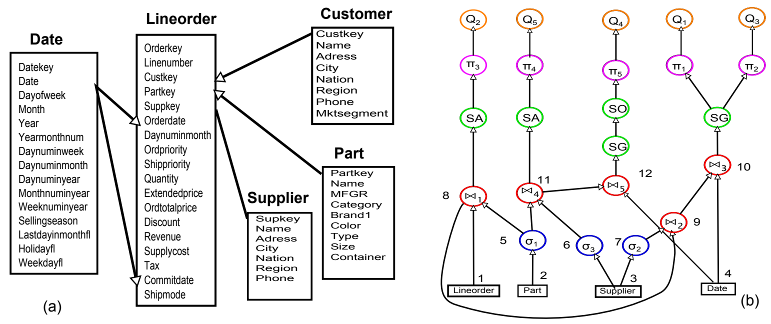


Fig. 1. SSB logical schema and MVPP instance

Figure 1(b) represents an example of GQP. All intermediate nodes of this graph are candidate for buffering. As a consequence, the GQP is the data structure that will be used to define our algorithm for solving the joint problem.

4 $\mathcal{BMQS}$ Problem Resolution

In this section we present in detail our proposal that consists in solving the joint problem of buffer management and query scheduling. We considering a cohabitation of the $\mathcal{RDW}$ buffer pool and the SSD forming a single pool in which the most important intermediate results of queries are candidate to be

[5] http://www.cs.umb.edu/~poneil/StarSchemaB.PDF

cached. The aim of this contribution is to avoid costly hard disk accesses. Figure 2 shows the global architecture of our methodology. Three main objectives have to be fixed to resolve our problem under this cohabitation: (i) the selection of candidate for buffering, (ii) a strategy to manage storage supports: RAM, SDD and HDD in terms of buffering and freeing and (iii) a cost model quantifying the quality of our results.

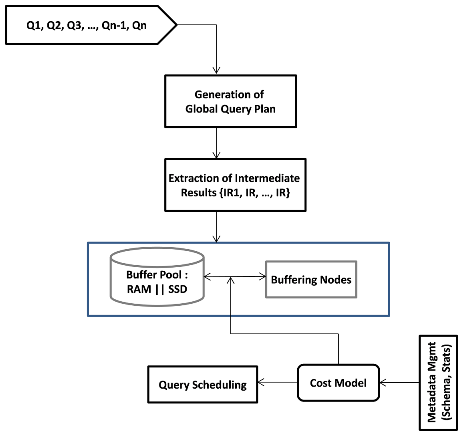

Q1, Q2, Q3, ..., Qn-1, Qn
Generation of Global Query Plan
Extraction of Intermediate Results {IR1, IR, ..., IR}
Buffer Pool : RAM || SSD
Buffering Nodes
Query Scheduling
Cost Model
Metadata Mgmt (Schema, Stats)

Fig. 2. Global Buffering Architecture

4.1 Objective 1: Candidate Selection

Regarding the workload; we use our global query plan (GQP) as a data structure to pick up its parameters. From this graph, we can easily construct the intermediate node usage matrix $\mathcal{INUM}$ which models the "usage" of nodes by queries in $\mathcal{Q}$. To this end, $\mathcal{INUM}$ rows model queries, whereas $\mathcal{INUM}$ columns model intermediate nodes. $\mathcal{INUM}[l][i] = 1$, with $1 \leq l \leq n$ and $1 \leq i \leq N$, if the intermediate node In_i is involved by the query Q_l in $\mathcal{Q}$, otherwise $\mathcal{INUM}[l][i] = 0$. Each intermediate node has a size and can be easily estimated by considering $\mathcal{RDW}$ parameters and selectively factors of joins and selections.

4.2 Objective 2: Strategy for Managing Cohabilitation

For the memory usage and buffers management policies we consider the following assumption: buffer manager fetches intermediate results from the RAM buffer and then from the SSD buffer.

