
# Fourth International Workshop on Data Management on New Hardware (DaMoN 2008)

June 13, 2008

Vancouver, Canada

In conjunction with ACM SIGMOD/PODS Conference

Qiong Luo and Kenneth A. Ross
(Editors)

Industrial Sponsors



# FOREWARD

## Objective

The aim of this one-day workshop is to bring together researchers who are interested in optimizing database performance on modern computing infrastructure by designing new data management techniques and tools.

## Topics of Interest

The continued evolution of computing hardware and infrastructure imposes new challenges and bottlenecks to program performance. As a result, traditional database architectures that focus solely on I/O optimization increasingly fail to utilize hardware resources efficiently. CPUs with superscalar out-of-order execution, simultaneous multi-threading, multi-level memory hierarchies, and future storage hardware (such as flash drives) impose a great challenge to optimizing database performance. Consequently, exploiting the characteristics of modern hardware has become an important topic of database systems research.

The goal is to make database systems adapt automatically to the sophisticated hardware characteristics, thus maximizing performance transparently to applications. To achieve this goal, the data management community needs interdisciplinary collaboration with computer architecture, compiler and operating systems researchers. This involves rethinking traditional data structures, query processing algorithms, and database software architectures to adapt to the advances in the underlying hardware infrastructure.

## Workshop Co-Chairs

Qiong Luo (Hong Kong University of Science and Technology, luo@cse.ust.hk)
Kenneth A. Ross (Columbia University, kar@cs.columbia.edu)

## Program Committee

| | |
|---|---|
| Anastasia Ailamaki | (Carnegie Mellon University) |
| Bishwaranjan Bhattacharjee | (IBM Research) |
| Peter Boncz | (CWI Amsterdam) |
| Shimin Chen | (Intel Research) |
| Goetz Graefe | (HP Labs) |
| Stavros Harizopoulos | (HP Labs) |
| Martin Kersten | (CWI Amsterdam) |
| Bongki Moon | (University of Arizona) |
| Jun Rao | (IBM Research) |
| Jingren Zhou | (Microsoft Research) |

TABLE OF CONTENTS

**Session 1: Query Processing on Novel Storage**

***CAM Conscious Integrated Answering of Frequent Elements and Top-k Queries over Data Streams*** ..... 1
Sudipto Das (University of California, Santa Barbara)
Divyakant Agrawal (University of California, Santa Barbara)
Amr El Abbadi (University of California, Santa Barbara)

***Modeling the Performance of Algorithms on Flash Memory Devices*** ..... 11
Kenneth A. Ross (IBM T. J. Watson Research Center and Columbia University)

***Fast Scans and Joins using Flash Drives*** ..... 17
Mehul A. Shah (HP Labs)
Stavros Harizopoulos (HP Labs)
Janet L. Wiener (HP Labs)
Goetz Graefe (HP Labs)

**Session 2: Parallelism and Contention**

***Data Partitioning on Chip Multiprocessors*** ..... 25
John Cieslewicz (Columbia University)
Kenneth A. Ross (Columbia University)

***Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines*** ..... 35
Ryan Johnson (Carnegie Mellon University)
Ippokratis Pandis (Carnegie Mellon University)
Anastasia Ailamaki (Ecole Polytechnique Federale de Lausanne)

**Session 3: Page Layout**

***Avoiding Version Redundancy for High Performance Reads in Temporal DataBases*** ..... 41
Khaled Jouini (Universite Paris Dauphine)
Geneviève Jomier (Universite Paris Dauphine)

***DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing*** ..... 47
Marcin Zukowski (CWI Amsterdam)
Niels Nes (CWI Amsterdam)
Peter Boncz (CWI Amsterdam)

# CAM Conscious Integrated Answering of Frequent Elements and Top-k Queries over Data Streams*

Sudipto Das Divyakant Agrawal Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
{sudipto, agrawal, amr}@cs.ucsb.edu

## ABSTRACT

Frequent elements and top-$k$ queries constitute an important class of queries for data stream analysis applications. Certain applications require answers for both frequent elements and top-$k$ queries on the same stream. In addition, the ever increasing data rates call for providing fast answers to the queries, and researchers have been looking towards exploiting specialized hardware for this purpose. Content Addressable Memory(CAM) provides an efficient way of looking up elements and hence are well suited for the class of algorithms that involve lookups. In this paper, we present a fast and efficient CAM conscious integrated solution for answering both frequent elements and top-$k$ queries on the same stream. We call our scheme *CAM conscious Space Saving with Stream Summary (CSSwSS)*, and it can efficiently answer continuous queries. We provide an implementation of the proposed scheme using commodity CAM chips, and the experimental evaluation demonstrates that not only does the proposed scheme outperforms existing CAM conscious techniques by an order of magnitude at query loads of about 10%, but the proposed scheme can also efficiently answer continuous queries.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous.

## General Terms

Stream Algorithms, Design, Performance.

## Keywords

Data Streams, Frequent elements queries, Top-$k$ queries, Content Addressable Memory, Network Processor.

*This work is partly supported by NSF Grants IIS-0744539 and CNS-0423336


*Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008),* June 13, 2008, Vancouver Canada


## 1. INTRODUCTION

Data stream applications, such as click stream analysis for fraud detection and network traffic monitoring, have gained in popularity over the last few years. Common queries posed by the users include frequent elements [16, 4, 7, 19, 17], top-$k$ queries [6, 18, 17], quantile summarization [11], heavy distinct hitters [22] and many more. The frequent elements query looks for elements whose frequency is above a certain threshold. For example, a network administrator interested in finding the IP addresses that are contributing to more than 0.1% of the network traffic will issue a frequent elements query. On the other hand, the 100 most popular search terms in a stream of queries constitute a top-$k$ query. Quantile queries are used for stream summarization such as percentiles and medians, whereas a heavy distinct hitter query is used for detecting malicious activities such as spreading of worms in the network.

The frequent elements and top-$k$ queries are used by different analysis applications such as network and web traffic monitoring, click stream analysis, financial monitoring and so on. Besides applications where the users are either interested in frequent elements *or* top-$k$ elements, there are certain applications where the user is interested in both frequent elements *and* top-$k$ elements on the same stream of tuples. As an example, if we consider the case of a search engine, in order to optimize the performance of the engine, the designer might decide to cache the answers to the 1000 most popular queries. This requires a top-$k$ query on the stream of query terms. On the other hand, for auctioning the search keywords, the designer will be interested in the queries which are above a certain threshold, and accordingly assign bidding price to these keywords. In this case a frequent elements query is used for some support threshold of 0.5%.

Even though frequent elements and top-$k$ queries seem to be very similar, there is a fundamental difference. In frequent elements computation, there is a notion of the minimum possible frequency of an element but no ordering information is necessary, whereas in answering top-$k$ queries, the exact frequencies of the elements might not be of interest as long as an ordering of the elements is known. As a result, answering frequent elements cannot be used as a pre-processing step for top-$k$ queries, and

vice versa. A specialized solution is therefore sought for the applications that need frequent elements and top-$k$ queries on the same stream of elements. Metwally et. al. [17] suggest an efficient integrated solution, known as *Space Saving*, for answering both frequent elements and top-$k$ queries. The authors propose a counter based technique for frequency counting, and a data structure to order the elements by their frequencies so that top-$k$ queries can be easily answered.

Besides the need for having an integrated solution for frequent elements and top-$k$ queries, the ever increasing data stream rates call for fast and efficient processing. In addition, new hardware paradigms have opened up new frontiers for efficient data management solutions leveraging specialized hardware features [5, 10, 8, 12, 3, 25, 23, 2]. A Content Addressable Memory (CAM) can also be exploited for accelerating stream processing. An interesting property of CAM is that in addition to normal read and write operations, it supports constant time lookup operation in hardware. A CAM obviates the need for complex data structures, such as Hashtables or search trees, to process efficient lookups. Hence, a CAM can be efficiently used by algorithms that perform frequent lookups or searches. Bandi et al. [1] propose a CAM conscious adaptation for the *Space Saving* algorithm and demonstrate acceleration compared to a software implementation. A disadvantage of the proposed adaptation is that the elements are not sorted by their frequencies. As a result, the algorithm cannot answer top-$k$ queries efficiently. Additionally, since the elements are not sorted, adapting the approach in [1] for continuous queries, or even moderately high query loads, is not straightforward. In this paper, we propose a CAM conscious version of the *Space Saving* algorithm that also maintains the ordering of the elements and hence can answer both frequent elements and top-$k$ queries on the same stream. Our scheme, *CAM conscious Space Saving with Stream Summary (CSSwSS)*, can efficiently answer continuous queries. The major contributions of this paper are summarized as follows:

- This is the first approach of using CAM for providing an "integrated" solution to frequent elements and top-$k$ queries.
- We propose a CAM based data structure to count occurrences of the elements and efficiently maintain the sorted order of the elements in terms of their frequency. We explore the possible design alternatives and analyze their advantages and disadvantages.
- We provide an implementation of the proposed algorithm using a commodity CAM chip, and report the performance of this algorithm on an experimental prototype using synthetic data sets.

The rest of the paper is organized as follows: Section 2 summarizes the work that has been carried in frequent elements and top-$k$ computation, and in using specialized hardware for data management operations, Section 3 explains the hardware prototype used for our implementation, Section 4 explains in detail our proposed algorithm, Section 5 provides a thorough experimental evaluation and analysis of the results and Section 6 concludes the paper.

## 2. RELATED WORK

Frequent elements and top-$k$ queries are among the most common queries in data stream processing applications, and a large number of approaches have been suggested to answer these queries. The algorithms for answering frequent elements queries are broadly divided into two categories: *sketch based* and *counter based*. The *sketch based* techniques such as the one proposed by Charikar et al. [4] try to represent the entire stream's information as a "sketch" which is updated as the elements are processed. Since the "sketch" does not store per element information, the error bounds of these techniques are not very stringent. In addition, these techniques generally process each stream element using a series of hash functions, and hence the processing cost per element is also high. These approaches are therefore not suitable for providing fast answers to queries.

On the other hand, the *counter based* techniques such as [17] monitor a subset of the stream elements and maintain an approximate frequency count of the elements. Different approaches use different heuristics to determine the set of elements to be monitored. For example, Manku et al. [16] propose a technique called *Lossy Counting* in which the stream is divided into rounds, and at the end of every round, potentially non-frequent elements are deleted. This $\epsilon$-approximate algorithm has a space bound of $O(\frac{1}{\epsilon}\log(\epsilon N))$, where $N$ is the length of the stream. Panigrahy et al. [19] suggest a sampling based counting technique which monitors a subset of elements and manipulates the counters based on whether a sampled element is already being monitored or not. For a bursty stream this approach has a space bound of $O(\frac{F_2}{t})$, where $F_2$ is the $2^{nd}$ frequency moment and $t$ is the minimum frequency to be reported. Most of these *counter based* algorithms [16, 17, 19] are a generalization of the classic *Majority* algorithm [9], and the goal is to minimize the space as well as reduce the error in approximation.

Different solutions have also been suggested for answering top-$k$ queries. Mouratidis et al. [18] suggest the use of geometrical properties to determine the $k$-skyband and use this abstraction to answer top-$k$ queries, whereas Das et al. [6] propose a technique which is capable of answering ad-hoc top-$k$ queries, i.e., the algorithm does not need *apriori* knowledge of the attribute on which the top-$k$ queries have to be answered.

With the growing data rates and faster processing speed requirements, researchers are also striving for accelerating these queries. Bandi et al. [1] suggested the use of Content Addressable Memories (CAM) for accelerating the frequent elements queries. The authors leverage the constant time lookups of CAM to accelerate a couple of counter based techniques. As pointed out earlier, this approach cannot efficiently answer continuous top-$k$ and frequent elements queries. Bandi et al. [2] also proposed the use of CAM for Database operations. Other approaches for data management on new hard-

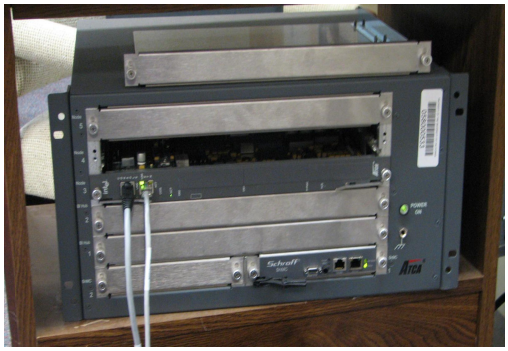


**Figure 1: Hardware prototype.**

ware paradigms have also been proposed. For example, Gold et al. [10] leveraged Network Processors for accelerating database operators, while Cieslewicz et al. [5] propose the use of Chip Multiprocessors for answering aggregation queries. The use of Graphics processors has also been proposed by Fang et al. [8]. For this work, we concentrate on using CAM for efficient answering of frequent elements and top-$k$ queries.

# 3. HARDWARE PROTOTYPE

The hardware prototype (Figure 1) used for our implementation consists of two major constituents: the Network Processing Unit (NPU) and the Ternary Content Addressable Memory (TCAM). In this section, we explain the features of the two constituent parts.

## 3.1 NPU Architecture

For the implementation in this paper, we use the Intel IXP2800 network processor [15]. This network processor consists of a Control Plane Processor (CPP) and 16 independently operating data plane processors referred to as Micro Engines (ME). The CPP is a 32-bit XScale core that runs Monta Vista linux, operates at a maximum clock speed of 700MHz, and is designed to work as a "master" assigning tasks to the ME. On the other hand, the ME is designed to perform simple data plane operations very fast. The MEs have a very simple design and instruction set and have been optimized to quickly perform simple operations. Each ME operates at a maximum clock speed of 1.4GHz and has 8 hardware thread contexts. Therefore, the NPU provides $128(16 \times 8)$ hardware threads. A hardware thread is different from a software thread since each thread has its own set of registers and hence switching contexts between threads incurs minimal overhead. Each ME has a set of general purpose registers and a small instruction cache.

In addition to the Thread Level Parallelism (TLP), the NPU also provides a form of pipeline parallelism. The MEs are arranged in a pipelined fashion and an ME shares a set of registers with its immediate next ME. These registers are called nearest neighbor registers and are designed for fast inter-communication between the MEs. For our implementation we use only a single ME and do not use either of the above mentioned forms of parallelism.

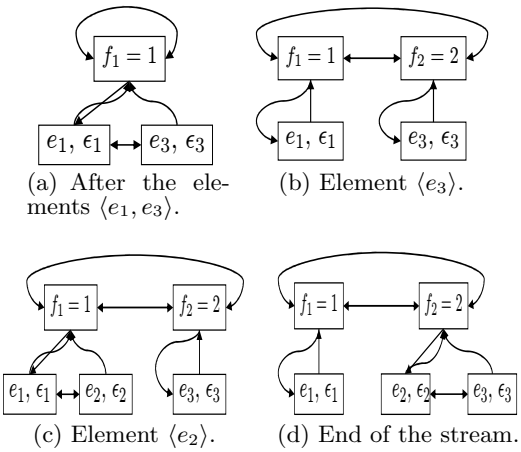
The CPP as well as the 16 MEs share some common on-chip resources like the PCI unit, hashing unit, the on-chip shared memory known as scratchpad, as well as the industry standard DRAM and SRAM interfaces. Main memory is available in the form of DRAM and SRAM and is present on-board. A TCAM chip can be efficiently interfaced with the NPU through the SRAM interface.

Some important features of this architecture are as follows: *First*, even though the NPU provides a lot of parallelism, the architecture is so simple that only a small set of instructions are supported and is suited for simple operations. For example, floating point operations are not supported by the ME. So the applications for which the NPU can be used are very limited. *Second*, the CPP, which acts as a master, runs at a speed slower than the ME. Therefore, when running parallel threads, the master should also perform simple tasks or else the master might become a bottleneck. *Finally*, the MEs do not have any support for a memory hierarchy. This is both an advantage as well as a disadvantage. The programmers have the freedom to decide precisely where their data resides, but this design increases the overhead for the application design.

## 3.2 TCAM Architecture

Content addressable memories (CAM) provide efficient lookup operations in hardware and have been typically used for networking applications such as IP address lookups for packet forwarding or implementation of access control lists. For our implementation, we use the IDT 75K62134 chip [13] which is a Ternary Content Addressable Memory (TCAM). In addition to the properties of a CAM, a TCAM has ternary capabilities (i.e. it can represent a "don't-care" state and hence can efficiently represent ranges) and are therefore suited for longest prefix matching in IP forwarding. For example, a TCAM can store an IP range as say 168.111.$*$, such that any IP address in the range 168.111.0.0 to 168.111.255.255 will match this entry. A typical TCAM chip, similar to the one used in this paper, consists of either 36-bit or 72-bit word-arrays and typical sizes are $128K$ or $256K$ of these entries. The proposed algorithms do not use the ternary capabilities, so a TCAM and a CAM become functionally equivalent.

A TCAM consists of a two dimensional array of bits, and the lookup is performed by a parallel comparison of the search key with all the stored words in a SIMD-like fashion. If the search key is present in the TCAM core, then a match is reported and this is referred to as a *hit*. If multiple copies of the search key are found, then the smallest index is reported by a priority encoder, and a special *multi-hit* bit is set. Since all the elements in the TCAM array are compared in parallel, the power consumption of the TCAM is pretty high. To lower the power consumption, the chip we use allows division of the TCAM into a number of search databases, so that the application designer can selectively power down the databases that are not in use and ignore them for comparison. The TCAM chip also supports *mask registers*, which can be used to selectively mask certain bits dur-



(a) After the elements $\langle e_1, e_3 \rangle$. (b) Element $\langle e_3 \rangle$. (c) Element $\langle e_2 \rangle$. (d) End of the stream.

**Figure 2: This figure illustrates the *Stream Summary* data structure for an example stream of elements $\langle\ e_1, e_3, e_3, e_2, e_2\ \rangle$.**

ing the `LOOKUP` and `WRITE` operations. The bits that are masked do not participate in comparison during the `LOOKUP` operations, and are not affected by the `WRITE` operation. In addition to these operations, a TCAM also supports reads and writes similar to conventional memories.

Some important characteristics of the TCAM are as follows: *First*, the chip can support a lookup throughput of 100 million lookups per second, but that is possible only through parallel access to the TCAM and pipelining of requests. The chip under consideration can support up to 128 pending requests. For our implementation, we do not consider the parallelism supported by the TCAM. *Second*, even though the TCAM efficiently interfaces with the NPU, the word sizes of two devices do not match. The TCAM core is 72-bit wide, whereas the SRAM bus of the NPU is only 32-bit wide. As a result, all communication between the NPU and the TCAM takes place through multi-word transfers. Due to this overhead, it is very difficult to attain the maximum supported throughput.

# 4. SYSTEM DESIGN

This section explains the proposed algorithm for the hardware prototype described in Section 3.

## 4.1 Background

Metwally et. al. [17] proposed an "integrated" scheme for answering frequent elements and top-$k$ queries. The proposed counter based technique, called *Space Saving*, is based on the intuition that frequent elements are of importance only in a skewed stream, and in a skewed stream, the frequency of the elements of interest is much higher compared to the low frequency ones. This heuristic is used to limit the number of elements which need to be monitored. A nice property of this algorithm is that the number of elements to be monitored is independent of the length of the stream or the size of the alphabet, and depends on only the user specified error bound. In this algorithm, with the arrival of an element in the stream, if the element is already being monitored, then its count is incremented, otherwise the new element replaces the element with the minimum frequency. The space bound for this algorithm is $O(\frac{1}{\epsilon})$, where $\epsilon$ is the user specified error bound. An $\epsilon$-*approximate* algorithm is one such that given a support $\phi$, instead of reporting the elements above the frequency $\phi N$, the algorithm reports the elements whose frequency is above $(\phi - \epsilon)N$.

To answer top-$k$ queries and keep track of the minimum frequency element, this algorithm maintains a data structure, called *Stream Summary* [7, 17], which can efficiently keep the elements sorted by their frequency. The *Stream Summary* data structure consists of "frequency buckets" which correspond to frequency values of at least one element that is being monitored, and each bucket consists of the elements which have the same approximate frequency as is represented by the bucket. Figure 2(a) provides an illustration of the *Stream Summary* structure with two elements with frequency 1. Figure 2 illustrates how the elements in the stream are processed and maintained in sorted order of their frequency. For example, in the example stream, Figure 2(a) shows the structure after elements $\langle e_1, e_3 \rangle$ have been processed. When the element $e_3$ appears again, its frequency is incremented to 2, and Figure 2(b) shows the state of *Stream Summary* structure. The elements are always kept sorted, and the per-element processing cost is a constant [7, 17].

Authors in [1] propose an adaptation of the *Space Saving* algorithm to use the constant time lookup of TCAM. They store the elements and their frequencies in the TCAM. As the stream is being processed, the elements can be looked up in constant time. In addition to that, the *Space Saving* algorithm needs to keep track of the minimum frequency element. This can also be done efficiently by looking up the minimum frequency from the TCAM and always keeping track of the minimum frequency. As mentioned earlier, this TCAM adaptation does not sort the frequencies and hence answering queries is costly. In this paper, we refer to this algorithm as *CAM conscious Space Saving (CSS)*.

## 4.2 Proposed Algorithm

*Stream Summary* is a powerful structure that can be used to efficiently keep the elements sorted by their frequency. This structure can be easily adapted to the CAM setting. With every stream element, the algorithm needs to determine whether the element is already being monitored. Since CAM provides efficient lookups, we can place the elements in the CAM. On the other hand, in order to keep the elements sorted by their frequency, the circular doubly-linked structure of frequency buckets is maintained in SRAM. Figure 3 provides an illustration of the *CAM adapted Stream Summary* data structure. In Algorithm 1, we provide the details of how the *Space Saving* algorithm can be adapted to use the *CAM adapted Stream Summary*. We call our algorithm *CAM conscious Space Saving with Stream Summary (CSSwSS)*. Algorithm 2 gives an overview of the supporting routines used by Algorithm 1.

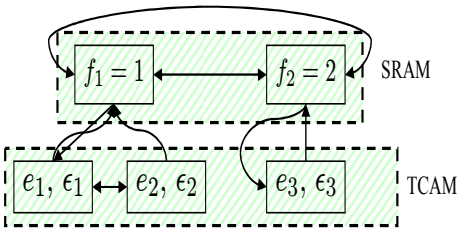



**Figure 3: TCAM adaptation of the stream summary data structure.**

As described in Section 3.2, the TCAM chip we use for our implementation provides support for programmable size of search keys. This is accomplished by having TCAM core of different sizes. An advantage of having a larger TCAM core is that more data can fit into a single TCAM entry; the disadvantage being that each TCAM operation becomes costlier when compared to smaller TCAM cores. Due to this programmable size, two possible implementations of the proposed approach are possible, based on what information is kept in the TCAM and what is kept outside. It must be noted that in addition to storing the elements, its frequency, the error in frequency approximation, and the link structure for the sort order also need to be stored.