4.3 Objective 3: The Cost Model

A cost model is a central component in database systems. It is used by the query optimizer to evaluate the cost of a query execution plan. It is also used by algorithms selecting optimization structures such as materialized views, indexing, etc. [16]. Several cost models have been proposed in the literature to estimate the query processing cost in terms of the number of inputs/outputs in the traditional storage [9]. Other cost models are also developed in the context of SSD [17]. A few cost models have been elaborated in mixed contexts. A realistic cost model needs parameters issued from four components: (1) the $\mathcal{RDW}$ including the size of different tables, length of instance of a given table, etc., the number of a distinct values of all attributes, etc., (2) the workload, (3) the memory usage and buffers management policies and (4) the hardware device: read latency (L_{Read}), write latency (L_{Write}) , read bandwidth (B_{Read}) and write bandwidth of storage device (B_{Write}). We call $T_{read}(Ir,SD)$ and $T_{write}(Ir,SD)$ the loading and writing time of all pages of a node Ir in the storage device SD [1].

$$T_{read}(Ir,SD) = L_{read} + \frac{|Ir| \cdot page_{size}}{B_{read}} \tag{1}$$

$$T_{write}(Ir,SD) = L_{write} + \frac{|Ir| \cdot page_{size}}{B_{write}} \tag{2}$$

Now, we have all ingredients to describe our buffer management and query scheduling algorithms.

4.4 Buffer Management Algorithm

Our buffer management algorithm aims at improving the workload execution time by avoiding as much hard disk accesses as possible. This is made possible by the prior knowledge of candidate nodes (objective 1) and an accurate cost model (objective 3). The buffer manager is based on two metrics. The first is the frequency i.e. the number of times an intermediate result is used by the workload and the second is the reference used by the Least Frequently Used (LFU) algorithm in case of node eviction.

When a query is treated the Buffer Manager (BM) checks the presence of the concerned node first on the RAM and then on the Flash memory. Once the concerned node is located, its frequency value is decremented. Once the frequency reaches zero value, the node is discarded. When we need to write a node, the Buffer Manager checks for space first on the RAM and then on the Flash memory. In case no space is available neither on the RAM nor on the Flash memory a node is evicted by the Buffer Manager. The will be evicted (victim) node is selected using the LFU algorithm on which the *FindVictimNode* procedure is based. If two victim nodes have the same reference value, the Buffer Manager discards the one on the RAM buffer to free more space and improve access time.

Algorithm 1 illustrates the above described BM treatment.

Example 2. In this example we use the same workload as in example 1. The most important nodes are labeled as 1, 2, ..., 12. For instance full access to table lineorder, part, supplier and date are labeled respectively as 1, 2, 3 and 4. Each

query is represented as a list of nodes:
Q1= 1, 2, 5, 8, 3, 7, 9, 4, 10; Q2=1, 2, 5, 8; Q3=1, 2, 5, 8, 3, 7, 9, 4, 10; Q4=2, 5, 3, 6, 11, 4, 12; Q5=2, 5, 3, 6, 11.

The execution of the five queries in the same order can be viewed as a reference string composed by the concatenation of all queries nodes: 1, 2, 5, 8, 3, 7, 9, 4, 10, 1, 2, 5, 8, 1, 2, 5, 8, 3, 7, 9, 4, 10, 2, 5, 3, 6, 11, 4, 12, 2, 5, 3, 6, 11.

Without loss of generality, we suppose that nodes have the same size, the RAM buffer can hold three nodes while the flash buffer can hold six.

The following table depicts the reference count and the frequency of each node in the workload

Nodes#	1	2	3	4	5	6	7	8	9	10	11	12
Frequency	3	5	5	3	5	2	2	3	2	2	2	1

The LFU algorithm proceeds as follows :

- replace the node which has been referenced least often,
- for each page in the reference string, we need to keep a reference count. All reference counts start at 0 and are incremented every time a page is referenced.

So, we get RAM with 3 nodes and flash with 6 nodes. After executing Q1 we obtain this result:

Buffer	RAM			FLASH								
Q1	1	2	5	8	3	7	9	4	10	6	11	12
Frequency	2	4	4	2	4	1	1	2	1	2	2	1
Reference	1	1	1	1	1	1	1	1	1	0	0	0

Q1=1, 2, 5, 8, 3, 7, 9, 4, 10. All nodes are put in RAM and FLASH buffers.

Now, we execute Q2. The result show that all nodes are already in buffers. Just updates reference for LFU and Frequency.

Buffer	RAM			FLASH								
Q2	1	2	5	8	3	7	9	4	10	6	11	12
Frequency	1	3	3	2	4	1	1	2	1	2	2	1
Reference	2	2	2	1	1	1	1	1	1	0	0	0

When executing Q3, nodes with frequency=0 are freed.

Buffer	RAM			FLASH								
Q3		2	5	8	3			4	10	6	11	12
Frequency	0	2	2	1	3	0	0	1	0	2	2	1
Reference	3	3	3	2	2	2	2	2	2	0	0	0

When executing Q4, the nodes 6, 11 and 12 are buffered and their reference and frequency are updated.

Buffer	RAM			FLASH								
Q4	6	2	5	8	3	11	12	4	10	6		
Frequency	1	2	2	1	3	1	0	1	0	2		
Reference	1	3	3	2	2	1	1	2	2	0		

After executing Q5 nothing happens since all nodes are already buffered.

Algorithm 1: Read and write operation of the buffer manager

```
Input: Intermediate node $n$
Output: Read/write the result data of $n$
if operation = read then
    if $n$ reside in RAMBuffer then
        Return the result data of $n$ from RAM buffer;
        frequency($n$) = frequency($n$) - 1; //decrement the frequency of $n$
    else
        if $n$ reside in FlashBuffer then
            Return the result data of $n$ from Flash buffer;
            frequency($n$) = frequency($n$) - 1; //decrement the frequency of $n$
        end
    end
    if frequency(n) = 0 then
        release($n$); // release buffer space occupied by $n$
    end
end
//else write operation
if operation = write then
    if RAMBuffer has available space then
        Put the result data of $n$ in RAM buffer;
    else
        if FlashBuffer has available space then
            Put the result data of $n$ in Flash buffer;
        else
            FindVictimNode(n);
        end
    end
end
```

4.5 Heuristic Query Scheduler

In this section, we propose a query scheduling algorithm based on divide and conquer approach to reduce the complexity of the scheduling problem. To do so, we propose to partition queries into fragments, where each one contains a subset of queries. This generation is obtained using the following approach. From the Intermediate Intermediate Node Usage Matrix (Section 4.1), we propose to generate the *Query Affinity Matrix* $\mathcal{QAM}$, which models the "affinity" between two queries Q_i and Q_j. To this end, $\mathcal{QAM}$ rows and columns both model queries, hence $\mathcal{QAM}$ is a symmetric matrix. $\mathcal{QAM}[i][j]$, with $1 \leq i \leq n$ and $1 \leq j \leq n$, represents the number of the shared intermediate results involving the queries Q_i and Q_j. Table 1 shows an example matrix involving ten queries. We use the graphical approach proposed in [11] to partition the affinity graph represented by the affinity matrix. This algorithm constructs a complete weighted graph, where each edge represents the affinity measure. From this graph, a set of cycles are generated, where each cycle contains the queries having similar affinities. As final, each cycle represents a query fragment.

Table 1. Query Affinity Matrix

Queries	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10
q1	3	0	0	0	4	0	3	0	0	0
q2	0	2	0	4	0	4	0	0	5	0
q3	0	0	2	0	0	0	0	4	0	5
q4	0	4	0	3	0	3	0	0	3	0
q5	4	0	0	0	3	0	3	0	0	0
q6	0	4	0	3	0	3	0	0	4	0
q7	3	0	0	0	3	0	2	0	0	0
q8	0	0	4	0	0	0	0	3	0	4
q9	0	5	0	3	0	4	0	0	2	0
q10	0	0	5	0	0	0	0	4	0	2

The partitioning algorithm applied in this matrix generates three partitions: $P_1 : [q_1, q_5, q_7]$, $P_2 : [q_2, q_4, q_6, q_9]$ and $P_3 : [q_3, q_8, q_{10}]$. These partitions are then used to schedule the queries globally and locally. At the global level, we propose to order partitions based on their cardinalities in descending order. We begin with dense partitions to buffer intermediate results that are shared with a maximum number of queries. At the local level, we order queries of each partition according to their execution costs in ascending order.

The global ordering performed in the above partitions is: $P_1 \longrightarrow P_2 \longrightarrow P_3$. Locally each partition is then ordered as follows: $P_2 : [q2, q4, q6, q9], P_1 : [q_1, q_5, q_7]$ and $P_3 : [q_3, q_8, q_{10}]$. The intra partition sort gives the following final scheduling: [q6 $\rightarrow$ q4 $\rightarrow$ q2 $\rightarrow$ q9 $\rightarrow$ q5 $\rightarrow$ q1 $\rightarrow$ q7$\rightarrow$ q10 $\rightarrow$ q3 $\rightarrow$ q8].