**Narrow TCAM Core (72-bit)**: The TCAM chip used in this implementation consists of an array of 72-bit wide entries. As a result, the 72-bit core size is an obvious choice. In this design, as the number of bits in the TCAM core is less than the information to be stored, part of information is stored in the TCAM and the rest of the information is stored in SRAM with a pointer to the corresponding information stored in TCAM. Since the elements need to be looked up, they must be placed in the TCAM. The placement of other items is an implementation choice. In this implementation, we place the element and the link structure of the elements in the TCAM, while the error and a pointer to the frequency bucket is placed in the SRAM. An advantage of this approach is that TCAM space is preserved as we are using the bare minimum when it comes to TCAM resources. The disadvantage is an added level of indirection which is incurred when processing a stream element. Since all information is not stored in the TCAM, a TCAM `LOOKUP` is followed by a SRAM read to obtain all information corresponding to an element.

**Wide TCAM Core (144-bit)**: The chip being used allows programmable word size. So the device may be programmed to have a core size of 144-bits, where two consecutive 72-bit entries are combined. As the TCAM entries are wider in this design, all necessary information corresponding to an element, i.e. the error and pointer to the frequency bucket, fit into a single wide TCAM entry. The advantage of this design is that the additional level of indirection is avoided, the disadvantages being more TCAM space being utilized (which might be undesirable as the number of TCAM entries is limited), and wide core makes each TCAM operation costlier compared to the operations with a narrow core.

**Algorithm 1** *CAM conscious Space Saving with Stream Summary (CSSwSS)*

```
/* mask_1 register masks the element component */
/* LOOKUP, WRITE are TCAM operations. */
/* min_fr is the pointer to the minimum frequency bucket in
Stream Summary. */
Procedure ComputeFrequentTopK(stream, table_size,
min_fr)
for each element ⟨e⟩ in stream do
  hit ← LOOKUP(e, index, mask_1)
  if (hit) then
    /* The element is already being monitored, so increment
    its counter. */
    cur_fr ← FrequencyBucketAtIndex(index)
    added_bucket ← IncrementCounter(cur_fr, index)
    added_elem ← index
  else
    /* The Element is not being monitored, so either add
    to the end of the list if there is space, or overwrite the
    minimum. */
    if (cur_size < table_size) then
      /* Space is left, so add element */
      WRITE(free_index, e, 0, ⟨link_structure⟩)
      if (min_fr = NULL || min_fr→freq > 1) then
        /* A new Frequency Bucket must be added to the
        list. */
        new_node ← AllocateNewNode()
        new_node→freq ← 1
        AddElementToList(new_node, free_index)
        added_bucket ← new_node
        added_elem ← free_index
      else
        /* A bucket with frequency 1 already exists, add
        this element to that bucket. */
        AddElementToList(min_fr, free_index)
        added_bucket ← min_fr
        added_elem ← free_index
      end if
      free_index++
    else
      /* Overwrite the minimum element. */
      index ← min_fr→elem
      WRITE(index, e, min_fr→freq, ⟨link_structure⟩)
      added_bucket ← IncrementCounter(min_fr, index)
      added_elem ← index
    end if
  end if
end for
end Procedure ComputeFrequentTopK
```

## 4.3 Query Processing

The *CAM adapted Stream Summary* data structure can be used to efficiently answer the frequent elements and top-$k$ queries. In this paper we consider two different types of queries, viz. *Continuous Queries* and *Interval Queries*. *Continuous Queries* are posed with every update, i.e. with every stream element processed, and the answer cache is always up-to-date. On the other hand, *Interval Queries*, as the name suggests, are posed at regular intervals, and hence the answer cache is updated at regular intervals. In this section, we provide a detailed analysis of the algorithm for using this structure to efficiently answer the queries. First we will consider the frequent elements query, and then move on to top-$k$ query.

Algorithm 3 provides a scheme for answering continuous frequent elements queries. Since the elements are sorted, answering this query amounts to keeping a

**Algorithm 2** Supporting method for *CSSwSS*

```
/* READ is a TCAM operation. */

Procedure IncrementCounter(freq_node, element_index)
/* Increments the count of the specific element. */
new_fr ← freq_node→freq + 1
next_node ← freq_node→next
if (freq_node→count = 1 & next_node→freq ≠ new_fr) then
   /* This bucket can be promoted to become the new bucket.
   */
   freq_node→freq ← new_fr
   return freq_node
end if
RemoveElementFromList(freq_node, index)
if (next_node→freq = new_fr) then
   /* The element moves to the next node. */
   AddElementToList(next_node, index)
   return next_node
else
   /* A New Frequency Bucket need to be inserted next to the
   current node. */
   new_node ← AllocateNewNode()
   AddToNext(new_node, freq_node)
   AddElementToList(new_node, index)
   return new_node
end if
end Procedure IncrementCounter
```

**Algorithm 3** *Answering continuous frequent elements queries*

```
/* added_bucket and added_elem are the bucket added and
elements added.*/
Procedure ContinuousQueryFrequent(φ)
/*ptr_φ is the pointer to the bucket with minimum frequency
above support φ*/
/* φ is the minimum frequency to be reported. */
if (ptr_φ ≠ NULL) then
   next_node ← ptr_φ→next
   min_freq ← ptr_φ→freq
   if (min_freq < φ) then
      /* At least one element has become infrequent. */
      if (next_node→freq > min_freq) then
         ptr_φ ← next_node
      else
         ptr_φ ← NULL
      end if
   end if
   if (added_bucket→freq ≥ φ & added_bucket→freq <
   min_freq) then
      /* An infrequent element has become frequent. */
      ptr_φ ← added_bucket
   end if
else
   /* No frequent element yet. */
   if (added_bucket→freq ≥ φ) then
      ptr_φ ← added_bucket
   end if
end if
end Procedure ContinuousQueryFrequent
```

pointer to the bucket that has the minimum frequency above the support $\phi$. We can use the *CAM adapted Stream Summary* structure to efficiently maintain this pointer ($ptr_\phi$). The intuition behind this algorithm is that after processing an element, a bunch of elements might become infrequent, while only one element can become frequent. These elements can be determined from Theorems 4.1 and 4.3 and Corollary 4.2.

THEOREM 4.1. *When processing continuous frequent elements queries, if element $e_i \in bucket_i$ becomes infrequent then all elements $e_j \in bucket_i$ and only elements $e_j$ become infrequent.*

PROOF. The proof is of the theorem consists of two parts.

- *All elements $e_j \in bucket_i$ becomes infrequent if $e_i \in bucket_i$ becomes infrequent.* This is intuitive from the structure of the *CAM adapted Stream Summary* structure as all elements in the same frequency bucket have the same frequency.

- *Only elements $e_j$ become infrequent.* Since we are answering continuous queries, the length of the stream increases by 1 at each step. Now, since $\phi = \lambda N$, where $0 < \lambda < 1$ and $N$ is the length of the stream, and if $f_i$ is the frequency of $e_i$, as $e_i$ was reported as frequent in the previous step, and considering the fact that $N$ increased by 1 after the previous step, all $f_k > f_i$ must be reported as frequent. The *CAM adapted Stream Summary* structure ensures that all buckets to the right of $bucket_i$ (refer to Figure 2 for illustration) will have $f_k > f_i$. Therefore, only the elements $e_j$ become infrequent if $e_i$ become infrequent.

□

COROLLARY 4.2. *Only the elements with the minimum frequency amongst the set of reported frequent elements can become infrequent.*

PROOF. The proof follows from Theorem 4.1, since for each new element, only the elements from a single frequency bucket can become infrequent, and since the buckets in *CAM adapted Stream Summary* are in sorted order. □

THEOREM 4.3. *When processing continuous frequent elements queries, only the element seen last can become frequent.*

PROOF. An element $e_i$ with frequency $f_i$ will be reported as frequent $\iff$ $f_i > \phi$ where $\phi = \lambda N$ and $0 < \lambda < 1$. Since the number of elements $N$ is monotonically increasing, therefore $\phi$ is also monotonically increasing. If $e_l$ is the last element processed, then $f_l$ is the only frequency that increases over the previous step. Hence $e_l$ can be the only element that might become frequent. □

From the theorems, it is evident that updating $ptr_\phi$ induces a constant cost per element being processed: reporting an element becoming frequent is constant and reporting $p$ elements as infrequent is $O(p)$. This is independent of the number of elements in the stream or the number of elements monitored in the *CAM adapted Stream Summary* structure. This is a drastic improvement from the *CAM conscious Space Saving (CSS)* in [1], where the cost of query answering is $O(n)$, $n$ being the number of elements currently being monitored, and evidently, $p \ll n$ in most iterations.

Since answering continuous queries is not efficient for *CSS*, in our experiments we compare *CSS* with *CSSwSS* using varying query load, i.e., instead of the queries being continuous, now the queries are issued at regular

**Algorithm 4** *Answering frequent elements queries at regular intervals*

```
/* added_bucket and added_elem are the bucket added and
elements added.*/
Procedure IntervalQueryFrequent(φ)
/*ptr_φ is the pointer to the bucket with minimum frequency
above support φ*/
/* φ is the minimum frequency to be reported. */
if (ptr_φ ≠ NULL) then
   cur_fr ← ptr_φ
else
   cur_fr ← added_bucket
end if
next_node ← cur_fr→next
prev_node ← cur_fr→prev
freq ← cur_fr→freq
if (freq ≥ φ) then
   /*Check if any infrequent element has become frequent.*/
   while (prev_node→freq < cur_fr→freq & prev_node→freq
   ≥ φ) do
      cur_fr ← prev_node
      prev_node ← cur_fr→prev
   end while
   ptr_φ ← cur_fr
else
   /* Some frequent elements have become infrequent, so up-
   date the cache. */
   while (next_node→freq > cur_fr→freq) do
      cur_fr ← next_node
      next_node ← cur_fr→next
   end while
   if (next_node→freq < cur_fr→freq) then
      ptr_φ ← NULL
   else
      ptr_φ ← cur_fr
   end if
end if
end Procedure IntervalQueryFrequent
```

**Algorithm 5** *Answering continuous top-k queries*

```
/* added_bucket and added_elem are the bucket added and
elements added.*/
Procedure ContinuousQueryTopK(k)
/* Set_k is the set of elements in top-k cache with minimum
frequency. */
/* ptr_k is the pointer to the bucket containing elements in
Set_k*/
if (num_elems_topk < k) then
   /* There are not enough elements in the top-k set. */
   if (added_elem ∉ top-k set) then
      Report added_elem ∈ top-k
      num_elems_topk++
   end if
   if (num_elems_topk = k) then
      /* k elements have now been reported, enter maintenance
      mode. */
      ptr_k ← min_freq
      Set_k ← ElementsInBucket(ptr_k)
   end if
else
   /* Check for updates, if any. */
   ptr_k^+ ← ptr_k→next
   if (added_bucket = ptr_k^+ & ptr_k^+→freq − ptr_k→freq = 1)
   then
      if (added_elem ∈ Set_k) then
         Set_k ← Set_k − added_elem
      else
         Select elem ∈ Set_k
         Set_k ← Set_k − elem
         Report elem ∉ top-k
         Report added_elem ∈ top-k
      end if
      if (Set_k is empty) then
         ptr_k ← ptr_k^+
         Set_k ← ElementsInBucket(ptr_k)
      end if
   end if
end if
end Procedure ContinuousQueryTopK
```

interval. Algorithm 4 gives an overview of the scheme used for answering queries at regular intervals. Since the queries are not issued after processing every single element, Theorems 4.1 and 4.3 do not hold. Hence Algorithm 3 cannot be used for this purpose. It can be seen that Algorithm 4 uses an idea similar to that of Algorithm 3, except that Algorithm 4 scans through the structure, because the number of elements that have been processed since the last invocation is not known. But it can be easily proved that the time taken by Algorithm 4 is $O(m)$, where $m$ is the number of elements processed since the last invocation. So, in the worst case it might reduce to a continuous query, but as demonstrated in the experiments in a later section, the performance is much better for most practical cases.

The *CAM adapted Stream Summary* structure can be used to efficiently answer continuous top-$k$ as well. The idea is the same as with continuous frequent elements queries and the top-$k$ set is selected using the layout of the *CAM adapted Stream Summary* structure. The algorithm is very similar to the corresponding algorithm in [17] and has been adapted to efficiently leverage the *CAM adapted Stream Summary*. An overview of the algorithm for continuous top-$k$ monitoring is provided in Algorithm 5.

Again, it is straightforward to see that the per-element cost of continuous maintenance of set of top-$k$ elements is pretty small and the *CAM adapted Stream Summary* structure can be used to efficiently maintain the top-$k$ set. It is therefore evident that *CSSwSS* can be used to efficiently answer frequent elements and top-$k$ queries.

# 5. EXPERIMENTAL EVALUATION

We implement the *CAM conscious Space Saving with Stream Summary (CSSwSS)* algorithm on a commodity TCAM chip IDT75K62134 [13] which interfaces efficiently with the IXP2800 NPU [15]. The TCAM chip has $128K$ 72-bit wide entries, supports programmable word size, and up to 128 parallel request contexts. Development is done using the Teja NP ADE [21] and Intel Development Workbench [14]. Implementation of the algorithms involve coding in $TejaC^{TM}$ and $MicroC^{TM}$. Times reported are actual execution times of the algorithms on the MEs (Micro Engine), and are obtained from the *Timestamp* register common to all the ME's. The experiments have been repeated multiple times and the values have been averaged over multiple runs.

The experiments have been performed with synthetic Zipfian data which has been shown to closely resemble realistic data sets [24]. The data set used for experiments consists of 1 million hits, taken from an alphabet of size $10,000$. The alphabet is the number of distinct elements in the stream. Since the performance of the *Space Saving* algorithm is not dependent on the size of the alphabet, we expect similar results for smaller or larger alphabet sizes. The zipfian factor is varied from 0 to 3 in steps of 0.5, where zipfian factor 0 represents uniform distribution while 3 is a highly skewed distribu-

| TCAM Operation | 72-bit | 144-bit |
|---|---|---|
| READ | 0.32 | 0.64 |
| LOOKUP | 0.2971 | 0.3673 |
| WRITE | 0.32 | 0.3314 |

**Table 1: Time (in secs) for a million TCAM operations for different sizes of TCAM core.**

tion. The error bound $\epsilon$ is set to 0.001. We compare the performance of the proposed algorithms with the CAM conscious *Space Saving* adaptation in [1], which we refer to as *CAM conscious Space Saving (CSS)*.

## 5.1 Cost of Frequency Counting

In this section, we experimentally evaluate the two possible design alternatives. First we need to determine the cost of the primitive operations in narrow and wide TCAM cores. The operations of interest are `LOOKUP`, `READ` and `WRITE`. We evaluate this using an experiment where we time only the operations of interest, and Table 1 summarizes the results.

There are a few interesting observations about the statistics in Table 1. `LOOKUP` is one of the most important operations, and this operation is done at least once for each stream element. As we increase the width of the TCAM core, the cost of `LOOKUP` increases, but the increase in the cost of `WRITE` does not increase significantly. The highest increase is for the `READ` operations where the time almost doubles, as the chip used only supports up to 72-bit wide `READ` operations. As a result, a 144-bit `READ` comprises of two 72-bit `READ` operations. But this is not very alarming as the layout of elements can be designed in a manner that would never require a 144-bit `READ`, i.e. the algorithm will need either the first or the second 72-bit entry but not both, so that only the cost of 72-bit wide `READ` operations is incurred.

Another interesting observation about the times in Table 1 is that the `LOOKUP` throughput is nowhere near the peak throughput of about 100 million `LOOKUP`s per second as stated in Section 3.2. This is primarily because we use only a single ME for implementing our algorithm and accessing the TCAM, and the high throughput can be obtained by parallel accesses to the TCAM. We will exploit this parallelism in future work. In addition, it must be noted that we are using a single ME that runs at a speed of only 1.4GHz.

Before evaluating the performance of the algorithms for answering queries, we evaluate the cost of counting the frequencies and keeping them sorted. Since the authors in [1] provide an efficient frequency counting algorithm (*CSS*), we compare the performance of the proposed schemes with *CSS*. Figure 4 provides a comparison of time taken for frequency counting. From the figure it can be seen that for uniform data, keeping the elements sorted is almost twice as costly as simple frequency counting. But as the data becomes skewed, the elements can be kept sorted almost for free. This difference in performance is due to the fact that the proposed scheme involves managing the structure of frequency buckets, which becomes costly when the sort order of elements is changing rapidly, which is the case for the uniform distribution (zipfian factor close to 0). On the other hand, when the data is skewed, the sort order does not change much, and the proposed scheme performs better as the cost of maintaining the minimum frequency element in *CSS* now becomes significant, which is not present in *CSSwSS*.

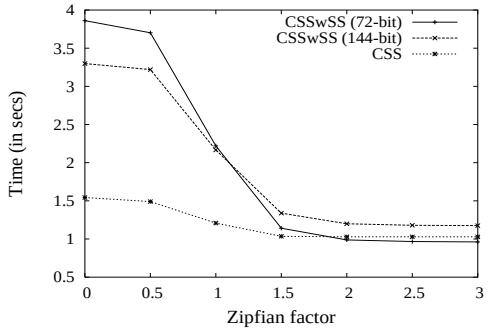


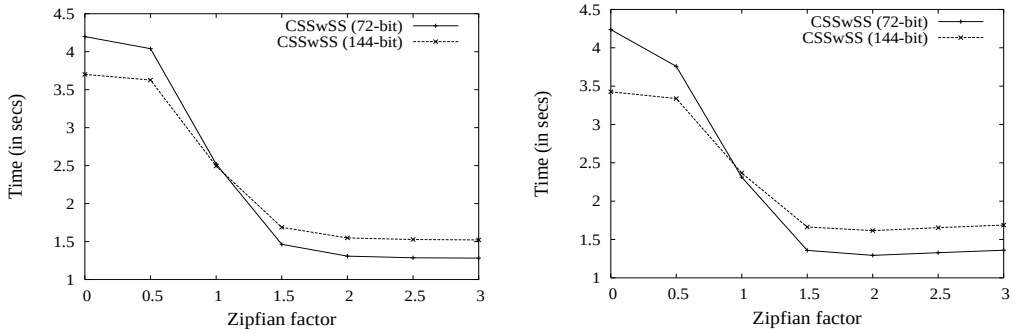

**Figure 4: Comparing the performance of *CSS* with *CSSwSS* for 72 & 144 bit TCAM core sizes. This figure reports only the time for frequency counting.**

Now analyzing the performance of the two alternative designs, it can be seen from Figure 4 that the approach using 72-bit TCAM core performs better for moderate and heavily skewed distributions, whereas the 144-bit design performs better for uniform distribution. This is because for uniform data, the *CAM adapted Stream Summary* structure undergoes a lot of changes, and the overhead due to the added level of indirection for the 72-bit implementation dominates. However, for skewed data, the increased cost of the 144-bit TCAM operations makes the 72-bit implementation cheaper.

Since the zipfian factor ranging between 1 and 2 represents most realistic data sets, from Figure 4 we can conclude that the implementation using 72-bit wide cores would perform better for real data. In addition to the savings in time, another advantage of the 72-bit implementation is that it saves TCAM space which can be a scarce resource.

## 5.2 Cost of Answering Queries

In this section, we analyze the cost of answering frequent elements and top-k queries using the proposed algorithm. Since the proposed algorithm keeps the elements sorted, it can therefore be easily adapted to answer queries efficiently and incrementally. As *CSS* does not maintain the elements in sorted order, there is no straightforward technique for the incremental reporting of frequent elements. Every time a query arrives, the *best-effort* adaptation of *CSS* would scan through all the counters to report the frequent elements. On the other hand, the *CAM adapted Stream Summary* structure in *CSSwSS* can be leveraged to incrementally report the frequent elements. Section 4.3 provides efficient algorithms to answer queries and also provides a thorough analysis of the cost incurred by these algorithms. In this section, we evaluate these algorithms experimentally.




(a) Frequent elements (b) Top-$k$

**Figure 6: Comparing performance of implementations of *CSSwSS* answering continuous queries.**

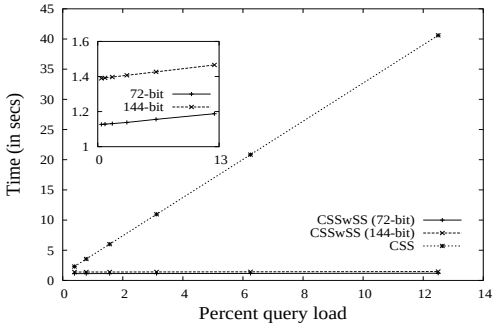



**Figure 5: Comparison of times for answering continuous frequent elements queries for input data of Zipfian factor 1.5.**

Figure 5 compares the time taken to answer frequent elements queries by the two implementations of *CSSwSS*, and by *CSS*. For this experiment, we vary the query load, and the load is measured as a percentage of queries per update. The efficiency of *CSSwSS* is evident from the fact that even at moderately low query loads of about 10%, it is an order of magnitude faster than *CSS*. We repeated the above experiment for the same range of zipfian factors (0.0 – 3.0), but report the one with zipfian factor of 1.5 as this data set is illustrative of real data. We observed similar behavior for other values of zipfian factor. The graph in the inset compares the two implementations of *CSSwSS* and it can be seen that the 72-bit implementation is a clear winner.

Our proposed algorithms can efficiently answer continuous queries and this is illustrated in Figure 6 where we evaluate the performance of the proposed scheme when answering continuous queries. Figure 6(a) represents continuous frequent elements queries whereas Figure 6(b) illustrates continuous top-$k$ queries. Again, as *CSS* does not sort the elements, it cannot efficiently answer continuous frequent elements and top-$k$ queries, so we do not consider it for this comparison. These results corroborate the analysis performed in Section 4.3. Figures 4 and 6 have similar trends for the two implementations of *CSSwSS*. This is because answering the queries only involves processing the linked structure of frequency buckets, which is identical for both the implementations. Therefore, the 72-bit implementation would be suited for most practical data sets.