5 Experimental Study

To show the effectiveness of our approach, a Java simulator is developed. In the following subsections, we will present experiments used to test our algorithms. The characteristics of the two simulated storage devices are presented in table 2.

Table 2. Example of HDD and SSD specifications

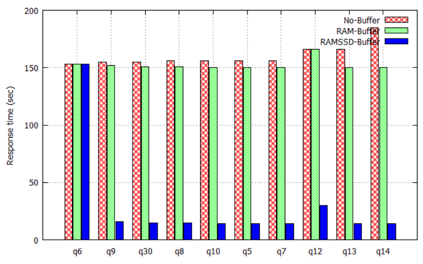
feature	HDD Barracuda 7200.10	SSD Intel DC S3700
Page size	8 KB	8 KB
Latency Read Sequential	16 ms	50 μs
Read Sequential IOPS	8400 IOPS	–
Latency Read Random	555 ms	–
Read Random IOPS	232 IOPS	47.500 IOPS
Latency Write Sequential	17 ms	65 μs
Write Sequential IOPS	7660 IOPS	–
Latency Write Random	674 ms	–
Write Random IOPS	191 IOPS	20.000 IOPS
Size	250 GB	100/200/400/800 GB

To evaluate our simulator capability we use the dataset of the Star Schema Benchmark (SSB) with a scale factor (SF=10), which generates 10 GB of data. The workload contains 30 star join queries defined on the schema of SSB. Several tests with different software and hardware configurations have been conducted on the SSB data warehouse. Graphics (a-c) shown in figure 3 are simulation results obtained with RAM size of 4GB and flash memory of 8GB. Graphics (d), (e) and (f) depict the whole workload answer times with respectively (4GB, 8GB), (1GB, 3GB) and (1GB, 8GB) of (RAM, FLASH) sizes. Tests are performed on three cases: (1) No cache (data on HDD) representing initial cost, (2) RAM buffer, (3) RAM buffer extended with flash.

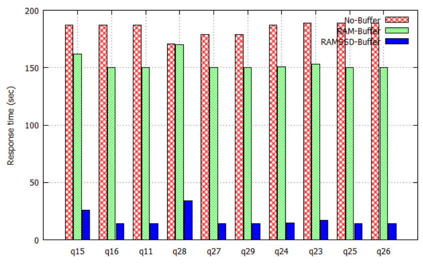
The simulation results are encouraging and show the feasibility our approach. Significant improvements have been made in queries execution time, with up to 86% in the case 8GB Flash. The worst case, where our approaches do not give a good result (meaning 0% of enhancement) is when all intermediates nodes sizes are greater than both the RAM buffer and the flash buffer.
With a scale factor SF=10, the database size is 10GB and we need 7GB of storage space to materialize intermediates nodes. Intermediates nodes sizes are very important, which explain the fact that with a RAM=1GB and FLASH=3GB we have not achieved any improvement. In the case of RAM=1GB and FLASH=8GB the same results are obtained as the first case (RAM=4GB and FLASH=8GB, but the last configuration has the best economic cost. Also, as depicted in figure 3(a), the query Q6 of our workload is the first query submitted to the execution engine. So no intermediate results are cached yet and hence it does not benefit from caching in all cases. Finally, the results show the high efficiency of the RAM-SSD buffer compared with no buffer and only RAM buffer.

6 Conclusion

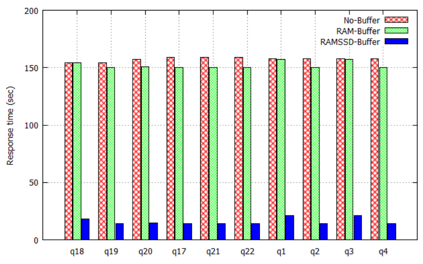
In this paper, we have discussed the issue of extending traditional buffer storage with flash memories. Under this situation a joint problem including buffer



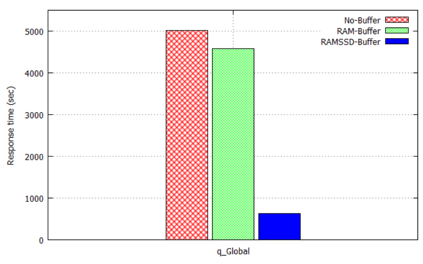
(a) First Cluster



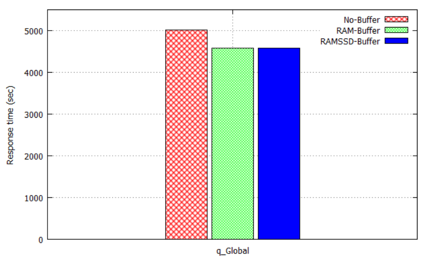
(b) second Cluster



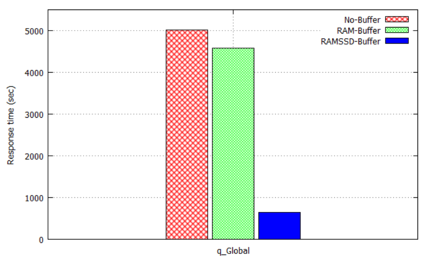
(c) Third Cluster



(d) Workload time with RAM=4GB and FLASH=8GB



(e) Workload time with RAM=1GB and FLASH=3GB



(f) Workload time with RAM=1GB and FLASH=8GB

Fig. 3. Effect of SSD Buffering on query execution time

management and query scheduling is studied. To deal with this problem under this storage architecture, we have fixed three main objectives: (i) the identification of candidates for buffering, (ii) the proposition of a strategy to ensure the co-habilitation for buffering and freeing, and (iii) the presence of a cost model quantifying the quality of the proposed solution. For the first objective, we have used the concept of multiquery optimization to identify the candidate intermediate nodes for buffering. Note that in the context of relational data warehouses, the number of nodes can be very high. For the second objective, we favor the traditional buffer for caching and freeing. Finally, a cost model estimating the query processing cost in terms of inputs and outputs is given taking into account the traditional buffer and flash memory parameters. A simulator is developed to test our findings. The obtained results are encouraging and show the interest of our co-habilitation.

Testing our proposal in a real environment is an ongoing work and prospect from improvement.

References

1. L. Bellatreche, S. Cheikh, S. Breß, A. Kerkad, A. Boukhorca, and J. Boukhobza. How to exploit the device diversity and database interaction to propose a generic cost model? In *IDEAS*, pages 142–147, 2013.
2. J. Boukhobza. *Data Intensive Storage Services for Cloud Environments*, chapter Flashing in the Cloud: Shedding Some Light on NAND Flash Memory Storage Systems. 2013.
3. H-T Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB*, pages 127–141, 1985.
4. D. W. Cornell and P. S. Yu. Integration of buffer management and query optimization in relational database environment. In *VLDB*, pages 247–255, 1989.
5. David J. DeWitt, Jaeyoung Do, Jignesh M. Patel, and Donghui Zhang. Fast peak-to-peak behavior with ssd buffer pool. In *ICDE*, pages 1129–1140, 2013.
6. J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging dbms buffer pool using ssds. In *SIGMOD*, pages 1113–1124, 2011.
7. L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of nand flash memory. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 2–2, 2012.
8. Z. He and P. Veeraraghavan. Fine-grained updates in database management systems for flash memory. *Inf. Sci.*, 179(18):3162–3181, 2009.
9. A. Kerkad, L. Bellatreche, and D. Geniet. Queen-bee: Query interaction-aware for buffer allocation and scheduling problem. In *DaWaK*, pages 156–167, 2012.
10. I. Koltsidas and S. Viglas. Designing a flash-aware two-level cache. In *ADBIS*, pages 153–169, 2011.
11. S. B. Navathe and M. Ra. Vertical partitioning for database design: a graphical algorithm. *SIGMOD Rec.*, 18(2):440–450, June 1989.
12. Parthasarathy R. From microprocessors to nanostores: Rethinking data-centric systems. *Computer*, 44(1):39–48, 2011.
13. Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.
14. D. Thomas, A. A. Diwan, and S. Sudarshan. Scheduling and caching in multiquery optimization. In *COMAD*, pages 150–153, 2006.
15. E. Wolfgang and H. Theo. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, December 1984.
16. J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.
17. O. Yi and H. Theo. Improving database performance using a flash-based write cache. In *DASFAA*, pages 2–13, 2012.