## 5.3 Discussion

In Section 4 we propose efficient CAM conscious algorithms and in Sections 5.1 and 5.2 we evaluate these algorithms to demonstrate the gains over the existing CAM conscious technique. But there are some important implications of the experimental results. *First*, in the experimental prototype used for the experiments in this paper, the cost of a single TCAM operation is about 300ns, which is an order of magnitude higher than the typical TCAM speeds used in simulation of different TCAM algorithms. For example, authors in [20] report typical TCAM `LOOKUP` operation taking around 20ns. The TCAM chip used for our experiments, the IDT75K62134 chip from IDT [13], can support about a 100 million lookups per second, which gives per operation cost of 10ns. From our analysis of the cost of TCAM operations we inferred the following as primary reasons for the difference in performance: the added level of abstraction provided by Teja ADE which is used for implementing the algorithms, the difference in the word sizes of the NPU and the TCAM, and the use of a single Micro Engine to access the TCAM. *Second*, a simple analysis of the algorithms reveals that on an average, for a uniform distribution, every element incurs $5-7$ TCAM operations while for a skewed distribution it incurs $1-3$ TCAM operations. From the times reported in Figure 4 and Table 1 it can be seen that about $50-70\%$ of the frequency counting time is spent on the TCAM operations. Therefore, a decrease in the cost of TCAM operations would considerably reduce the time taken by the CAM conscious algorithms.

Another important point to note is that these times cannot be compared directly with the times of a software hash-based technique running on a standard computer. The primary reason is the difference in the two architectures. As pointed out in Section 3.1, the NPU has a very simple design, whereas most standard processors have higher clock speed and are highly optimized for

single thread performance. But as demonstrated in [2], TCAM does not interface well with conventional processors and requires an Application Specific IC (ASIC) which becomes the communication bottleneck. Therefore, even though TCAM provides good potential for designing efficient stream management algorithms, practical limitations restrict the gains obtained from a real experimental setup such as the NPU-TCAM prototype used in this paper.

# 6. CONCLUSION

In this paper, we propose a CAM conscious integrated solution for answering frequent elements and top-$k$ queries. We provide an implementation of the proposed algorithm using commodity CAM chip. Evaluation using realistic synthetic data sets show that CSSwSS performs an order of magnitude better compared to the existing technique even with moderate query loads of about 10%. Therefore we can see that the proposed scheme can efficiently answer frequent elements queries. In addition, CSSwSS can efficiently answer continuous queries as well. We analyze two design alternatives for implementing our proposed scheme, and from the experiments, we conclude that the implementation using 72-bit TCAM core performs better for non-uniform data distributions.

Although we use the NPU for our implementation, we do not leverage the parallelism provided by the NPU and supported by the TCAM. In the future, we plan to exploit this parallelism to further improve the processing rates. In addition, we plan to explore the possibility of using TCAM for other stream management operations.

# 7. REFERENCES


[1] N. Bandi, A. Metwally, D. Agrawal, and A. E. Abbadi. Fast data stream algorithms using associative memories. In *SIGMOD*, pages 247–256, Beijing, China, 2007.

[2] N. Bandi, S. Schnieder, D. Agrawal, and A. E. Abbadi. Hardware Acceleration of Database operations using Content Addressable Memories. In *DaMoN*, 2005.

[3] B. Bhattacharjee, N. Abe, K. Goldman, B. Zadrozny, V. R. Chillakuru, M. del Carpio, and C. Apte. Using secure coprocessors for privacy preserving collaborative data mining and analysis. In *DaMoN '06*, 2006.

[4] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *ICALP'02*, pages 693–703, 2002.

[5] J. Cieslewicz and K. A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *VLDB 2007*, pages 339–350, 2007.

[6] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, pages 183–194, 2007.

[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *ESA*, volume 2461, pages 348–360, 2002.

[8] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Gpuqp: query co-processing using graphics processors. In *SIGMOD '07*, pages 1061–1063, 2007.

[9] M. Fischer and S. Salzberg. Finding a majority among n votes: Solution to problem 81-5. In *Journal of Algorithms 3(4)*, pages 376–379, 1982.

[10] B. T. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *DAMON '05*, 2005.

[11] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, 2001.

[12] N. Hardavellas, I. Pandis, R. Johnson, N. G. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 79 – 87, 2007.

[13] Integrated Devices Technologies, Integrated IP Co-processor IDT 75K62134. `http://idt.com/?genID=75K62134&source=products_genericPart_75K62134`, 2006.

[14] Intel Internet Exchange Architecture for Network Processors. Technical report, Intel Corp., 2002.

[15] Intel IXP 2800 Network Processor Product Brief. `http://www.intel.com/design/network/prodbrf/279054.htm`, 2006.

[16] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.

[17] A. Metwally, D. Agrawal, and A. E. Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.*, 31(3):1095–1133, 2006.

[18] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[19] R. Panigrahy and D. Thomas. Finding Frequent Elements in Non-Bursty Streams. In *ESA*, pages 53 – 62, October 2007.

[20] D. Shah and P. Gupta. Fast updating algorithms for tcams. *IEEE Micro*, 21(1):36–47, 2001.

[21] Teja networking systems and teja np application development platform. `http://www.teja.com`, 2006.

[22] S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*, 2005.

[23] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. Improving database performance on simultaneous multithreading processors. In *VLDB '05*, pages 49–60, 2005.

[24] G. K. Zipf. *Human Behavior and The Principle of Least Effort*. Addison-Wesley, Cambridge, MA, 1949.

[25] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *DaMoN '06*, 2006.

# Modeling the Performance of Algorithms on Flash Memory Devices

Kenneth A. Ross
IBM T. J. Watson Research Center and Columbia University
rossak@us.ibm.com, kar@cs.columbia.edu

## ABSTRACT

NAND flash memory is fast becoming popular as a component of large scale storage devices. For workloads requiring many random I/Os, flash devices can provide two orders of magnitude increased performance relative to magnetic disks. Flash memory has some unusual characteristics. In particular, general updates require a page write, while updates of 1 bits to 0 bits can be done in-place. In order to measure how well algorithms perform on such a device, we propose the "EWOM" model for analyzing algorithms on flash memory devices. We introduce flash-aware algorithms for counting, list-management, and B-trees, and analyze them using the EWOM model. This analysis shows that one can use the incremental 1-to-0 update properties of flash memory in interesting ways to reduce the required number of page-write operations.

## 1. INTRODUCTION

Solid state disks and other devices based on NAND flash memory allow many more random I/Os per second (up to two orders of magnitude more) than conventional magnetic disks. Thus they can, in principle, support workloads involving random I/Os much more effectively.

However, flash memory cannot support general in-place updates. Instead, a whole data page must be written to a new area of the device, and the old page must be invalidated. Groups of contiguous pages form erase units, and an invalidated page becomes writable again only after the whole erase unit has been cleared. Erase times are relatively high (several milliseconds). Flash-based memory does, however, allow in-place changes of 1-bits to 0-bits without an erase cycle [5]. Thus it is possible to reserve a region of flash memory initialized to all 1s, and incrementally use it in a write-once fashion.

Traditional measures of algorithm complexity do not model flash I/O behavior well, because the high cost of a general update (relative to a 1-to-0 update) is not accounted for. Previous models for write-once memory ("WOM") have been proposed to model devices like paper tape and optical disks in which the write process is destructive, so that once a bit is set it cannot be unset [10]. Maier proposes using write-once storage for a "Read-Mostly Store" (RMS) where the memory is gradually consumed as updates occur [8]. However, these models are too restrictive for devices like flash memory where a bulk erase allows memory to be reused.


*Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008),* June 13, 2008, Vancouver, Canada.


### 1.1 The EWOM Model

We propose a new model for evaluating an algorithm on a flash-like device. We call it the "Erasable Write Once Memory" model, or the "EWOM" model. In addition to counting traditional algorithmic steps, we count a page-write step whenever a write causes a 0 bit to change to a 1 bit. If an algorithm performs a group of local writes to a single page as one transactional step, we count the group as a single page-write step. Even if only a few bytes are updated, a whole page must be written.

The true cost of a page-write step has several components. There is an immediate cost incurred because a full page must be copied to a new location, with the bits in question updated. If there are multiple updates to a single page from different transactional operations, they can be combined in RAM and applied to the flash memory once, although one must be careful in such a scheme to guarantee data persistence if that is an application requirement.

There is also a deferred cost incurred because the flash device must eventually erase the erase unit containing the old page. It is a deferred cost because the write itself does not have to wait for the erase to finish; the erase can be performed asynchronously. Nevertheless, erase times are high, and a device burdened by many erase operations may not be able to sustain good read/write performance. Further, in an I/O intensive workload a steady state can be reached in which erasure cannot keep up, and writes end up waiting for erased pages to become available.

There is an additional longer-term cost of page erases in terms of device longevity. On current flash devices an erase unit has a lifetime of about $10^5$ erases. Thus, if special-purpose algorithms reduce the number of erases

needed by a factor of $f$, the expected lifetime of the device can in principle be multiplied by $f$.

Our model can distinguish between situations where the I/O device is saturated, and where the device is lightly loaded. Algorithms might include a low-priority background process that asynchronously traverses data structure elements and reorganizes them to improve performance. The extra I/O workload will not be noticeable in a lightly-loaded setting, and most data structure elements will end up in the optimized state. In a saturated or near-saturated scenario, however, the background process will rarely run, and the data structure elements will remain in the unoptimized state.

We choose not to model "seek" time for flash memory. While there is a small overhead involved in moving from one memory location to another, this overhead is small relative to the erase costs. Further, this cost is orders of magnitude smaller than seek times for magnetic disks, whose performance models often distinguish between sequential and random I/O.

Traditional I/O devices have a fixed block transfer size, and it is customary to count the number of blocks transferred when measuring I/O complexity. RAM allows fine-grained data access, and so it is customary to simply count the number of computational steps to perform a given operation as the complexity measure. Flash memory occupies a middle-ground between traditional I/O devices and RAM. Some flash devices require transfers to happen in block-sized units, where a single device may support multiple block sizes, while others allow fine-grained access. For the purposes of the present work, we will adopt the convention that the flash memory is a fine-grained access device for reads and 1-to-0 writes, and we measure complexity by counting the total number of computational steps. For general updates, we also count a page write.[1]

## 1.2 Pages and Erase Units

Erase units are typically large, around 128KB. Copying a full erase unit on every update would not be efficient. It is therefore common for data copying to happen in page-sized units, where the page size $P$ depends on how the device is configured. A typical value of $P$ might be 2KB, meaning 64 pages in a 128KB erase unit.

We assume that there is a memory mapping layer that maps logical page addresses to physical page addresses. Such mapping is commonly implemented in hardware within page-granularity devices: when an update happens, the physical address changes, but the logical address remains the same so that updates do not need to be propagated to data structures that refer to the data page. When the device itself does not provide such a layer, it is common to implement such a layer in software. The mapping layer also ensures that wear on the device is shared among physical pages, because flash pages have a limited lifetime of approximately $10^5$ erase cycles. The mapping layer can also hide faulty or worn-out pages from the operating system. The EWOM model assumes that a logical-to-physcial mapping layer is present.

If updates are performed on pages, then at any point in time, an erase unit may contain some valid pages and some invalid pages that need to be erased. If an erase unit contains valid pages, then those valid pages must be written to alternate locations before the erase unit can be erased. We assume that the same hardware or software that monitors the logical-to-physical mapping of pages also monitors the validity of pages for the purposes of managing erase units for garbage collection.

In a lightly loaded device, such extra copying might not be noticeable. However, in a heavily loaded system, with high demand for new erase units, this overhead will be noticeable.

A "best-case" workload for erase-unit recycling would occur when all pages in an erase unit are invalid at erase time. This kind of workload might happen if the data access pattern is highly clustered, such as when a file is sequentially updated, page by page. In that case, each page write contributes to approximately $P/E$ erases, where $E$ is the size of an erase unit.

A "worst-case" workload would occur when all erase units available for recycling hold just one invalid page. This kind of workload might happen on a device that is almost full, and for which the data access pattern is scattered over the various erase units. In this case, each page write causes an erase.

There are obviously many intermediates between the best and worst cases, and the range is wide. Thus it is not always possible to predict the erase frequency given just the page update frequency. More information about workload characteristics is usually needed.

# 2. COUNTING

We begin our analysis with a simple task: maintain a counter in EWOM storage. The counter is initialized to zero, and may be incremented. A naive, in-place solution would rewrite the counter, stored in conventional binary form, on every update. Since an increment always changes some 0 to a 1, every update requires a page write. Reads have cost proportional to the word size $W$ of the counter in binary.[2]

An alternative solution represents the counter in unary form, with the number of zero bits indicating the count. An increment operation can be handled by changing a bit from 1 to 0 without a page-write operation. Unary counters are severely limited in their counting capacity since they have space complexity linear in the current value of the counter. Reads and writes can be handled in logarithmic time using an exponential expansion followed by a binary search to find the first 0 in the bit array.

A hybrid scheme stores a binary base counter, together with a unary increment counter of fixed length $L$, where $L \leq P - W$ so that the counter fits in a page [2]. The counter is computed by adding the base counter to the offset of the first zero in the unary array, which can be found using binary search. A page-write is needed

[1] If we fill a page using simple 1-to-0 writes, there are no page write operations counted.

[2] Depending on one's memory model, $W$ is either constant or $O(\log n)$, where $n$ is the value of the counter.

| Method | Space (bits) | Read time | Write time | Page-Writes |
|---|---|---|---|---|
| Naive | $W$ | $W$ | $2W$ | 1 |
| Unary | $n$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $\frac{1}{P}$ |
| Hybrid (lightly loaded) | $W + L$ | $W + 1$ | 2 | 0 |
| Hybrid (saturated) | $W + L$ | $W + O(\log L)$ | $O(\log L) + \frac{2W}{L}$ | $\frac{1}{L}$ |

**Figure 1: Amortized complexity for counting**

every $L$ steps, at which time the base counter is recomputed, and the unary counter is reset.

A low-priority asynchronous operation may look through pages containing counters, and also perform this recompute/reset operation. We assume that in the lightly loaded case, the asynchronous background updates happen at least as often as writes, and promptly after those writes. This assumption means that the state of the counter on the flash device will usually have a zero unary increment value, with the binary part of the counter containing the current count. Reads become simpler (because they don't have to traverse the unary increment value), and writes become simpler (because there is always space for unary increments — no page-writes are necessary).

The complexity of these alternatives is summarized in Figure 1, where $n$ is the number of increment operations, and $P$ is the size of a page in bits. Read time is measured in terms of the number of bit operations needed. For the write step, we assume that the writer does not know the previous value of the counter, only that the counter needs to be incremented. As Figure 1 shows, the hybrid counting method amortizes page-writes almost as well as the unary method, while keeping read and write performance close to the naive method.

## 2.1 Arbitrary Increments

One can generalize the hybrid method if increments (or decrements) by arbitrary amounts are possible. A single base counter is maintained in binary form. A unary counter is kept for recording increments by multiples of $2^0$, $2^1$, $2^2$, etc. An increment is broken down into its binary form, and the corresponding unary counters are updated. A separate set of counters is maintained for decrements. Read operations need to scan through the various counters to compute the net change to the binary stored value.

In the event that one of the unary counters is full, it may still be possible to process an addition without a page write by decomposing the addition into a larger number of smaller increments. For example, if the unary counter corresponding to $2^5$ is full, we could add the value $2^5$ by appending two bits to the unary counter corresponding to $2^4$.

Other configurations are also possible. For example, instead of recording increments using a unary counter for each power of 2, one could use unary counters for powers of an arbitrary value $k$. The number of bits to set for each counter would be determined by the corresponding digit of the value to be added when written in base-$k$ notation.

# 3. LINKED LISTS

A linked list is a commonly used data structure. In an EWOM context, standard list operations would require a page write. A page write would be needed to keep track of the tail of the list, to implement list element deletion, to insert an element into the list, and to update nodes within the linked list.

Suppose that we interpret the all-1 bit pattern as a NULL pointer. Then one can append to the list using only 1-to-0 updates by updating the NULL pointer in the last element of the list to point to a new element. The new element itself would be written in an area of the page initialized to all-1s. Unlike traditional append operations to a list, this variant would need to first traverse the entire list. On the other hand, a page-write is avoided.

Deletions would need to be handled in an indirect way, such as by using a "deleted" flag within the node structure. This would complicate list traversal slightly, because deleted nodes would remain in the list and need to have their flags checked.

Like for counting, we could implement a low-priority background process that "cleans up" lists on a page and writes a new page. In this new page, the deleted elements would be omitted. One could also store a shortcut to the current tail, so that future append operations do not have to start from the head of the list.

# 4. BLOOM FILTERS

Some data structures are inherently monotonic in their update behavior, and map well to the EWOM model without modification. An example is the Bloom filter [1]. If we interpret a vector of 1 bits to mean the empty Bloom filter, then every insertion can be achieved by setting some 1 bits to 0 bits. In the EWOM model, insert operations do not need to perform any page-writes.

# 5. B-TREES

Within a database system, one of the places where random I/O occurs frequently is in accessing B-tree indexes in response to OLTP workloads. Indexes are searched (to find the record to update), new records are inserted, and old records are deleted. One way to deal with B-tree update-heavy workloads is to batch the updates. That way, the costs associated with restructuring a page can be amortized over many updates. Batching happens implicitly when a page resides in the database system's buffer pool.[3] Batching can also happen close to the

[3]Note that the database system is ensuring persistence in this case by maintaining a recovery log.

physical device in a RAM-based cache. However, if the locality of reference of the database access is poor, such as when the table and/or index is much bigger than the buffer pool and records are being accessed randomly, there will be little effective batching in practice.

We therefore propose a new way to organize leaf nodes in a B-tree to avoid the page-write cost most of the time, while still processing updates one at a time. We focus on leaf nodes because that is where the large majority of changes happen.

Suppose that an entry in a leaf node consists of an 8-byte key, and an 8-byte RID referencing the indexed record. We assume a leaf node can hold $L$ entries, taking $16L$ bytes. We shall assume that a leaf node has size that exactly matches the page size of the device.

With the requirement that leaf nodes be at least half full, a conventional B-tree leaf node will contain between $L/2$ and $L$ entries stored in sorted key order. The ordering property allows for keys to be searched in logarithmic time using binary search.

A first attempt at a page-write-friendly leaf node would be to store all entries in an append-only array in the order of insertion [8]. A bitmap would be kept to mark deleted entries. When the node becomes full, it is split, and (nondeleted) entries are divided among the two resulting pages. The obvious drawback of this approach is that search time within the node will be linear rather than logarithmic, dramatically slowing down both searches and updates.

## 5.1 The Proposed Approach

Apart from the initial root node, all leaf nodes are created as a result of a split. When a split happens, we sort the (nondeleted) records into key order, and store them in that order in the append-only array. We keep track of the endpoint of this array by storing it explicitly in the leaf node. Subsequent insertions are then appended to the array as before.

So far, we have improved performance slightly because one can do a binary search over at least half of the entries, followed by a linear search of the remaining entries to find a key. However, the asymptotic complexity is still linear in the size of the array.

To speed up the search of the newly-inserted elements we store some additional information. Choose positive integer constants $c$ and $k$. For every $c$ entries in the new insertions, we store a $c$-element index array. Each entry in this index array stores an offset into the segment of new insertions, and the index array is stored in key order. (It is not maintained incrementally; it is generated only when there have been $c$ new insertions.)

To search an array of $m$ new elements ($m \leq L$), we need at most $(m/c)\log_2 c + (c-1)$ comparisons. While we have reduced the asymptotic search time by a factor of $c/\log_2 c$, it remains linear in $m$. The trick is to apply this idea recursively.

Suppose that after $kc$ elements, instead of a $c$-element offset array, we store a $kc$-element offset array covering the previous $kc$ newly inserted records. Now we need at most one linear search of at most $c-1$ elements, at most $k-1$ binary searches of $c$ elements, and $\lfloor \frac{m}{kc} \rfloor$ binary searches of $kc$ elements. If we keep scaling the offset array each time $m$ crosses $c$, $kc$, $k^2c$, $k^3c$ etc., then the total cost is $O(\log^2 m)$. (There are $O(\log m)$ binary searches, each taking $O(\log m)$ time.)

A complete search therefore takes $O(\log(n/L)+\log^2 L) = O(\log n+\log^2 L)$ time, where $n$ is the number of elements in the tree.

The space overhead of this approach is the total size of the index arrays. This size is equal to

$$\begin{aligned} & \lfloor \tfrac{m}{c} \rfloor c + \lfloor \tfrac{m}{ck} \rfloor (ck-c) + \lfloor \tfrac{m}{ck^2} \rfloor (ck^2-ck) + \ldots \\ \approx \quad & m \log_k (m/c) = O(m \log m). \end{aligned}$$

The overhead for one node is thus $O(L \log L)$, and the overhead for the entire tree is $O(n \log L)$. This is a classical computer-science trade-off in which we use more space to reduce the time overhead. Different choices for $c$ and $k$ represent alternative points in the space-time trade-off.

In practice, the space overhead is unlikely to be onerous. For example, suppose that the page size is 16KB. 8KB can be devoted to new entries and the offset arrays. This places an upper bound of 512 new entries. If $c = 32$ and $k = 3$, the largest index array we will build will have 288 entries. The total space in bytes to store $m$ new entries is then

$$16m+32(\lfloor m/32 \rfloor)+(96-32)(\lfloor m/96 \rfloor)+2(288-96)(\lfloor m/288 \rfloor).$$

(Here, we're assuming one byte offsets for up to 255 elements, and two-byte offsets for 256 or more elements.) Based on these numbers, we could store 446 new entries in the leaf node before we ran out of space. 1056 bytes out of 16K bytes (6.4%) is the space overhead, ignoring the pointer to the start of the new elements and the bits to record deletions.

Under lightly loaded conditions, where one has spare cycles to do background leaf optimization, one could convert a leaf node to sorted format and reset the pointers to new entries, writing the resulting node to a new memory location. For such "fresh" leaf nodes, search time goes down from $O(\log^2 m)$ time to $O(\log m)$ time. Note that because of the logical-to-physical page mapping, parent nodes are unchanged by leaf freshening.

## 5.2 Analysis

Every $c$ entries, an updating transaction needs to sort $c$ elements costing $O(c \log c)$ time. When the system gets to a $k^i c$-byte boundary, it only needs to sort the last $c$ elements, then merge $k$ ordered lists of size $k^{i-1}c$, which can be done in $O(c \log c + k^i c \log k)$ time. Amortizing over all insertions, the cost per insertion has order

$$\begin{aligned} & \textstyle\sum_{i=1}^{\lfloor \log_k (m/c) \rfloor} (k^i c \log k)(\lfloor \frac{m}{ck^i} \rfloor)/m \\ \approx \quad & \log k \lfloor \log_k (m/c) \rfloor \approx \log(m/c) \end{aligned}$$

Similarly, split processing can merge the array segments rather than fully sorting the array.

One needs to know where the array of new values ends, in order to decide when to terminate the search, and where to append new values. The simplest way to do this is to assume that a pattern of all 1-bits is not a valid (key,RID) pair. One can then binary search to find the last valid pair. One could try to explicitly store

| Method | Space (bits) | Read time | Write time | Page Writes |
|---|---|---|---|---|
| Standard | $O(n)$ | $O(\log n)$ | $O(\log n)$ | 1 |
| Append-Only | $O(n)$ | $O(\log n + L)$ | $O(\log n)$ | $\frac{1}{L}$ |
| Hybrid (lightly loaded) | $O(n \log L)$ | $O(\log n)$ | $O(\log n)$ | 0 |
| Hybrid (saturated) | $O(n \log L)$ | $O(\log n + \log^2 L)$ | $O(\log n + \log L)$ | $O(\frac{\log L}{L})$ |

**Figure 2: Amortized B-tree complexity for a tree of size $n$, treating $c$ and $k$ as constants.**

| |
|---|
| Pointer to the endpoint of initial sorted prefix. |
| Prefix $P$ containing sorted (key,RID) pairs. Calculated during most recent page-write. |
| Sequence $S$ of (key,RID) pairs in insertion order. |
| Indexes of first group of $c$ elements of $S$, in sorted order. |
| Indexes of second group of $c$ elements of $S$, in sorted order. |
| ... |
| Indexes of $k - 1^{\text{st}}$ group of $c$ elements of $S$, in sorted order. |
| Indexes of first group of $kc$ elements of $S$, in sorted order. |
| Indexes of $k + 1^{\text{st}}$ group of $c$ elements of $S$, in sorted order. |
| ... |
| Deletion bits |
| Log Sequence Number (using generalized counter) |

**Figure 3: The final structure of a B-tree node of (key,RID) pairs.**

the offset using the counters of Section 2, but such a method would consume more space than necessary.

Figure 2 shows the amortized asymptotic complexity for the proposed B-tree structure. We assume that writes do not need to check whether the key already exists in a node before insertion. If such a check is necessary, the entry for the append-only method would become $O(\log n + L)$ and the entry for the hybrid method with saturated writes would become $O(\log n + \log^2 L)$. Note that even in the saturated setting, the hybrid method only needs a page-write every $O((\log L)/L)$ insertions, while having better asymptotic read complexity than the append-only method.

## 5.3 Refinements

We have assumed that leaf nodes contain (key,RID) pairs. Sometimes, to save space, B-tree leaf nodes are designed to associate a key with a list of RIDs. The proposed structure can be modified so that at the time of reorganization (i.e., when a page-write occurs), the initial segment of data is in (key,RID-list) form. An alternative would be to keep a linked list of RIDs for each key, using the linked-list techniques described in Section 3.

In real B-tree implementations, a leaf node contains a log sequence number (LSN) recording information relevant for node recovery in case of failure. On an EWOM device, the LSN could be implemented using a generalized counter as described in Section 2.1. Note that LSNs are monotonically increasing, meaning that only increments, not decrements, need to be considered.

The final structure of a B-tree node is summarized in Figure 3. This figure shows a node containing (key,RID) pairs. If RID-lists were used, a region within the page would be used as a heap for allocating new RID nodes to add to RID-lists.

# 6. RELATED WORK

An interesting technique related to counting was proposed by Rivest and Shamir for the WOM model [10]. They show, for example, that it is possible to overwrite an arbitrary number from $\{0, 1, 2, 3\}$ with another arbitrary number from that set (a) using only monotonic bit changes, and (b) with only 3 bits of storage. Each possible number has two valid 3-bit encodings, such as

$$0 : 000, 111;\ 1 : 001, 110;\ 2 : 010, 101;\ 3 : 100, 011$$

The first code for a number is used for the initial write. The second code is used for the subsequent write, unless the second write has the same value, in which case there is no change. One could extend techniques like this to the EWOM model by erasing when necessary, which in this example would be after two or more updates.

Others have studied B-tree implementations for flash devices. Wu et al. [11] describe a B-tree method that uses a combination of RAM-resident buffers and "index units" representing flash-resident incremental changes to a B-tree node. The logical view of a B-tree node is reconstructed using the node together with these index units. The work of Wu et al. assumes a page-level interface to the flash device, without fine-grained access.

Nath et al., also study B-tree indexes on flash devices, with the aim of minimizing power and maximizing performance on a low-power mobile device [9]. Their system optimizes B-tree parameters in a self-tuning fashion, based on the workload and device characteristics. They also employ a page-level interface to the flash memory.

OLTP workloads frequently need to perform small updates in place. Lee and Moon show how to restructure

database pages and modify the logging protocol to minimize the required number of page erases [7]. Multiple versions of a data element are kept on a page in a write-once log-like structure within the page, and reads must consult the log to look for changes. Data is written to the flash storage in sector-sized units (512 bytes in [7]).

# 7. CURRENT DEVICES

The two basic types of flash memory available today are NOR-flash and NAND-flash. These technologies have contrasting behaviors that make them suitable for different classes of application [4]. For example, NAND-flash tends to have larger capacity, faster writes and erases, and page-level data access. NOR-flash tends to have faster reads, and fine-grained random access to data. Hybrid NAND/NOR devices exist (e.g., [6]).

The types of flash memory interaction allowed by a device vary. Some devices implement only a page-level API such as FTL [5], and updates to pages always cause a new page to be written. Such a choice allows an SSD device to resemble a magnetic disk device, and be used in existing systems that employ disk devices. Other devices (together with a software layer) expose flash as a "Memory Technology Device" (MTD) via UBI [3], which allows partial updates to pages. Low level flash interfaces have been defined by the ONFI working group[4]. In this paper, we assume an interface in which partial writes to a page are allowed, as long as they only involve trasitions from a 1 bit to a 0 bit.

Not every flash device may provide an interface that allows fine-granularity in-place 1-to-0 updates. As mentioned above, flash-based solid-state disks currently provide disk-like APIs, with pages or sectors as the unit of data transfer. Nevertheless, future devices may provide finer-grained APIs if there is a potential performance improvement. The results of this paper are a step in this direction, showing what is possible with such an API.

Some flash devices store error-correcting codes in reserved portions of the flash memory. Incremental changes to pages would also require incremental changes to the error-correcting codes. Even if the data changes are monotonic 1-to-0 writes, the resulting error-correcting code changes are unlikely to be monotonic. It may thus be necessary to reserve space for an array of error-correcting code values, and to write a new element into the array after each write.

While our EWOM model is motivated by flash memory, it is also possible that other technologies such as PRAM memory may, in the future, have similar block erase characteristics.

# 8. CONCLUSIONS

We have described a new model for measuring the performance of algorithms on write-once devices with an erase capability. We have adapted several standard algorithms to take account of the high page-write cost of arbitrary updates, and have analyzed their performance.

The results of this paper are unlikely to represent the final word on how to implement even the few techniques we have addressed. For example, it may be possible to trade space for time (or write performance for read performance) in different ways to get new algorithmic variants.


## Acknowledgements

Thanks to Bishwaranjan Bhattacharjee, Christian Lang, Bruce Lindsay, Tim Malkemus, George Mihaila, Haixun Wang, and Mark Wegman for helpful discussions and suggestions.


# 9. REFERENCES


[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[2] Paul England and Marcus Peinado. System and method for implementing a counter, 2006. US Patent Number 7,065,607.

[3] T. Gleixner, F. Haverkamp, and A. Bityutskiy. *UBI - Unsorted Block Images*, 2006.

[4] Toshiba Inc. NAND vs. NOR flash memory, 2006. Downloaded May 2008 from http://www.toshiba.com/taec/components/Generic/Memory_Resources/NANDvsNOR.pdf.

[5] Intel Corp. *Understanding the Flash Translation Layer (FTL) Specification*, 1998.

[6] T. H. Kuo et al. Design of 90nm 1Gb ORNAND flash memory with MirrorBit technology. In *Symposium on VLSI Circuits, Digest of Technical Papers*, pages 114–115, 2006.

[7] Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 55–66, New York, NY, USA, 2007. ACM.

[8] David Maier. Using write-once memory for database storage. In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 239–246, New York, NY, USA, 1982. ACM.

[9] Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for nand flash. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 410–419, New York, NY, USA, 2007. ACM.

[10] Ronald L. Rivest and Adi Shamir. How to reuse a write-once memory (preliminary version). In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 105–113, New York, NY, USA, 1982. ACM.

[11] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient B-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Sys.*, 6(3):19, 2007.


[4] `www.onfi.org`

# Fast Scans and Joins using Flash Drives

Mehul A. Shah Stavros Harizopoulos Janet L. Wiener Goetz Graefe

HP Labs
{firstname.lastname}@hp.com

## ABSTRACT

As access times to main memory and disks continue to diverge, faster non-volatile storage technologies become more attractive for speeding up data analysis applications. NAND flash is one such promising substitute for disks. Flash offers faster random reads than disk, consumes less power than disk, and is cheaper than DRAM. In this paper, we investigate alternative data layouts and join algorithms suited for systems that use flash drives as the non-volatile store.

All of our techniques take advantage of the fast random reads of flash. We convert traditional sequential I/O algorithms to ones that use a mixture of sequential and random I/O to process less data in less time. Our measurements on commodity flash drives show that a column-major layout of data pages is faster than a traditional row-based layout for simple scans. We present a new join algorithm, *RARE-join*, designed for a column-based page layout on flash and compare it to a traditional hash join algorithm. Our analysis shows that RARE-join is superior in many practical cases: when join selectivities are small and only a few columns are projected in the join result.

## 1. INTRODUCTION

With the ever increasing disparity between main memory and disk access times, enterprise applications are hungering for a faster non-volatile store. In this paper, we explore how to leverage one such promising technology, flash drives, for data analysis applications.

Driven by the consumer electronics industry, flash is becoming a practical non-volatile storage technology. Flash drives are ubiquitous in cameras, cell-phones, and PDAs. Major PC vendors are shipping laptops with flash drives. Moreover, flash is starting to make its way into the enterprise market. For example, vendors such as SimpleTech, Mtron, and FusionIO are selling flash-based solid-state drives aimed at replacing SCSI drives and entire disk arrays. But, are flash drives an effective replacement for traditional disks?

Flash drives have several traits that make them attractive for read-mostly enterprise applications such as web-page serving and search. Table 1 compares flash drives to disks. Flash drives offer more random read I/Os per second (1500 to 100,000 IO/s), offer comparable sequential bandwidth (20-80 MB/s), and use a tenth of the power (0.5 W). Flash is cheaper than DRAM (∼$18/GB) and is non-volatile. Moreover, flash continues to get faster, cheaper, and denser at a rapid pace. In particular, NAND flash density has doubled every year since 1999 [13].

Unfortunately, flash offers little or no benefit when used as a simple drop-in replacement for disk for data analysis workloads in databases. Traditional query processing algorithms for data analysis are tuned for disks; they stress sequential I/O and avoid random I/O whenever possible. Thus, they fail to take advantage of the fast random reads of flash drives.

In this paper, we investigate query processing methods that are better suited for the characteristics of flash drives. In particular, we focus on speeding up scan (projection) and join operations over tables stored on flash. Our algorithms use a mixture of random reads and sequential I/O. When only a fraction of the input (rows and columns) are needed, these algorithms leverage the fast random reads of flash to retrieve and process less data and thereby improve performance.

To make scans and projections faster, we examine a PAX-based page layout [2], which arranges rows within a page in column-major order. When only a few columns are projected, this layout avoids transferring most of the data while incurring the cost of "random" I/Os to seek between different columns. We explore the tradeoff between row-based and PAX-based layouts on flash experimentally. Our results show that a PAX-based layout is as good or better even at a relatively small page size of 64KB, a size that works well with traditional buffer management.

We then present a new join algorithm, called RARE-join (RAndom Read Efficient Join), that leverages the PAX-based layout. RARE-join first constructs a join index and then retrieves only the pages and columns needed for computing the join result. We show both analytically and using times from our scan experiments that this join outperforms traditional hash-based joins in many practical cases: when join selectivities are small and only a few columns are projected in the join result [6]. Although the specific methods we leverage in our algorithms have been previously studied, motivating their use and applying them in the context of flash storage is our main contribution.

In the next section, we give an overview of flash tech-


Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008), June 13, 2008, Vancouver, Canada.

| | NATA Disk | USB Flash | IDE Flash | FC Flash |
|---|---|---|---|---|
| **GB** | 500 | 4 | 32 | 146 |
| **$/GB** | $0.20 | $5.00 | $15.62 | |
| **Watts (W)** | 13 | 0.5 | 0.5 | 8.4 |
| **seq. read (MB/s)** | 60 | 26 | 28 | 92 |
| **seq. write (MB/s)** | 55 | 20 | 24 | 108 |
| **ran. read (IO/s)** | 120 | 1,500 | 2,500 | 54,000 |
| **ran. write (IO/s)** | 120 | 40 | 20 | 15,000 |
| **IO/s/$** | 1.2 | 75 | 5 | |
| **IO/s/W** | 9.2 | 3,000 | 5,000 | 6,430 |

**Table 1: Disk and Flash characteristics from manufacturer specs or as measured where possible. Prices from online retailers as of May 16, 2008. SATA-disk: Seagate Barracuda; USB-Flash: Buffalo; IDE-Flash: Samsung 2.5" IDE; FC-Flash: STech's ZeusIOps 3.5" FibreChannel.**

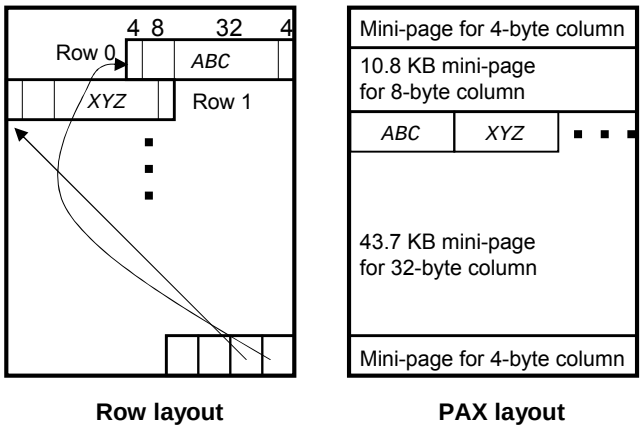



**Figure 1: Row and PAX page layout**

nology. Section 3 describes our experiments with scans and projections. Section 4 describes our join algorithm and compares it to traditional join methods. In section 5, we present the related work and then we conclude in Section 6.

## 2. FLASH CHARACTERISTICS

There are two types of flash available: NAND and NOR. NAND flash is typically used for data storage, and NOR is typically used in embedded devices as a substitute for programmable ROM. Since current solid state drives are typically composed of NAND flash, we focus on NAND.

Table 1 summarizes the relevant characteristics of current flash drives compared to disks.[1] Along with the conventional metrics, the table also lists the random I/O rate per dollar (IO/s/$), which measures the drive's price-performance, and the random I/O rate per Watt consumed (IO/s/W), which measures the drive's energy-efficiency. Although flash drives are more costly per gigabyte, they well outperform disk drives on metrics such as IO/s, IO/s/$ and IO/s/Watt.

NAND flash is typically organized into blocks of 128KB, which are the minimum erase units, and these blocks are subdivided into pages of 2KB. Once erased, all bits in the block are set to "1". Subsequently, selected bits can be programmed or set to "0" at a finer granularity. There is no generic rewrite option. Thus, unoptimized random writes are slow because they typically involve a read, erase (which is slow), and program.

Currently, most NAND flash is limited to about 100,000 erase-write cycles per block. To avoid premature failure, most flash drives include wear leveling logic that remaps writes to evenly update all blocks. With wear leveling, writing continuously to a 32GB drive at 40 MB/s would cause the drive to wear-out after 2.5 years. Since most drives are not fully utilized, this typically implies a lifespan of 5-10 years, which is acceptable.

## 3. SCANS AND PROJECTIONS

Relational scans and projections return some of the columns for some of the rows in a table. Since "seeks" are relatively cheap on flash, it can be cost-effective to introduce additional seeks instead of reading data not needed in the query. In this section, we consider the PAX page layout, which is efficient for reading one column at a time but requires a seek to skip over columns.

In Section 3.1, we describe the PAX page layout, discuss the drawbacks for using PAX to reduce disk I/O, and show why it is suitable for flash. In Section 3.2 we present our implementation and experimental results that verify the benefits of our approach.

### 3.1 PAX on Flash

Most commercial relational DBMS use a row-based page format where entire rows are stored contiguously, as shown on the left side of Figure 1. A slot array of 2-byte slots at the end of the page contains pointers to the start of each row. In this example, the table has four columns of sizes 4, 8, 32, and 4 bytes; the row size is 48 bytes. The page size is 64 KB. A full page contains 64 KB / (48+2) bytes = 1310 rows. We ignore page headers here for simplicity.

In contrast, the PAX (partition attributes across) layout [2] creates *mini-pages* within each page. The rows of the page are vertically partitioned among the mini-pages. Each mini-page stores data for a single column. Each mini-page for a fixed-length column is an array of column values with an entry for every row on that page; the $i$th entry on each mini-page is the column value for row $i$. Mini-pages for variable-length columns use slot arrays. The right side of Figure 1 shows this layout for the same example table. The 4-byte columns get 64 KB * (4/48) = 5460 byte mini-pages; the 32-byte column's mini-page is 43,680 bytes. Since no slot arrays are needed for these fixed length columns, the page with PAX layout holds data for 1365 rows.

In the original PAX proposal, Ailamaki et al. argued for a PAX layout to improve CPU cache utilization when scanning a subset of the columns [2]. They did not, however, consider a change to disk I/O access patterns. Here, we consider how a PAX layout can be used with flash to reduce total data transferred and thereby improve performance for scans.

With the row-based layout, a scan query that needs only a subset of the table columns must retrieve all of the columns of the table. With the PAX layout, however, a scan query can read only the required columns by "seeking" to the next column's mini-page (when the columns are not adjacent).

[1]Although we do not quote a price for the ZeusIOps drive, enterprise flash drives like this one are significantly more expensive in terms of $/GB.

When the time spent seeking from one mini-page to the next is less than the time to read the unneeded mini-pages between them, performing the random read (seek) is better.

Enterprise disk drives can read sequential pages at around 100 MB/s and a short seek takes about 3-4 ms. Therefore, a seek to skip mini-pages on disk must skip at least 300-400 KB (100 MB/s × 3-4 ms) to be worthwhile. If mini-pages are 300 KB, then full pages must be multiple MB. However, the "right" page size in a relational DBMS reflects many other factors, such as buffer pool size, buffer-cache hit ratios, update frequency, and per-page algorithmic overheads. Unfortunately, these and other economic considerations [4] lead most commercial RDBMSs to use much smaller page sizes, typically between 8 KB and 64 KB.

Although the PAX layout does not improve read performance for disks, it is worthwhile on flash. The seek overhead on flash is much smaller. For the IDE flash drive in Table 1 with 28 MB/s sequential read bandwidth and 0.25 ms seek time, it makes sense to skip mini-pages of only 7 KB (28 MB/s × 0.25 ms). Full pages can therefore be only 32-128 KB. With a 32 KB (or larger) page, a scan query that projects less than three-quarters of a table will complete faster using a PAX layout. Since mini-pages are not always aligned at page-size boundaries of flash pages; some extra data will be read when bringing a single column into memory. In the next section, we verify these numbers experimentally.

Column stores also partition the table data by columns to allow fast access to a subset of the columns [14]. Our query processing methods apply to such column layouts as well as to PAX layouts on flash. However, a column layout has two important limitations when compared to PAX layout. First, many parts of a traditional database engine, such as the storage layer, I/O subsystem, buffer pool, recovery system, indexes, some operators, and so on, expect and operate on fixed size pages. Thus, a column layout requires touching all these components and effectively redesigning the engine. A PAX layout, on the other hand, only requires reimplementing the storage layer methods that retrieve data from a page, since only the page organization has changed from a traditional row layout, not the page contents. Second, column stores may require multiple I/Os to update multiple columns of a single row. With a PAX layout, only one I/O is needed, since all columns are stored on the same page. We therefore investigate performance with PAX layouts since they involve a less disruptive change.

## 3.2 Experiments

In our implementation, we use tightly packed mini-pages. We read data in multiples of 4 KB, even if the needed mini-page is less than 4KB. For our experiments, we modified the code released in [6], which is a bare-bones high-performance storage manager. This code implements relational scanners on a single-threaded C++ code base, using Linux's Asynchronous I/O capabilities to issue multiple outstanding I/O requests and overlap computation with disk transfer time.

We evaluate a single scan query of the form "select O1, O2, .. from ORDERS where predicate(O1)" in which the predicate yielded 10% selectivity. The ORDERS table is loosely modeled after TPC-H data. We simplify the schema: the table contains eight 4-byte integers for a row length of 32 bytes; we create 60 million rows, for a total table size of about 2 GB. We ran the experiments on the Samsung

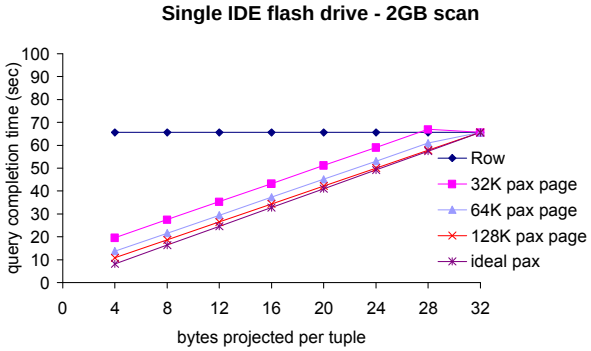



**Figure 2: Comparing row vs. PAX layouts for scans and projections.**

32 GB IDE flash drive (from Table 1) formatted with the Linux ext3 file-system. The baseline case is a scan of the entire table, which corresponds to the performance of a row store.

Figure 2 compares the performance of scans as we vary the number of columns projected for three different page sizes: 32 KB, 64 KB, and 128 KB. For all page sizes, the fewer columns the query projects, the better it performs. The "ideal" curve assumes the true read bandwidth of the IDE drive and no "seek" delays. As we increase the page size (and thus each mini-page read is larger), the overhead of seeking is amortized. PAX pages of 32 KB are as good as the baseline (row layout) case no matter how many columns are read, and better when up to 88% of the data are read. PAX pages of 64 KB always outperform the baseline case and come close to ideal performance.

# 4. JOINS

In this section, we show how to leverage the PAX layout to compute joins. We analytically compare a traditional Grace-hash join to a new join algorithm, RARE-join, that incurs additional random I/Os to save total data accessed.

## 4.1 RARE-join

Our new join algorithm, called the RARE-join (RAndom Read Efficient join), has two main conceptual steps.

- It computes a join index by accessing only the join columns of the input tables.
- It adapts Li and Ross' jive-join [10] for PAX layouts on flash. Jive-join uses a join index to compute the join result in a single read pass through the input.

The main idea is to save I/Os by accessing only the join columns and mini-pages holding the values needed in the result rather than the entire input. The savings comes at the cost of increased "seeks" and computing the join index, which we show can be worthwhile when using flash.

To describe RARE-join and illustrate its benefits, we compare it with a well-known hash-based join: Grace-hash [8]. Traditional Grace-hash operates over a row layout. We also include a simple variant, Grace-PAX, which operates over a PAX layout; the underlying scans access only the mini-pages for columns projected in the result. We analyze two basic modes for these joins: when there is enough memory

to compute the join in one pass through the input, and when more passes are needed. We also analyze an important, in-between mode for RARE-join: when it makes one pass over the input but must materialize the join index.

For each algorithm, we assume that I/O cost dominates runtime and give its costs in terms of the number of I/Os required. We assume that the costs of a sequential read, a sequential write, and a random read are the same. To correct for this simplifying assumption, we adjust the I/O costs in our examples using measurements from Section 3 where possible and appropriate. Table 2 shows the notation that we use for the pseudocode and cost equations.

| Symbol | Meaning |
|---|---|
| $T_1$ | Table 1 |
| $R$ | Join result |
| $J_1$ | Join column of $T_1$ |
| $V_1$ | Remaining columns of $T_1$ projected in result |
| $id_2$ | Row-id of join result from Table 2 |
| $I_2$ | Temp file filled with $id_2$ |
| $JI$ | Temp file holding join index |
| $M$ | Memory available for join |
| $h$ | Hash-table overhead |
| $\sigma_{p1}$ | Fraction of $T_1$ pages needed for computing the join |

**Table 2: Notation used for cost equations. $T_2$ is always the smaller table and its symbols are analogous to those for $T_1$. The $|X|$ notation specifies the number of page I/Os for $X$.**

## 4.2 One-pass joins

Grace hash can compute the result in one simple pass over the input if a hash-table on $T_2$, the smaller table, can fit in memory, i.e. $h|T_2| < |M|$, It first reads and builds a hash-table on $T_2$. Then it reads $T_1$, probes the hash-table, and spools the results to $R$. All accesses are sequential and the total I/O cost is simply:

$$|T_1| + |T_2| + |R| \tag{1}$$

Grace-PAX is better for two reasons. First, it needs less memory to operate in one pass because only the join and projection columns of $T_2$ must fit in memory, $h(|J_2|+|V_2|) < |M|$. Second, since it skips the unneeded columns, the total I/O cost is less:

$$|J_1 + V_1| + |J_2 + V_2| + |R| \tag{2}$$

In roughly the same memory as a 1-pass Grace-PAX, a 1-pass RARE-join, shown in Figure 3, reduces the I/O cost further. RARE-join reads and builds a hash-table on the join column, $J_2$, and row-id, $id_2$. Then, it probes the hash-table with $J_1$. Unlike Grace, it fetches only those mini-pages necessary to produce the join result. For all matches, RARE-join fetches and pins the mini-pages containing row $id_2$ from $V_2$ and fetches the mini-pages containing row $id_1$ from $V_1$. Since it scans in $T_1$ order, the new $V_1$ mini-pages can immediately replace old ones while the $V_2$ pages are buffered. RARE-join spools the result to $R$. More precisely, the memory requirement is: $h(|J_2| + |id_2|) + \sigma_{p2}|V_2| < |M|$, and the total I/O cost is:

1. Read $J_2$ and build hash-table
2. Read $J_1$ and probe hash-table

**foreach** *join result* <$id_1$, $id_2$> **do**
- Read projected values of row $id_1$ from $V_1$
- Read projected values of row $id_2$ from pinned $V_2$ mini-pages else from flash
- Write result into $R$

**Figure 3: 1-pass RARE-join: when the hash-table on $J_2$ and needed mini-pages of $V_2$ fit in memory.**

1. Read $J_2$ and build hash-table
2. Read $J_1$ and probe hash-table

**foreach** *join result* <$id_1$, $id_2$> **do**
- Read projected values of row $id_1$ from $V_1$
- `/* R and I2 are both partitioned by id2 */`
- Write projected values into partition of $R$
- Write $id_2$ into partition of $I_2$

3. Read $I_2$ and process it.

**foreach** *partition of* $I_2$ **do**
- **foreach** $id_2$ *in partition* **do**
  - Read projected values of row $id_2$ from $V_2$
  - Write values into partition of $R$

**Figure 4: $(1+\epsilon)$ pass RARE-join: when the hash-table on $J_2$ and output buffers fit in memory.**

$$|J_1| + \sigma_{p1}|V_1| + |J_2| + \sigma_{p2}|V_2| + |R| \tag{3}$$

Thus, given sufficient memory for the 1-pass case, RARE-join outperforms Grace-PAX which outperforms Grace in our cost model. In reality, however, the advantages depend upon the overheads for each I/O of mini-pages and the "page" selectivity. Depending on these parameters, we can adapt RARE-join to make it behave more like Grace-PAX: fetch $V_2$ with $J_2$ or $V_1$ with $J_1$ or both.

## 4.3 More than 1 pass

If there is not enough memory to hold the hash-table on $T_2$ for Grace or on $J_2$ and $V_2$ for Grace-PAX, both degrade into a two pass algorithm. The first pass partitions both tables on the join column such that the runs of the smaller table fit into memory. This pass involves a read and write. The second pass reads each partition into memory and computes the join. Thus, the total I/O cost for Grace is:

$$3(|T_1| + |T_2|) + |R| \tag{4}$$

and likewise for Grace-PAX:

$$3(|J_1 + V_1| + |J_2 + T_2|) + |R| \tag{5}$$

Most joins will need at most two passes with flash, since the outgoing buffer size can be small, e.g., 64 KB. With 2 GB of main memory, there is room to create 32,000 partitions. Therefore, a two-pass Grace join suffices for $T_2$ up to 65TB, which is much larger than the size of current flash drives.

### 4.3.1 $(1+\epsilon)$ pass RARE-join

RARE-join has more flexibility than Grace and thereby provides improved performance. If $J_2$ fits in memory, but

| Name | Address | Age | Team |
|---|---|---|---|
| Ben | 18 Main St | 7 | Orange |
| Julie | 21 Iris Ln | 8 | Red |
| Sam | 110 Hays Dr | 7 | Green |
| Sarah | 2 Main St | 7 | Blue |
| Alex | 90 Primrose | 8 | Red |
| Lena | 44 Madison | 7 | Orange |

**Figure 5: Player Table.**

| Team | Field | Time | Row Id |
|---|---|---|---|
| Red | Terman | 1 | 1 |
| Orange | Ohlone | 9 | 2 |
| Orange | Carmelo | 3 | 3 |
| Blue | Briones | 2 | 3 |

**Figure 6: Game Table.**

$V_2$ does not, RARE-join can still compute the result with one pass through the input, but must materialize the join index. This $(1+\epsilon)$ pass RARE-join is shown in Figure 4. To illustrate the algorithm, we step through it for the example join query: "select name, team, time from player, game where player.team = game.team;" using the Player and Game tables shown in Figures 5 and 6.

As in the 1-pass case, RARE-join first builds a hash-table on $J_2$ and probes it with $J_1$. Figure 7 shows the hash-table for our example.

The result of probing the table in step 2 is the join index, which has one entry for each row in the join result $R$. Since the probes occur in $J_1$ order, the necessary $V_1$ mini-pages can be read sequentially and written directly to the result $R$. Since we use a PAX layout, we write only the $V_1$ columns to $R$ and leave the portion of each page for $V_2$ columns "blank" until step 3. Note, this write pattern for $R$ is efficient on flash since we pay the cost of the erase in this phase and "program" the $V_2$ values in step 3.

Unlike in the 1-pass case, we cannot fit $V_2$ in memory. A simple approach is to read the needed $V_2$ mini-pages on demand. This approach, however, might retrieve the same mini-pages more than once since we generate results in $J_1$ order. Instead, like jive-join, we partition the join index into runs in step 2, so that the $V_2$ mini-pages referenced in each run can fit in memory. Actually, jive join [10] creates sorted runs of the join index, which it then merges so that it can later fetch the rows of $T_2$ sequentially. We borrow from this idea, but observe that, with flash, we need not access $V_2$ in sequential order. We only need to ensure that all values needed from a single page are obtained with one page read.

Therefore, in step 2, we simply partition the join index by the $T_2$ page number (encoded in $id_2$), so all row ids for the same $T_2$ pages go in the same partition. We need not materialize the $id_1$ column of the join index since $V_1$ is streamlined to the result in step 2. Thus, the $I_2$ partitions only contain $id_2$ values and are implicitly in $T_1$ order.

For step 3, the entire set of $V_2$ pages in a partition must fit in memory at once. The number of partitions needed is therefore $|V_2|/|M|$ and the partitioning function can be either a hash or range partitioning scheme. We partition $R$ the same way so that in step 3, we can fill in the blank parts of $R$ with corresponding $V_2$ values, one partition at a time. After step 3, we combine the pages from all partitions of $R$ into a single file simply by linking them together. Figures 8 and 9 show the contents of the partitions in our example and Figure 10 shows the final result table.

| Blue, 3 |
|---|
| Red, 1 |
| Orange, 2 → Orange, 3 |

**Figure 7: $(1+\epsilon)$ RARE: hash-table on Game.Team**

| | |
|---|---|
| 1 | 2 |
| 4 | 3 |
| 1 | 2 |
| | 3 |

**Figure 8: $(1+\epsilon)$ pass RARE, step 2: $I_2$ partitions.**

Since we need one buffer page for each partition of $R$ and $I_2$ in step 2 and there are at most $|V_2|/|M|$ partitions, the memory requirement is $h(|J_2|+|id_2|)+2(|V_2|/|M|) < |M|$. The total I/O cost of all three steps is:

$$|J_1| + \sigma_{p1}|V_1| + |J_2| + \sigma_{p2}|V_2| + |R| + 2|I_2| \quad (6)$$

Combining the previous two equations, RARE outperforms Grace-PAX when:

$$2|J_1| + (3-\sigma_{p1})|V_1| + 2|J_2| + (3-\sigma_{p2})|V_2| > 2|I_2| \quad (7)$$

The left hand side is savings from reading the join columns and only the needed mini-pages of $V_1$ and $V_2$ once instead of three times. This savings must outweigh the additional cost of materializing and reading the row-ids $id_2$ from the join index.

We illustrate the potential benefits of RARE-join with the following example. Suppose $T_1$ and $T_2$ each have 8 columns of 4 bytes and the join result contains only 3 columns from each, i.e. 5 total with the common join column. Let $T_1$ and $T_2$ contain 256 million rows (8 GB) apiece. Further suppose half the rows in $T_1$ each match one row in $T_2$ and the page selectivities are 1. Also, assume a system with 2 GB of memory. We can then estimate the savings using the performance numbers from Section 3 as follows.

Both $V_1$ and $V_2$, which hold two columns, are 2 GB, and $J_1$ and $J_2$ are 1 GB each. This setup puts Grace-PAX in the two-pass mode and RARE-join in the $(1+\epsilon)$ pass mode. Assuming row-ids are 4 bytes, $R$ and $I_2$ each will have 128M rows and be 2.6 GB and 512 MB, respectively. Assuming we use 64 KB pages, reading $J_1$ (one column) takes 55.1 s, reading $V_1$ (two columns) takes 86.5 s, and reading $J_1+V_1$ (three columns) takes 117.8 sec; the transfer times are the same for $J_2$ and $V_2$. Note, these account for the mini-page "seek" overheads as measured in Section 3. We estimate that writing $R$ takes 91.4 s and one pass through $I_2$ takes 18.3 s. Therefore, Grace-PAX will take $3(117.8\times 2)+91.4 = 798.2$ s while RARE-join will take $2(55.1+86.5)+91.4+2*18.3 = 411.2$ s, a savings of 387 s or speedup of 1.94x. The savings from making only a single pass through the input is 423 s, and the penalty for reading and writing the join index is only 36.6 s.

### 4.3.2 Two-pass RARE-join

Figure 11 shows the pseudocode for RARE-join when $J_2$ and the outgoing buffers do not fit in memory. In this case,

| | | |
|---|---|---|
| Julie | Red | |
| Sarah | Blue | |
| Alex | Red | |

| | | |
|---|---|---|
| Ben | Orange | |
| Ben | Orange | |
| Lena | Orange | |
| Lena | Orange | |

**Figure 9: $(1+\epsilon)$ pass RARE, step 2: partitioned $R$**

| | | |
|---|---|---|
| Julie | Red | 1 |
| Sarah | Blue | 2 |
| Alex | Red | 1 |
| Ben | Orange | 9 |
| Ben | Orange | 3 |
| Lena | Orange | 9 |
| Lena | Orange | 3 |

**Figure 10: $(1+\epsilon)$ pass RARE, end: Result, $R$**

steps 1 and 2 are similar to those in Grace-hash join. RARE-join hash partitions the join column of both tables so that each $J_2$ partition can fit in memory. In step 3, it computes and materializes the join index $< id_1, id_2 >$ for each partition. Note that within each partition, the join index is ordered by $id_1$.

In step 4, RARE-join merges the partitions of $JI$ into $T_1$ order and fetches the needed projection columns $V_1$. It spools the result and $id_2$ values to partitions of $R$ and $I_2$, as in the $(1+\epsilon)$ pass algorithm. Then, step 5 is the same as step 3 of the $(1+\epsilon)$ RARE algorithm. Note, the join index is exactly twice the size of $I_2$, since it contains $id_1$ and $id_2$. Again, RARE-join fetches only the needed mini-pages of $V_1$ and $V_2$, but makes multiple passes over the join columns. The total cost is therefore

$$3|J_1| + \sigma_{p1}|V_1| + 3|J_2| + \sigma_{p2}|V_2| + |R| + 6|I_2|. \quad (8)$$

Two-pass RARE-join therefore beats two-pass Grace-PAX when

$$(3 - \sigma_{p1})|V_1| + (3 - \sigma_{p2})|V_2| > 6|I_2| \quad (9)$$

The left hand side is the savings from only accessing the needed pages of $V_1$ and $V_2$ once instead of thrice. The right hand side is the penalty for materializing and passing over the join index multiple times.

We again illustrate the savings from RARE-join with an example. Suppose $T_1$ and $T_2$ have the same schema as in the previous example but are four times larger, 32 GB each. Also, consider the same join query as above with the same row and page selectivities. In this case, $J_1$ and $J_2$ are 4 GB each, and $V_1$ and $V_2$ are 8 GB each. These input sizes place both RARE-join and Grace-PAX in the two-pass mode. The result has 512M rows, with 5 attributes of 4 bytes each. Thus, $R$ is 10.2 GB and $I_2$ is 2 GB. Assuming the same performance as above, reading $J_1$ (one column) takes 220 s, reading $V_1$ (two columns) takes 346 s, and reading $J_1 + V_1$ (three columns) takes 471.2 s. We estimate writing $R$ takes 366 s, and one pass through $I_2$ takes 73.1 s. Therefore, Grace-PAX will take $3(471.2 \times 2) + 366 = 3193$ s while RARE-join will take $3(220\times2)+(346\times2)+366+(6\times73.1) = 2817$ s, a savings of 376 s or speedup of 1.12x. The penalty for I/O on the join index was 439 s, but the savings from making only one pass through the projected columns was 815 s. A more selective query would only improve the RARE-join performance relative to Grace-PAX.

1. Read $J_2$ and partition it (hash on join value)
2. Read $J_1$ and partition it (same hash function)
3. Compute $JI$

**foreach** *partition of* $J_2$ **do**
- Read $J_2$ and build hash-table
- Read partition of $J_1$ and probe hash-table
- **foreach** *row in join result* **do**
  - Write $id_1, id_2$ in $JI$ partition

4. Merge partitions of $JI$ on $id_1$

**foreach** *join result* $<id_1, id_2>$ **do**
- Read projected values of row $id_1$ from $V_1$
- `/* R and I2 are partitioned by id2 */`
- Write projected values into partition of $R$
- Write $id_2$ into partition of $I_2$

5. Read $I_2$ and process it.

**foreach** *partition of* $I_2$ **do**
- **foreach** $id_2$ *in partition* **do**
  - Read projected values of row $id_2$ from $V_2$
  - Write values into partition of $R$

**Figure 11: Two-pass RARE-join: when the hash-table on $J_2$ and output buffers do not fit in memory.**

Extending RARE-join for more passes is analogous to extending Grace-hash, so we omit the description here.

## 4.4 Discussion

Although we believe our cost model is sufficient to highlight the potential benefits of RARE-join, we still need to implement and measure its benefits. We need to measure its true performance and map out the tradeoffs in comparison to Grace, Grace-PAX, and more sophisticated variants of hash-based joins. There are a number of complicating factors that might affect performance. For example, our analysis ignores CPU costs, underestimates I/O overheads, and ignores the fact that sequential writes on flash are slower than sequential reads.

A disadvantage of RARE-join, similar to jive-join, is that the join results must be materialized. For some data analysis functions, such as computing materialized views, this is not an issue. However, when used in a pipelined query plan, the above comparison is unfair. In that case, we need to penalize RARE-join with the cost for reading and writing $R$. Even so, RARE-join can be more efficient if the join result size or selectivity is sufficiently small.

Nonetheless, there are still opportunities to improve RARE-join. The hash-table on the join column could use compression for duplicate values. We could modify the algorithm to pipeline results better at the cost of additional I/Os. As Hybrid hash join does, we could potentially use available memory more effectively on the first pass through the data. We would also like to consider adapting other join algorithms, such as index-nested loops join and sort-merge join, for PAX layouts on flash.

# 5. RELATED WORK

We briefly review recent work on using flash in databases. Graefe [4] revisits the five-minute rule in the context of flash and suggests that flash serve as the middle level of a 3-level

memory hierarchy. Given current technology, this analysis shows that 32KB is too small for pages stored on a SATA disk and fairly large for pages stored on NAND flash. He lists several potential uses for flash, some which treat flash as an extension of memory and others that treat it as a faster disk. Graefe [5] also considers sorting over flash, although that paper is primarily concerned with improving memory utilization and robustness rather than with improving sort performance. In contrast, we focus solely on query processing over flash. For the future, we should consider adapting our methods to a 3-level hierarchy.

Lee and Moon [9] also present new variants of standard database algorithms that are adapted for the characteristics of flash. They consider techniques for updating rows in pages on flash. To avoid random writes, their approach logs updates to database pages in a clean "log" section at the end of each flash erase block rather than applying the updates in place. Once the log section is exhausted, they relocate the entire erase block and apply the updates. This approach amortizes the cost of the erase over multiple updates.

Next, we outline previous ideas that we adapted for query processing on flash drives. Ailamaki et al. [2] proposed the PAX database page design to improve the cache performance of TPC-H queries rather than to save on disk I/O. Reading only the relevant columns for each query is the central theme of column-oriented DBMSs such as C-Store and MonetDB [3, 14]. These systems reportedly perform well on certain types of queries [1, 6], using traditional disk drives. For example, Harizopoulos et al. [6] show that a carefully designed column store can out-perform a row-store for read-mostly workloads. Further, Abadi et al. [1] look at join processing over column layouts. As mentioned earlier, we can easily apply our algorithms to column stores on flash and provide similar benefits as with PAX. We, however, focus on a PAX layout since it imposes less disruptive changes to traditional database architectures. Moving to a flash storage and using the PAX page layout blurs the line between column-stores and row-stores. Like us, Zhou and Ross [15] use a scheme similar to PAX, called MBSM, that co-locates column values in blocks within larger "super-blocks" to reduce I/O. They optimize their methods, however, for traditional disks rather than flash.

Li and Ross [10] present efficient join algorithms, jive-join and slam-join, that leverage a join index and stores the results in a column-oriented format. We modify the jive-join by streamlining it with join index creation and by avoiding the unnecessary steps used to optimize disk accesses. To make disk I/Os sequential, jive-join sorts the join index before fetching the matching pages from the inner table and re-orders the returned tuples to match the order of the outer. Although this difference does not affect total data transferred, it introduces additional CPU overheads which can be important.

Some have also explored the energy-efficiency benefits of flash. Rivoire et al. [11, 12] show that using flash can improve the energy-efficiency of database operations like sort. Kgil and Mudge [7] employ flash for a buffer cache for web-servers to reduce their energy use.

# 6. CONCLUSION

In this paper, we present techniques for making core query processing operations, i.e. scans and joins, faster when using flash. Our techniques rely on using a PAX-based page layout, which allows scans to avoid reading columns not needed for the query. A PAX layout works well for flash drives since they offer much shorter seek times than traditional disks. We then present a join algorithm, RARE-join, that leverages the PAX structure to read only the columns needed to compute the join result. Roughly speaking, RARE-join first computes a join index by retrieving the join-columns and then fetches the remaining columns for the result. We show that RARE-join using a PAX layout beats traditional hash-based joins when few columns are returned and the selectivity is low.

Several directions suggest themselves for future work. Obviously, additional measurements on new hardware is an ongoing task. We also plan on studying scan and join performance on flash for generalized vertical partitioning, e.g., storing first name and last name together rather than separately. In addition, we plan on investigating merits and issues of RARE-join in complex query execution plans, e.g., pipelining and scheduling in bushy plans, memory management, and materialization of intermediate results. Finally, in addition to strict performance metrics, we plan on reviewing the new techniques with respect to energy efficiency as well as robustness of performance under adverse run-time conditions, e.g., errors in cardinality estimation, distribution and duplicate skew, and memory contention. We expect that steps in these directions will speed the eventual adoption of flash in enterprise systems.

# 7. ACKNOWLEDGMENTS

We thank Jennifer Burge for her initial experiments that illuminated the benefits of flash for database workloads.

# 8. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? *SIGMOD*, 2008.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. *VLDB*, pages 169–180, 2001.

[3] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. *CIDR*, 2005.

[4] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. *DaMoN*, 2007.

[5] G. Graefe. Sorting with flash memory. *Unpublished manuscript.*, 2008.

[6] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. *VLDB*, pages 487–498, 2006.

[7] T. Kgil and T. N. Mudge. Flashcache: a nand flash memory file cache for low power web servers. *CASES*, pages 103–112, 2006.

[8] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[9] S.-W. Lee and B. Moon. Design of flash-based dbms: an in-page logging approach. *SIGMOD*, pages 55–66, 2007.

[10] Z. Li and K. A. Ross. Fast joins using join indices. *VLDB J.*, pages 1–24, 1999.

[11] S. Rivoire, M. A. Shah, P. Ranganathan, and

C. Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. *SIGMOD*, pages 365–376, 2007.

[12] S. Rivoire, M. A. Shah, P. Ranganathan, C. Kozyrakis, and J. Meza. Models and metrics to enable energy-efficiency optimizations. *IEEE Computer*, 40:39–48, Dec. 2007.

[13] Samsung. Samsung Semiconductor Products. Online. http://www.samsung.com/global/business/ semiconductor/products/flash/Products_NANDFlash.html.

[14] M. Stonebraker et al. C-store: A column-oriented dbms. *VLDB*, pages 553–564, 2005.

[15] J. Zhou and K. A. Ross. A multi-resolution block storage model for database design. *IDEAS*, July 2003.

# Data Partitioning on Chip Multiprocessors

John Cieslewicz[*] [†]
Columbia University
New York, NY
johnc@cs.columbia.edu

Kenneth A. Ross[†]
Columbia University
New York, NY
kar@cs.columbia.edu

## ABSTRACT

Partitioning is a key database task. In this paper we explore partitioning performance on a chip multiprocessor (CMP) that provides a relatively high degree of on-chip thread-level parallelism. It is therefore important to implement the partitioning algorithm to take advantage of the CMP's parallel execution resources. We identify the coordination of writing partition output as the main challenge in a parallel partitioning implementation and evaluate four techniques for enabling parallel partitioning. We confirm previous work in single threaded partitioning that finds L2 cache misses and translation lookaside buffer misses to be important performance issues, but we now add the management of concurrent threads to this analysis.

## 1. INTRODUCTION

Partitioning is a core database task used for many purposes. Partitioning can divide a larger task into constituent smaller subtasks that can be processed more quickly than the overall task taken as a whole. An example of this is when a complete task, done all at once, would not be entirely cache resident and could therefore suffer from a high number of cache misses. Smaller subtasks may fit within a cache, thereby experiencing good cache performance. Partitioning is also important because of its ability to group like values. In the case of a hash join, both relations are partitioned using the same hash function and only tuples from equivalent partitions can join to form part of the result. Similarly, range partitioning based on a sort key can improve the performance of sorting, which itself is a core database operation. In the context of parallelism, partitioning assists load balancing by producing similarly sized subtasks.

A chip multiprocessor (CMP) is a single chip that supports multiple concurrent threads of execution with multiple processor cores per chip, multiple threads per core, or both. This paper uses a specific CMP, the Sun UltraSPARC T1, which has eight cores and four threads per core for a total of 32 threads on a single chip [11].

In this paper we examine in-memory hash-based partitioning performance on a chip multiprocessor. Though other types of partitioning (such as range partitioning) are common, we identify the process of writing partitioning output, rather than the method of computing a tuple's partition assignment, to be key to partitioning performance. Therefore, we focus on hash-based partitioning and explore different means of coordinating the writing of output.

High performance partitioning on a CMP requires balancing parallelism with interthread coordination and on-chip resource sharing. Because partitioning requires writing output to many different locations, shared cache and translation lookaside buffer (TLB) resources can become a source of contention between threads, causing lower performance. Cache and TLB pressure is a problem for single-threaded partitioning implementations [8]. The problem is compounded by the presence of multiple concurrent threads for two reasons. First, each additional thread may increase the number of output locations that are active at any one time. And second, the interthread coordination required to effectively manage the shared on-chip resources can become a bottleneck itself. In this paper we will describe the impact of these issues and present techniques that overcome them to achieve high performance partitioning on CMPs.

The rest of the paper is organized as follows. In Section 2 we will present work related to both partitioning and database operations on chip multiprocessors. Section 3 describes partitioning techniques and implementation options. Our experimental platform and setup is described in Section 4 and experiments are presented in Section 5. We present future work in Section 6 and conclude in Section 7.

## 2. RELATED WORK

Partitioning has been studied in a number of contexts, both parallel and not. It is central to many database operations, including joins and aggregates, and it is also important for load balancing [4, 3, 8, 10]. Many variants of parallel sorting include an initial partitioning step [5, 6]. As parallelism is now available on-chip, we revisit partitioning to investigate techniques that provide the best parallel partitioning performance on new CMPs.

Manegold et al. introduce a clustering (partitioning) algorithm that performs well on modern architectures by optimizing cache and TLB usage [8]. Though their analysis fo-

[*]Supported by a U.S. Department of Homeland Security Graduate Research Fellowship

[†]Supported by NSF grant IIS-0534389


*Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008)* June 13, 2008, Vancouver, Canada.

| Technique | Contiguous Output | Contention |
|---|---|---|
| Independent | No | No |
| Concurrent | Yes, per partition | Yes |
| Count-then-move | Yes | No |
| Parallel Buffers | Mostly per partition | No |

**Table 1: CMP Partitioning Techniques**

cused on single threaded execution, it provides an excellent starting point for exploring the issues associated with in-memory parallel partitioning on chip multiprocessors. Our work differs in two main ways. First, our goal is high performance parallel partitioning on a CMP rather than a uniprocessor. And second, our target platform uses very simple cores optimized for thread-level parallelism rather than single threaded performance.

A motivation for examining parallel partitioning arose from observations made during an analysis of aggregation on CMPs where it was found that contention between threads for shared resources as well as interthread communication were key factors affecting performance [2]. Because locality of reference is important to hash aggregation performance, parallel partitioning may be able to improve aggregation throughput by clustering the input. This paper explores the characteristics of such a parallel partitioning. As mentioned in Section 1, coordinating the writing of output between threads is an important aspect of a parallel partitioning implementation. The parallel buffer data structure proposed in [1] provides a framework for parallel, contention free access to a mostly contiguous shared buffer. Using the parallel buffer structure is one of the options for implementing parallel partitioning described next in Section 3.

# 3. PARTITIONING TECHNIQUES

Partitioning data is a two step process. In the first step, the output partition to which a tuple belongs is identified. In the second step, the tuple is copied to its output location. The choice of the partitioning function in the first step depends on the properties desired from the partitioning. In this paper we focus on hash partitioning. Identifying a tuple's partition using hashing is an $O(1)$ operation. We will use multiplicative hashing and describe the number of output partitions in terms of $b$, the number of hash bits used to partition the input. The use of $b$ hash bits results in $2^b$ output partitions. Because the input is read only, it can easily be divided among $t$ threads for parallelism. Writing the input to different partitions, however, involves trade-offs in thread coordination and resource sharing. We examine four methods of enabling parallelism in the writing of tuples to output partitions. The methods are summarized in Table 1 and described below.

## 3.1 Independent Output

In the *independent output* technique, each thread has its own output buffers for each output partition, i.e. $t * 2^b$ output buffers. There is no sharing of output space between threads and therefore no thread coordination required aside from assigning input tuples to each thread. Each buffer requires the storage of metadata, such as the current writing index and the size of the buffer. As the number of threads or hash bits increases, the number of buffers required also increases. At the same time, the expected size of each buffer, $\frac{N}{t*2^b}$, decreases[1], which means that the storage overhead associated with metadata increases.

The complete independence of each thread helps enable parallelism by avoiding any contention between threads. Unintentional *false sharing* in the cache can be avoided by ensuring that each thread's buffer metadata does not share a cache line with the metadata from another thread.

There are two main disadvantages of this approach. First, the metadata overhead increases as additional threads or partitions are used. And second, each partition is fragmented into $t$ separate buffers. The operator that next processes the partition must either accept fragmented input or a further consolidation step is needed.

## 3.2 Concurrent Output

The *concurrent output* technique uses a single buffer for each output partition. The one buffer is shared among all threads, which coordinate writing through the use of atomic instructions or locks. Specifically, the current writing index must be atomically incremented before each write. In contrast to the independent technique, the number of output buffers no longer depends on the number of threads, $t$. In terms of storage overhead, therefore, the concurrent output method scales better as more threads are used. Also, using this method, each partition's output is stored contiguously, which may make further processing of a partition easier.

The atomic instructions or locks required for the correctness of this technique are expensive and susceptible to contention. In the independent approach, incrementing a write index takes one cycle, whereas performing the atomic increment required for the concurrent technique takes 22 cycles[2]on the Sun T1 [12]. Not only do atomic instructions have longer latency, but they can also cause contention. When many threads attempt to atomically increment the same variable, only one can proceed, forcing all other threads to wait. In the worst case, all threads serialize due to atomic increments to the same variable(s), which severely degrades performance because no parallel computation occurs.

## 3.3 Count-Then-Move

The *count-then-move* technique uses two passes over the input to partition the data into a single contiguous buffer in which consecutive partitions reside within consecutive ranges in the buffer. In the first pass, each thread processes an assigned range of input tuples, counting the number of tuples it would have placed in each partition. This step requires $2^b$ counters per thread. Each thread works independently, so a high degree of parallelism is possible. As with the independent technique above, care should be taken to avoid false sharing in the cache by ensuring that no two threads' counters share a cache line.

Following the first pass, all of the threads must synchronize to signal that counting is complete. Using the counts supplied by all threads, each thread can then compute the exact offset at which it will start writing output for each partition. Each thread must therefore store $2^b$ writing offsets.

[1]Assuming that the partitioning keys are unique and uniformly distributed.

[2]The Sun UltraSPARC T1 may not be used in an SMP configuration as it does not support off-chip cache coherency and atomic operations. This atomic latency would be higher in a comparable processor that supported off-chip cache coherency and was used in an SMP configuration.

Once these offsets are computed, the second pass begins. Each thread processes the same assigned range of tuples from the first pass, but this time, once a tuple's assigned partition is determined, it is written using the thread's offset for that partition. The offset is then incremented. The second pass also requires no interthread coordination.

A drawback of this approach is that it requires two passes over the input to accomplish what other techniques accomplish in one pass. On the other hand, the entire result is contiguous, which may be more useful than the fragmented results produced by the other methods. The other methods require further processing to produce contiguous output.

### 3.4 Parallel Buffers

The partitioning technique using *parallel buffers* [1] is very similar to the concurrent output approach except that instead of coordinating between threads on every write, coordination occurs at the coarser granularity of a "chunk" of tuples. Each thread atomically obtains a contiguous chunk of one or more tuples from the buffer. It then has exclusive access to those tuples and only needs to engage in interthread coordination when it acquires a new chunk. The cost of atomic operations is amortized over many writes and the chance of contention for shared data structures is reduced. An appropriately sized chunk can eliminate all coordination contention, as shown in [1]. The parallel buffer is designed so that all but the first $t$ chunks are either completely full or completely empty and that there are no holes, i.e., all data not in the first $t$ chunks is contiguous.

Using parallel buffers has the advantage of avoiding interthread contention while using a mostly contiguous shared buffer. Output written into a parallel buffer is also ready to be read in parallel by subsequent tasks. A disadvantage is that each parallel buffer requires more metadata than the other techniques in order to support the management of chunks and parallel access by multiple threads. Additionally, each buffer must have at least one chunk for each thread. Even when the chunk size is set to just one tuple, the parallel buffer must be at least $t$ tuples in size.

## 4. EXPERIMENTAL SETUP

All experiments were conducted on real hardware, a Sun UltraSPARC T1, the details of which may be found in in Table 2. We chose this platform because the T1 and its recently introduced successor, the T2, are the commodity CMPs with the most on-chip thread level parallelism.

### Input Characteristics

The input to all experiments consists of 16 byte tuples with an 8 byte partitioning key and an 8 byte payload. The partitioning keys are unique and uniformly distributed. This input is similar to the input used in [8], but updated for a 64-bit processor. Due to space limitations we do not explore the implications of non-uniform and non-unique input here, but we do discuss some of these issues in Section 6.

[3]The miss latency varies with the workload and with the load on the various processors [7].

[4]The TLB is shared among the 4 threads on the core, but each thread's entries are kept mutually exclusive [11].

| | |
|---|---|
| Clock rate | 1 GHz |
| Cores (Threads/core) | 8 (4) |
| RAM | 8GB |
| Shared L2 Cache | 3MB, 12-way associative<br>64B Cache Line<br>Hit latency: 21 cycles<br>Miss latency: 90–155 cycles[3] |
| L1 Data Cache | 8KB per core<br>16B Cache Line<br>Shared by 4 threads |
| L1 Instruction Cache | 16KB per core<br>Shared by 4 threads |
| TLB | 64 Entries per core[4] |
| Supported Page Sizes | 8KB, 64KB, 4MB, 256MB |
| On-chip bandwidth | 132GB/s |
| Off-chip bandwidth | 25GB/s over 4 DDR2 |
| Operating System | Solaris 10 |
| Compiler | Sun C 5.9 |
| Flags | -fast -xtarget=native64 -mt |

**Table 2: Specifications of the Sun UltraSPARC T1.**

### Implementation Details

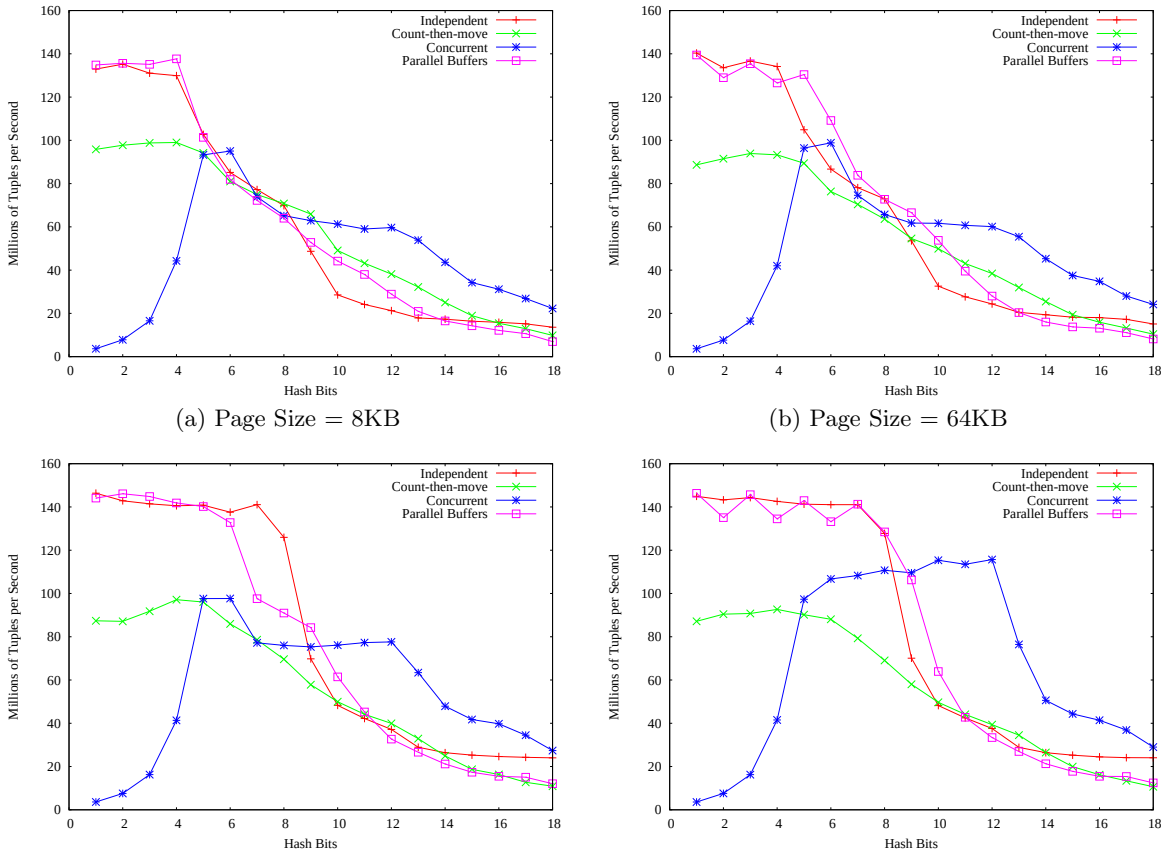
We used the pthreads library for all multithreading. Where possible we implement atomic operations with atomic intrinsics provided by the compiler rather than by using a mutex available in the threading library. This is advantageous because atomic instructions have lower latency than acquiring and releasing a lock.

Physical memory frame allocation by the operating system must be done atomically. Frame allocation may significantly reduce parallelism because threads must serialize when handling page faults that cause frame allocation from the operating system. To avoid this overhead, our code touches all of the pages to be used for writing output before collecting profiling or timing information. This is a reasonable modification because a long running database process could avoid this frame allocation bottleneck by reusing allocated buffers for partitioning and other operations.

To avoid the issue of growing an output buffer, the independent, concurrent, and parallel buffers use buffers that are allocated to be more than 50% larger than their expected size. This leads to some space overhead but simplifies the implementation and analysis. Issues such as needing to grow an output buffer in a thread safe manner are discussed in Section 6.

For all techniques, variables used for counting purposes are 32 bit integers, which helps lower the metadata overhead of each technique compared with using a 64 bit integer. For our experimental input, 32 bits is sufficient for this purpose, but in the future significantly larger partitions may require the use of 64 bit integers. The parallel buffers used in these experiments are a slightly improved version of the buffers used in [1]. The amount of metadata required per buffer has been reduced and code for writing and reading tuples has been made more modular, but the overall operation of the parallel buffer remains the same.

In the count-then-move technique, during the first pass one could record each tuple's assigned partition in a parallel array. During the second pass, this parallel array could be read instead of recomputing each tuple's partition. We tried both options and found that for hash partitioning the




(a) Page Size = 8KB

(b) Page Size = 64KB

(c) Page Size = 4MB

(d) Page Size = 256MB

**Figure 1: Comparing the throughput of four partitioning techniques using 32 threads and all available page sizes. The input cardinality was $2^{24}$ tuples.**

performance was comparable so for the experiments in this paper we recompute the hash function.

# 5. EXPERIMENTS

In this section we present an experimental evaluation of the four partitioning techniques described in Section 3. We also present data collected using the T1's hardware performance counters. All throughput values reported are averaged over eight trials of the same experiment.

## 5.1 Performance and Contention

The performance of the four partitioning techniques is shown in Figure 1. At best, partitioning can only equal the performance of copying the same quantity of data, i.e, a `memcpy`. As a comparison point, we present the performance of a multithreaded `memcpy` operation in Figure 2.

The performance of `memcpy` represents an extreme upper bound on partitioning performance because the C standard library `memcpy` uses a number of optimizations that our generic partitioning code cannot leverage. If a store instruction writes to a cache line that is not in the cache, normally a cache miss is first triggered to load the cache line and then the store proceeds. This is important because the store may only modify part of the cache line. However, it also means that writes can cause cache misses. Because `memcpy` knows that it is writing many full cache lines of data, it is able to use a block initializing store instruction that does not read the cache line from memory, but rather allocates a new zeroed cache line directly in the cache. Thus `memcpy` is able to avoid a large number of cache misses and reduce the amount of memory bandwidth that it requires. A generic partitioning algorithm has difficulty using such a store instruction because writes to the same output partition may be separated in time and across threads, therefore having an entire cache line to write at once may not be practical, but we examine using such an instruction in Section 5.6. Figure 2 also shows a naïve, user implemented multithreaded memcpy to give a sense of the best performance we might expect from any of our partitioning techniques. The independent and parallel buffer techniques actually slightly exceed the performance of this memcpy implementation for small numbers of partitions.

The only technique to suffer from contention is the concurrent output technique (Section 3.2). Figure 1 shows that when fewer than six hash bits are used, performance of the concurrent buffer technique suffers due to contention. This makes sense given that there were 32 threads used and five or fewer hash bits means that there were 32 or fewer output locations. Because the input data is uniform and unique, once the number of partitions exceeds the number of threads each thread will likely be writing to a different partition. Even though the parallel buffer technique shares a buffer, its use of chunks of tuples successfully avoids contention. In these experiments the maximum chunk size was 128 tuples, but

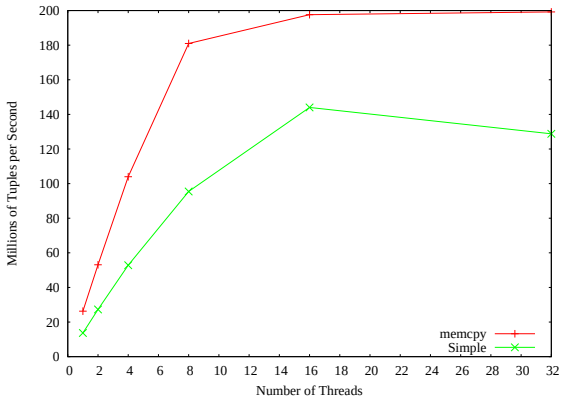



**Figure 2: Throughput for copying 40,000 tuples using C Standard Library `memcpy` and using a simple user implemented copying loop.**

that was scaled down to a size of 1 as the number of partitions increased and the expected partition size decreased.

The following sections explore in detail the factors influencing the performance reported in Figure 1, including TLB misses, cache misses, and scaling.

## 5.2 TLB Misses

TLB misses are known to be a significant performance issue during partitioning [8] and our experiments on a CMP confirm this observation. The TLB is a special cache that the processor uses to quickly translate virtual addresses to physical addresses. Because partitioning requires writing output to different partitions, TLB misses may become frequent if the TLB is not large enough to hold all of the pages to which output needs to be written. The size of a page of memory, therefore, influences the TLB's ability to contain all output pages. The Sun T1 has one 64-entry TLB per core that supports four different page sizes. Partitioning performance using those page sizes is shown in Figure 1. The difference in performance between different page sizes is due in large part to the impact of TLB misses.

TLB miss data is presented in Figure 3 and for each technique, two graphs are shown: those on the left plot only user level TLB misses, and those on the right plot both user and system level TLB misses.

Figure 3(a) shows the user level TLB misses incurred per tuple partitioned by the independent technique. A page size of 256MB results in zero TLB misses because the 64 entry TLB has a reach of 16GB, which is double the size of the machine's RAM! Each TLB is shared by four threads on one core. Because each thread has its own output location for each partition, the number of output locations the TLB must cover is $2^b * 4 = 2^{b+2}$. For small pages, the 64 TLB entry limit is reached when $b = 4$ as shown in Figure 3(a). Because the parallel buffers share a contiguous buffer, they share some output pages causing TLB misses to begin increasing at $b = 5$ instead (Figure 3(e)). The concurrent buffer has only one output location per partition regardless of the number of threads. It therefore sees no increase until $b = 6$ when the number of partitions equals the number of TLB entries (Figure 3(c)).

The plateau around one TLB miss per tuple seen for most techniques when using 8KB or 64KB pages is explained by requiring a TLB miss for the output location alone. This happens as soon as there are more than 64 output locations active per core. The number of misses then grows to two per tuple for both the independent and concurrent techniques, Figures 3(a) and 3(c), respectively. The second miss is explained by needing another miss to access the metadata associated with the output buffer. As the number of partitions increases, the storage overhead and, therefore, the number of pages required by the metadata also increases.

Figures 4(a) and 4(b) show the number of pages of metadata required for the independent and concurrent techniques, respectively. Each output buffer requires 16 bytes of state and there is one buffer per partition in the case of concurrent output or one buffer per thread per partition in the case of independent output. As Figure 4 demonstrates, the TLB cannot hold entries for all of the metadata pages when smaller page sizes are used. Because metadata pages also compete with input and output pages in the TLB, two or more TLB misses per tuple partitioned becomes increasingly likely for larger numbers of hash bits when smaller pages are used. Comparing Figures 4(a) and 4(b) with Figures 3(a) and 3(c) one can see the rise in TLB misses from one to two per tuple processed roughly coincides with the point at which the number of metadata pages exceeds the TLB capacity. This makes sense, because given our uniform input distribution, metadata pages, which contain the metadata for many output locations, are more frequently accessed than output pages. Therefore, metadata pages are more likely to be in the cache even when the number of output pages greatly exceeds the TLB capacity. It is only when the metadata pages also exceed the TLB capacity that we begin to see TLB misses for metadata as well. The impact of metadata on cache misses is discussed in Section 5.3.

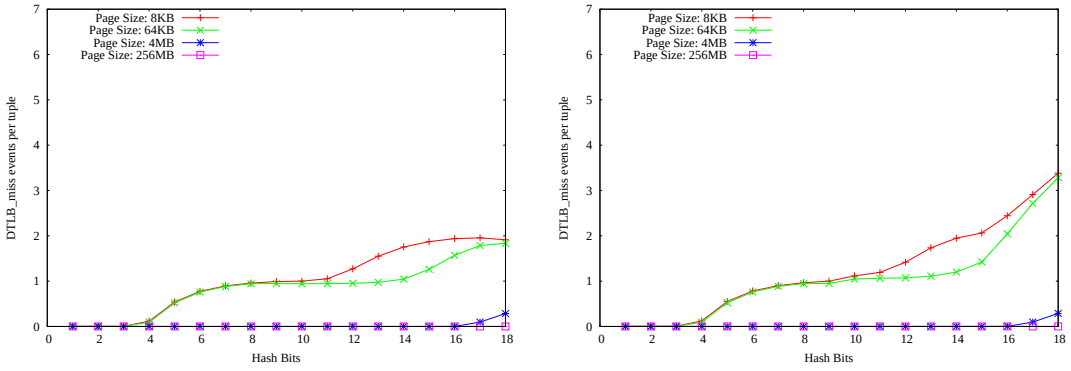
In general, when using smaller page sizes, increasing the number of output partitions beyond the reach of the TLB increases the number of TLB misses, which decreases performance. Large page sizes, such as those available on the T1, can mitigate this TLB reach issue.

The count-then-move technique's TLB misses (Figure 3(g)) also plateau around one miss per tuple, but then increase to at least three misses per tuple when more hash bits are used. This is because each pass incurs a TLB miss while reading the meta data, plus one miss when writing the output. On the first pass, the metadata miss occurs when the counter is incremented and on the second pass it happens when the write offset is read and incremented.

Similarly, parallel buffers (Figure 3(e)) incur more than two misses per tuple. This is because pieces of its metadata are stored separately to avoid false sharing between threads. More sophisticated metadata allocation could eliminate some of these TLB misses by placing all metadata for a buffer on the same page, albeit on separate cache lines to avoid false sharing.

The recorded user level misses make sense given the pattern of memory accesses caused by the different partitioning techniques. What is less obvious is the total misses – user and system level (the right column of Figure 3). In all cases except the count-then-move technique, the total TLB misses exceed the user level misses at high partitioning cardinalities.

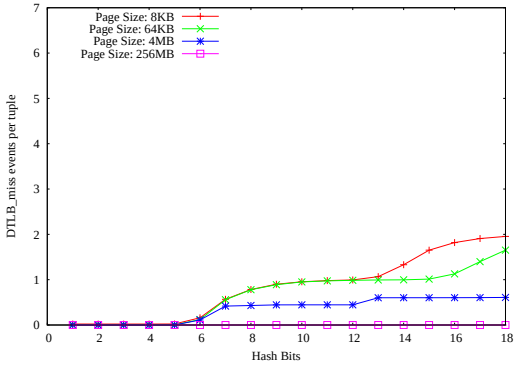
We suspect that the system is incurring additional system TLB misses while servicing user TLB misses. When a large number of partitions are used, the amount of metadata required is quite large and requires many pages when small



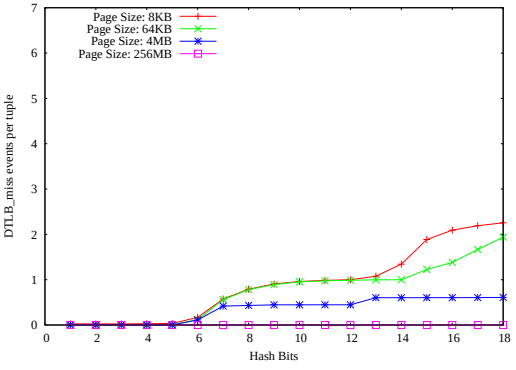

(a) Independent: User Only



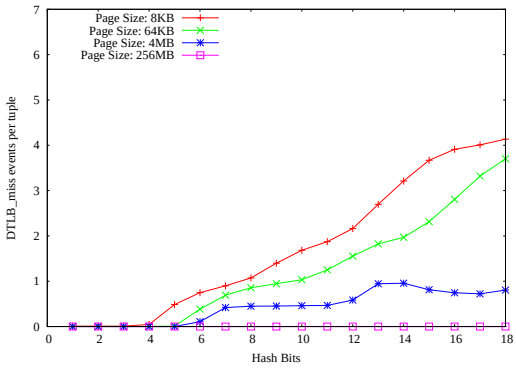

(b) Independent: User & System



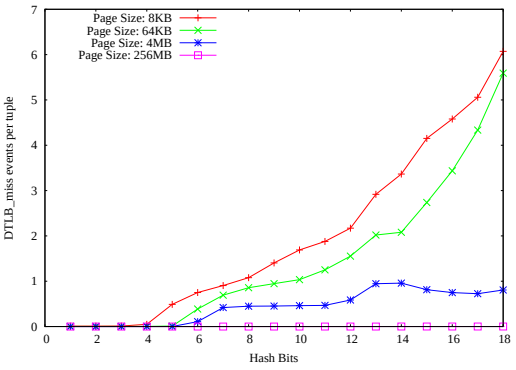

(c) Concurrent: User Only



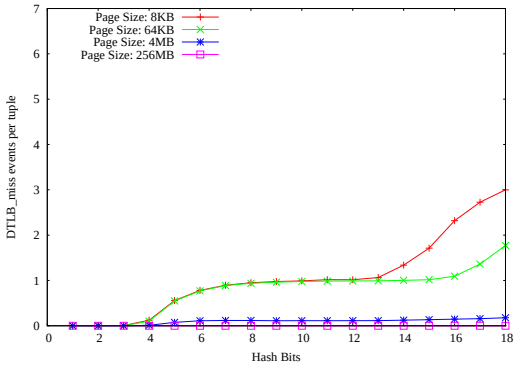

(d) Concurrent: User & System



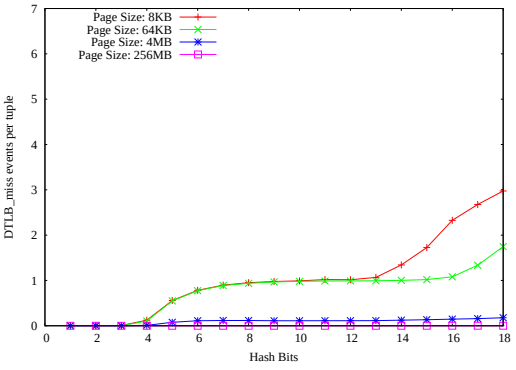

(e) Parallel Buffers: User Only




(f) Parallel Buffers: User & System

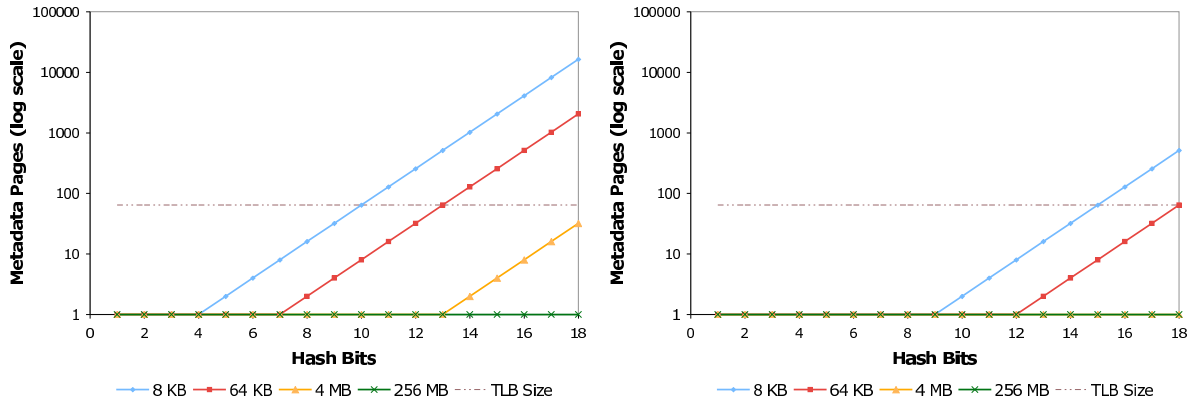



(g) Count-then-move: User Only




(h) Count-then-move: User & System

**Figure 3: TLB misses incurred by the user and system while partitioning. The input cardinality was $2^{24}$ tuples and 32 threads were used.**




(a) Independent with 32 Threads

(b) Concurrent

**Figure 4: The number of pages required to hold partitioning metadata for different page sizes.**

pages are used. This is in addition to the pages required for the $2^b$ output locations. Caching and searching all of the pages required for both writing output and managing metadata may require the operating system to use multiple levels of complex data structures[5], straining the system in such a way that it incurs TLB misses while servicing TLB misses. We advocate using large pages for partitioning as this bad TLB behavior is then avoided.

## 5.3 Cache Misses

The experiments using 256MB pages (Figure 1(d)), where there are no TLB misses, show the impact of cache misses. With large pages, the performance of the four techniques drops off at a higher number of hash bits (more partitions) than with smaller pages. This is because the TLB misses are avoided, but cache misses still impact performance.

Using hardware performance counters, we measured the number of L2 *read* misses per tuple for each tuple processed. Unfortunately, cache misses triggered by *writes* are not recorded by the hardware performance counters. The results however, do show the increasing impact of the space occupied by metadata used by the partitioning techniques. Parallel buffers, which require the most metadata per partition, incur the most L2 read misses for metadata as the number of partitions increases, while concurrent output requires the least metadata per partition and subsequently incurs the fewest L2 read misses.

Even though the performance counters do not count L2 cache write misses, we can still reason about them. To do so, we introduce the concept of an *active cache line*. A cache line is *active* during partitioning from the time that data is first written to it until it has been completely filled. Because our tuples (16B) are smaller than an L2 cache line (64B), a large amount of time may pass between the first and last write. If the time between writes is too great, the cache line may be evicted, resulting in a cache miss on every write.

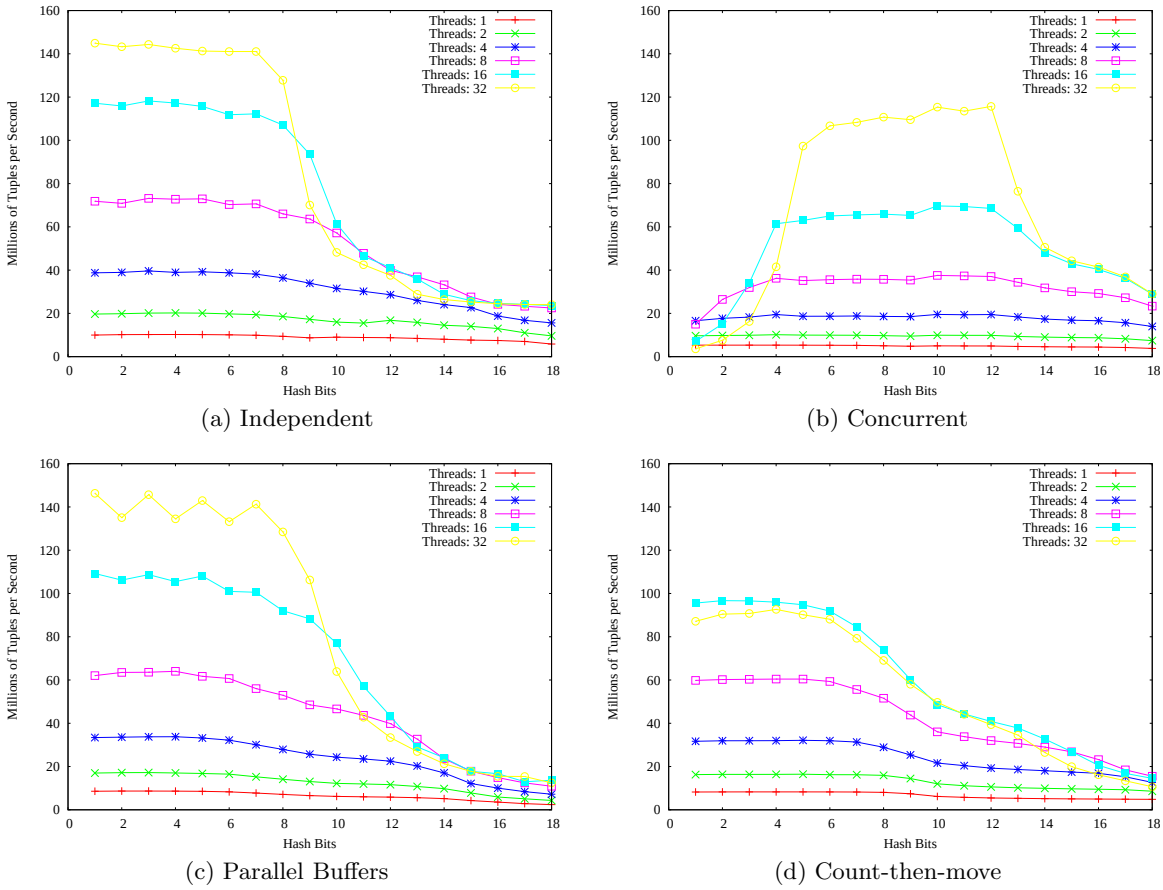
Ignoring the metadata, we can calculate the number of active cache lines for each partitioning method. If the number of active cache lines exceeds the 3MB L2 cache capacity, then the thrashing described above will occur, causing a cache miss on every write. The L2 cache contains 49152 cache lines. In the concurrent method, threads are sharing an output buffer and incrementing the output location one tuple at a time so there is just one active cache line per partition. Because our input is unique and uniform, we would therefore expect to see an increase in cache misses and therefore a drop in performance when the number of partitions exceeds the number of cache lines in the cache. Based on this analysis we predict that the performance of the concurrent technique would begin to drop off at 15 to 16 hash bits. However, Figure 1(d) shows that the performance drops off earlier. In our analysis we also need to include the metadata stored in the cache and the fact that input is also being continuously loaded into the cache. Each output buffer requires 16B of metadata so the number of cache lines needed for the metadata is $\frac{2^b * 16}{64}$. Additionally, because the input is uniform, the expected number of input tuples read between accesses to a partition is the number of partitions, $2^b$, which results in $\frac{2^b * 16}{64}$ cache lines of data added to the cache. With this amount of pressure on the cache capacity, not to mention the chance of a conflict miss, the drop in performance at 13 or 14 hash bits instead of the predicted 15 makes sense. Also, because output buffers are shared one tuple at a time, the point at which cache thrashing starts should be the same, regardless of the number of threads used as shown in Figure 5(b).

In contrast, the number of active cache lines in the other methods, increases as more threads are used because each thread has it own output location for each partition. Even in the case of parallel buffer output, although the buffer is shared, if the chunk size is equal to or greater than a cache line of tuples, each thread will write to unique cache lines within that buffer. Therefore, by using an analysis similar to the one above, we expect performance to worsen due to cache thrashing when fewer hash bits are used than with the concurrent output. This is confirmed in Figure 1(d). Furthermore, the inflection point should move left as more threads are used, which is confirmed in Figures 5(a) and 5(c) for the independent and parallel buffer output techniques, respectively. When few threads are used, there is little cache pressure for any number of partitions.

## 5.4 Scaling

All techniques stop scaling well once cache thrashing occurs as described above and shown in Figure 5. When cache misses are not an issue, additional threads improve perfor-

[5]A complete discussion of TLB miss handling in Solaris can be found in [9].




(a) Independent (b) Concurrent

(c) Parallel Buffers (d) Count-then-move

**Figure 5: Comparing the scaling of partitioning techniques as the number of threads is varied. The input cardinality was $2^{24}$ tuples and the page size was 256MB.**

mance, but the scaling is not perfect, i.e., a doubling of threads does not double the partitioning throughput. Also, as noted above, for all techniques other than the concurrent buffer, increasing the number of threads increases the cache pressure. This means that cache thrashing will occur at fewer and fewer output partitions as more threads are used. That cache thrashing is such an impediment to good scaling confirms the observation by [8] that suggests multi-pass partitioning, where each pass fits within the cache and the TLB, will give better performance than a single pass using a larger number of output partitions. For CMPs, this analysis is complicated slightly by the fact that output data structures that help to avoid thread contention also scale in terms of the number of active cache lines as the number of threads used increases. Future CMPs with more threads, all other characteristics being equal, will be limited to fewer and fewer cache resident output partitions per pass assuming that all threads are applied to the partitioning.

## 5.5 Multiple Passes

Figure 6 shows the performance of the parallel and concurrent buffer techniques using one and two passes. In the two pass versions, the first and second passes partition using $\lceil \frac{b}{2} \rceil$ and $\lfloor \frac{b}{2} \rfloor$ hash bits, respectively. As in [8], for very large numbers of partitions, making two passes using fewer hash bits performs better than one pass using all hash bits. A new

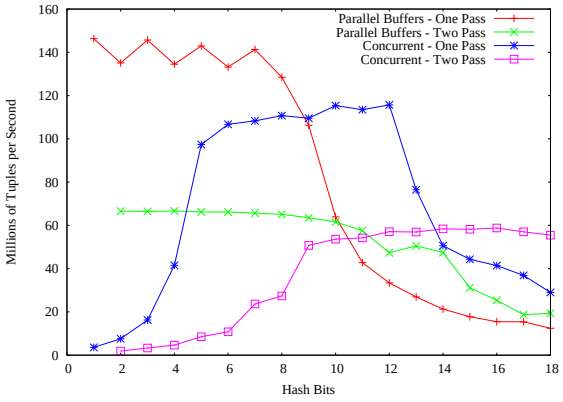
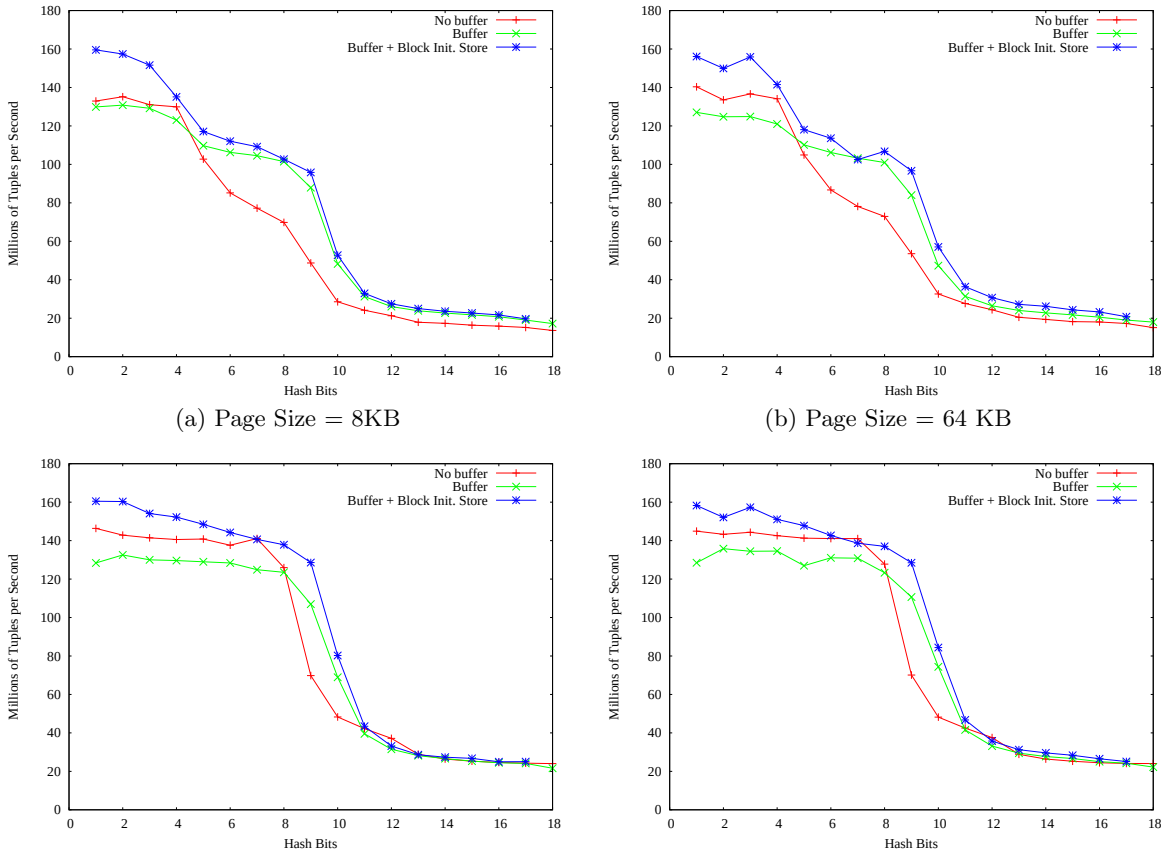



**Figure 6: One vs. two pass partitioning using 32 threads and 256MB pages to partition $2^{24}$ tuples.**

result, however, is the balancing of parallelism and the size of data structures that enable parallelism. Although parallel buffers avoid contention and have high performance when less than nine hash bits are used, two pass parallel buffer partitioning performance is worse than single pass concurrent buffer performance when nine or more hash bits are used. This is because the concurrent technique requires less metadata and fewer active cache lines per thread, therefore stay-




(a) Page Size = 8KB

(b) Page Size = 64 KB

(c) Page Size = 4MB

(d) Page Size = 256MB

**Figure 7: A throughput comparison of the independent output technique with different page sizes when using buffering with and without block initializing stores.**

ing cache resident to larger numbers of partitions. Even once one-pass concurrent partitioning performance drops off, two pass concurrent partitioning still performs better than two pass parallel buffer partitioning. Note also, though, that using two-pass concurrent partitioning results in a larger range of hash bits in which it experiences contention.

## 5.6 Buffering and Block Initializing Stores

Buffering tuples for writing to each output location may improve partitioning performance in some circumstances. One performance problem identified in Section 5.2 is that for high numbers of output partitions, each write results in a TLB miss. By collocating output buffers for all partitions, the buffers take up fewer pages than the corresponding active output partitions. Therefore, placing a tuple into a buffer is more likely to be a TLB hit. Further, when a buffer is flushed to the actual output location, only one TLB miss is required for the entire buffer, thus amortizing the cost of the TLB miss over many tuples. A comparison of the independent technique with and without a buffer is shown in Figure 7. Figure 8 demonstrates that buffering significantly lowers the number of TLB misses per tuple. Buffering alone, however, introduces additional overheads, both in terms of storage and computation, that make it perform worse than no buffering in some cases. This performance can be improved by using a block initializing store instruction when flushing a full buffer to its corresponding output partition.

When a valid block initializing store instruction is executed, rather than taking a cache miss to load the requested line from memory, a zeroed cache line is allocated directly in the L2 cache [12]. Using such an instruction when writing a buffer of tuples to an output partition, we avoid the latency of a cache miss and save some memory bandwidth. Block initializing stores on the UltraSPARC T1 must start at an offset aligned to the start of a cache line and should write a full cache line to memory. Therefore, buffering of tuples smaller than a cache line in size is required to use block initializing stores. Combining buffering with block initializing stores results in better performance as shown in Figure 7.

Because block initializing stores require the writing of entire cache lines, tuples that do not fit evenly withing a cache line or span multiple cache lines may be more challenging to properly buffer and write to a destination. In our experiment, not only are the tuples smaller than a cache line, but they also fit evenly within a cache line. Buffering is useful for our data, but may not be appropriate in all situations. A buffered partitioning implementation for tuples of arbitrary size would need to balance total buffer size with the benefit of amortizing TLB misses. If block initializing stores are to be used, one must also ensure correct writing of whole cache lines, which may be non-trivial if tuples span cache lines and the output buffer is shared among threads.

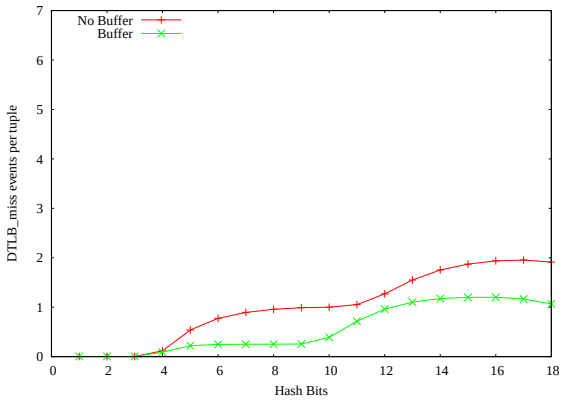



**Figure 8: TLB user level misses for the independent output technique using 8KB pages with and without buffering.**

## 6. FUTURE WORK

We realize that not all input is uniform and that uniform input, as used in these experiments, may not represent a worst case scenario for CMP partitioning. Future work includes examining CMP partitioning performance on non-unique, non-uniform input. Non-uniform distributions will have some "heavy hitter" values that may cause some partitions to grow much larger than the expected size given an assumption of uniform input.

In such a case, partition cardinality estimation may be required and output buffers must be able to grow efficiently and concurrently. Concurrent growth, in particular, may be a challenge for the techniques described in this paper that use shared buffers. The two-pass algorithm would remain unchanged for non-uniform input because the initial counting phase handles any amount of skew in actual partition size. The presence of heavy hitters in the input may actually improve performance by improving the locality of reference to some partitions, thereby reducing TLB and cache misses. On the other hand, frequently accessed partitions could become a source of contention if shared between threads.

Other future work includes using CMP partitioning to aid in the parallelizing or improving the performance of other CMP database operations. For instance, it was observed in [2] that an efficient means of clustering group by keys in the input to multithreaded hash aggregation could improve the overall performance of the aggregation operation. Unfortunately, the performance of CMP partitioning on this particular machine does not seem high enough to be able to improve the aggregation performance reported in [2].

CMP partitioning should also be investigated on CMPs with "fatter" cores than the UltraSPARC T1. These cores may have different performance characteristics due to more sophisticated support for prefetching and out-of-order execution and different capabilities in terms of atomic operations. As of publication, however, no commodity "fat" core CMP chips have on-chip thread-level parallelism to match the T1.

## 7. CONCLUSION

In this paper we have identified output coordination to be a key component of CMP partitioning performance and have presented four techniques for enabling parallel partitioning. As described in Section 3, all of the techniques have different characteristics in terms of the format of their output and the means with which they enable concurrent partitioning on a CMP. Our results confirm the work of Manegold et al. [8] in that reducing TLB and cache misses is paramount to achieving good partitioning performance. In our study, large pages and sharing output pages among threads eliminate or reduce TLB misses. We also find that cache misses are influenced by the number of partitions, whether the number of active cache lines scales with the number of threads used, and how much per thread metadata is required to manage the output buffers. Finally, an analysis of two pass partitioning shows the importance of balancing space efficiency with enabling contention free parallelism. For smaller numbers of partitions, it is important to use data structures that enable contention-free partitioning. As the number of partitions increases, one must switch to more compact, shared buffer strategies and employ more than one pass for maximum performance. In conclusion, high performance CMP partitioning requires a careful balancing of parallelism, contention, active cache lines, and metadata size.

## 8. ACKNOWLEDGMENTS

Thanks to Peter Boncz and Marcin Żukowski for the suggestion of exploring partitioning as a means to improve the performance of our adaptive aggregation technique presented at VLDB 2007. We also thank the anonymous reviewers for their helpful suggestions.

## 9. REFERENCES

[1] J. Cieslewicz et al. Parallel buffers for chip multiprocessors. In *DaMoN*, June 2007.

[2] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[3] D. J. DeWitt et al. Implementation techniques for main memory database systems. In *SIGMOD*, pages 1–8, 1984.

[4] D. J. Dewitt et al. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, 1990.

[5] D. J. DeWitt et al. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Parallel and Distributed Information Systems*, pages 280–291, 1991.

[6] G. Graefe. Parallel external sorting in volcano. Technical Report CU-CS-459-90, University of Colorado, 1990.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture*. Morgan Kaufman, 4th edition, 2007.

[8] S. Manegold et al. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[9] R. McDougall and J. Mauro. *Solaris Internals*, chapter 8-13. Prentice Hall, 2nd edition, 2007.

[10] A. Shatdal et al. Cache conscious algorithms for relational query processing. In *VLDB*, pages 510–521, 1994.

[11] Sun Microsystems, Inc. OpenSPARC T1 microarchitecture specification, August 2006.

[12] Sun Microsystems, Inc. UltraSPARC T1 supplement to the UltraSPARC architecture 2005, March 2006.

# Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines

Ryan Johnson*, Ippokratis Pandis*
*Carnegie Mellon University

Anastasia Ailamaki*†
†École Polytechnique Fédérale de Lausanne

## ABSTRACT

Critical sections in database storage engines impact performance and scalability more as the number of hardware contexts per chip continues to grow exponentially. With enough threads in the system, *some* critical section will eventually become a bottleneck. While algorithmic changes are the only long-term solution, they tend to be complex and costly to develop. Meanwhile, changes in enforcement of critical sections require much less effort. We observe that, in practice, many critical sections are so short that enforcing them contributes a significant or even dominating fraction of their total cost and tuning them directly improves database system performance. The contribution of this paper is two-fold: we (a) make a thorough performance comparison of the various synchronization primitives in the database system developer's toolbox and highlight the best ones for practical use, and (b) show that properly enforcing critical sections can delay the need to make algorithmic changes for a target number of processors.

## 1. INTRODUCTION

Ideally, a database engine would scale perfectly, with throughput remaining (nearly) proportional to the number of clients even for a large number of clients. In practice several factors limit database engine scalability. Disk and compute capacities often limit the amount of work that can be done in a given system, and badly-behaved applications (like TPC-C) generate high levels of lock contention and limit concurrency. However, these bottlenecks are all largely external to the database engine; within the storage manager itself, threads share many internal data structures. Whenever a thread accesses a shared data structure, it must prevent other threads from making concurrent modifications or data races and corruption will result. These protected accesses are known as critical sections, and can reduce scalability, especially in the absence of other, external bottlenecks.

For the forseeable future, computer architects will double the number of processor cores available each generation rather than increasing single-thread performance. Database engines are already designed to handle hundreds or even thousands of concurrent transactions, but with most of them blocked on I/O or database locks at any given moment. Even in the absence of lock or I/O bottlenecks, a limited number of hardware contexts used to bound contention for the engine's internal shared data structures. Historically, the database community has largely overlooked critical sections, either ignoring them completely or considering them a solved problem [1]. We find that as the number of active threads grows the engine's internal critical sections become a new and significant obstacle to scalability. Analysis of several open source storage managers [11] shows critical sections become bottlenecks with a relatively small number of active threads, with BerkeleyDB scaling to 4 threads, MySQL to 8, and PostgreSQL to 16. These findings indicate that many database engines are unprepared for this explosion of hardware parallelism.

As the database developer optimizes the system for scalability, algorithmic changes are required to reduce the number of threads contending for particular critical section. Additionally, we find that the method by which existing critical sections are enforced is a crucial factor in overall performance and, to some extent, scalability. Database code exhibits extremely short critical sections, such that the overhead of enforcing those critical sections is a significant or even dominating fraction of their total cost. Reducing the overhead of enforcing critical sections directly impacts performance and can even take critical sections off the critical path without the need for costly changes to algorithms.

The literature abounds with synchronization approaches and primitives which could be used to enforce critical sections, each with its own strengths and weaknesses. The database system developer must then choose the most appropriate approach for each type of critical section encountered in during the tuning process or risk lowering performance significantly.

To our knowledge there is only limited prior work that addresses the performance impact and tuning of critical sections, leaving developers to learn by trial and error which primitives are most useful. This paper illustrates the performance improvements that come from enforcing critical sections properly, using our experience developing Shore-MT [11], a scalable engine based on the Shore storage manager [4]. We also evaluate the most common types of synchronization approaches, then identify the most useful ones for enforcing the types of critical sections found in database code. Database system developers can then utilize this knowledge to select the proper synchronization tool for each critical section and maximize performance.

The rest of the paper is organized as follows. Sections 2 and 3 give an overview of critical sections in database engines and the scalability challenges they raise. Sections 4 and 5 present an overview of common synchronization approaches and evaluate their performance. Finally, Sections 6 and 7 discuss high-level observations and conclude.

## 2. CRITICAL SECTIONS INSIDE DBMS

Database engines purposefully serialize transaction threads in three ways. Database *locks* enforce consistency and isolation between transactions by preventing other transactions from accessing the lock holder's data. Locks are a form of logical protection and can be held for long durations (potentially several disk I/O times). *Latches* protect the physical integrity of database pages in the buffer pool, allowing multiple threads to read them simultaneously, or a single thread to update them. Transactions acquire latches just long enough to perform physical operations


Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008), June 13, 2008, Vancouver, Canada.

(at most one disk I/O), depending on locks to protect that data until transaction commit time. Locks and latches have been studied extensively [1][7]. Database locks are especially expensive to manage, prompting proposals for hardware acceleration [21].

Critical sections form the third source of serialization. Database engines employ many complex, shared data structures; critical sections (usually enforced with semaphores or mutex locks) protect the physical integrity of these data structures in the same way that latches protect page integrity. Unlike latches and locks, critical sections have short and predictable durations because they seldom span I/O requests or complex algorithms; often the thread only needs to read or update a handful of memory locations. For example, a critical section might protect traversal of a linked list. Critical sections abound throughout the storage engine's code. In Shore-MT, for example, we estimate that a TPC-C Payment transaction — which only touches 4-6 database records — enters roughly one hundred critical sections before committing. Under these circumstances, even uncontended critical sections are important because the accumulated overhead can contribute a significant fraction of overall cost. The rest of this section presents an overview of major storage manager components and lists the kinds of critical sections they make use of.

**Buffer Pool Manager.** The buffer pool manager maintains a pool for in-memory copies of in-use and recently-used database pages and ensures that the pages on disk and in memory are consistent with each other. The buffer pool consists of a fixed number of *frames* which hold copies of disk pages and provide latches to protect page data. The buffer pool uses a hash table that maps page IDs to frames for fast access, and a critical section protects the list of pages at each hash bucket. Whenever a transaction accesses a persistent value (data or metadata) it must locate the frame for that page, *pin* it, then latch it. Pinning prevents the pool manager from evicting the page while a thread acquires the latch. Once the page access is complete, the thread unlatches and unpins the page, allowing the buffer pool to recycle its frame for other pages if necessary. Page misses require a search of the buffer pool for a suitable page to evict, adding yet another critical section. Overall, acquiring and releasing a single page latch requires at least 3-4 critical sections, and more if the page gets read from disk.

**Lock Manager.** Database locks preserve isolation and consistency properties between transactions. Database locks are hierarchical, meaning that a transaction wishing to lock one row of a table must first lock the database and the table in an appropriate *intent* mode. Hierarchical locks allow transactions to balance granularity with overhead: fine-grained locks allow high concurrency but are expensive to acquire in large numbers. A transaction which plans to read many records of a table can avoid the cost of acquiring row locks by *escalating* to a single table lock instead. However, other transactions which attempt to modify unrelated rows in the same table would then be forced to wait. The number of possible locks scales with the size of the database, so the storage engine maintains a lock pool very similar to the buffer pool.

The lock pool features critical sections that protect the lock object freelist and the linked list at each hash bucket. Each lock object also has a critical section to "pin" it and prevent recycling while it is in use, and another to protect its internal state. This means that, to acquire a row lock, a thread enters at least three critical sections for each of the database, table, and row locks.

**Log Manager.** The log manager ensures that modified pages in memory are not lost in the event of a failure: all changes to pages are logged before the actual change is made, allowing the page's latest state to be reconstructed during recovery. Every log insert requires a critical section to serialize log entries and another to coordinate with log flushes. An update to a given database record often involves several log entries due to index and metadata updates that go with it.

**Free Space Management**. The storage manager maintains metadata which tracks disk page allocation and utilization. This information allows the storage manager to allocate unused pages to tables efficiently. Each record insert (or update that increases record size) requires entering several critical sections to determine whether the current page has space and to allocate new pages as necessary. Note that the transaction must also latch the free space manager's metadata pages and log any updates.

**Transaction Management**: The system maintains a total order of transactions in order to resolve lock conflicts and maintain proper transaction isolation. Whenever a transaction begins or ends this global state must be updated. In addition, no transaction may commit during a log checkpoint operation, in order to ensure that the resulting checkpoint is consistent. Finally, multi-threaded transactions must serialize the threads within a transaction in order to update per-transaction state such as lock caches.

# 3. THE DREADED CRITICAL SECTION

By definition, critical sections limit scalability by serializing the threads which compete for them. Each critical section is simply one more limited resource in the system that supports some maximum throughput. As Moore's Law increases the number of threads which can execute concurrently, the demand on critical sections increases and they invariably enter the critical path to become the bottleneck in the system. Database engine designers can potentially improve critical section capacity (i.e. peak throughput) by changing how they are enforced or by altering algorithms and data structures.

## 3.1 Algorithmic Changes

Algorithmic changes address bottleneck critical sections by either reducing how often threads enter them (ideally never), or by breaking them into several "smaller" ones in a way that distributes contending threads as well (ideally, each thread can expect an uncontended critical section). For example, buffer pool managers typically distribute critical sections by hash bucket so that only probes for pages in the same bucket must be serialized.

In theory, algorithmic changes are the superior approach for addressing critical sections because they can remove or distribute critical sections to ease contention. Unfortunately, developing and implementing new algorithms is challenging and time consuming, with no guarantee of a breakthrough for a given amount of effort. In addition, even the best-designed algorithms will eventually become bottlenecks again if the number of threads increases enough, or if non-uniform access patterns cause hotspots.

## 3.2 Changing Synchronization Primitives

The other approach for improving critical section throughput is by altering how they are enforced. Because the critical sections we are interested in are so short, the cost of enforcing them is a significant — or even dominating — fraction of their overall cost. Reducing the cost of enforcing a bottleneck critical section can improve performance a surprising amount. Also, critical sections

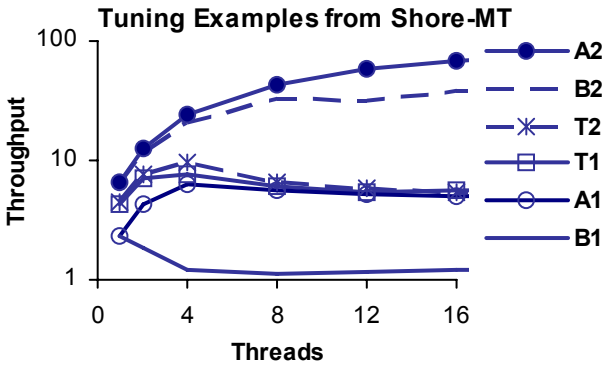



**Figure 1.** *Algorithmic changes and tuning combine to give best performance. A<n> is an algorithm change; B<n> is a baseline, T<n> is synchronization tuning.*

tend to be encapsulated by their surrounding data structures, so the developer can change how they are enforced simply by replacing the existing synchronization primitive with a different one. These characteristics make critical section tuning attractive if it can avoid or delay the need for costly algorithmic changes.

## 3.3 Both are Needed

Figure 1 illustrates how algorithmic changes and synchronization tuning combined give the best performance. It presents the performance of Shore-MT at several stages of tuning, with throughput given on the log-scale y-axis as the number of threads in the system varies along the x-axis. These numbers came from the experience of converting Shore to Shore-MT [11]. The process involved beginning with a thread-safe but very slow version of Shore and repeatedly addressing critical sections until internal scalability bottlenecks had all been removed. The changes involved algorithmic and synchronization changes in all the major components of the storage manager, including logging, locking, and buffer pool management. The figure shows the performance and scalability of Shore-MT at various stages of tuning. Each thread repeatedly runs transactions which insert records into a private table. These transactions exhibit no logical contention with each other but tend to expose many internal bottlenecks. Note that, in order to show the wide range of performance the y-axis of the figure is log-scale; the final version of Shore-MT scales nearly as well as running each thread in an independent copy of Shore-MT.

The "B1" line at the bottom represents the thread-safe but unoptimized Shore; the first optimization (A1) replaced the central buffer pool mutex with one mutex per hash bucket. As a result, scalability improved from one thread to nearly four, but single-thread performance did not change. The second optimization (T1) replaced the expensive pthread mutex protecting buffer pool buckets with a fast test and set mutex (see Section 4 for details about synchronization primitives), doubling throughput for a single thread. The third optimization (T2) replaced the test-and-set mutex with a more scalable MCS mutex, allowing the doubled throughput to persist until other bottlenecks asserted themselves at four threads.

B2 represents the performance of Shore-MT after many subsequent optimizations, when the buffer pool again became a bottleneck. Because the critical sections were already as efficient as possible, another algorithmic change was required (A2). This time the open-chained hash table was replaced with a cuckoo hash table to further reduce contention for hash buckets, improving scalability from 8 to 16 threads and beyond (details in [11]).

This example illustrates how both proper algorithms and proper synchronization are required to achieve the highest performance. In general, tuning primitives improves performance significantly, and sometimes scalability as well; algorithmic changes improve scalability and might help or hurt performance (more scalable algorithms tend to be more expensive). Finally, we note that the two tuning optimizations each required only a few *minutes* to apply, while each of the algorithmic changes required several *days* to implement and debug. The performance impact and ease of reducing critical section overhead makes tuning an important part of the optimization process.

# 4. SYNCHRONIZATION APPROACHES

The literature abounds with different synchronization primitives and approaches, each with different *overhead* (cost to enter an uncontended critical section) and *scalability* (whether, and by how much, overhead increases under contention). Unfortunately, efficiency and scalability tend to be inversely related: the cheapest primitives are unscalable, and the most scalable ones impose high overhead; as the previous section illustrated, both metrics impact the performance of a database engine. Next we present a brief overview of the types of primitives available to the designer.

## 4.1 Synchronization Primitives

The most common approach to synchronization is to use a synchronization primitive to enforce the critical section. There are a wide variety of primitives to choose from, all more or less interchangeable with respect to correctness.

**Blocking Mutex.** All operating systems provide heavyweight blocking mutex implementations. Under contention these primitives deschedule waiting threads until the holding thread releases the mutex. These primitives are fairly easy to use and understand, in addition to being portable. Unfortunately, due to the cost of context switching and their close association with the kernel scheduler, they are not particularly cheap or scalable for the short critical sections we are interested in.

**Test-and-set Spinlocks.** Test-and-set (TAS) spinlocks are the simplest mutex implementation. Acquiring threads use an atomic operation such as a SWAP to simultaneously lock the primitive and determine if it was already locked by another thread, repeating until they lock the mutex. A thread releases a TAS spinlock using a single store. Because of their simplicity TAS spinlocks are extremely efficient. Unfortunately, they are also among the least-scalable synchronization approaches because they impose a heavy burden on the memory subsystem. Variants such as test-and-test-and-set [22] (TATAS), exponential back-off [2], and ticket-based [20] approaches reduce the problem somewhat, but do not solve it completely. Backoff schemes, in particular, are very difficult (and hardware-dependent) to tune.

**Queue-based Spinlocks.** Queue-based spinlocks organize contending threads into a linked list queue where each thread spins on a different memory location. The thread at the head of the queue holds the lock, handing off to a successor when it completes. Threads compete only long enough to append themselves to the tail of the queue. The two best-known queuing spinlocks are MCS [16] and CLH [5][15], which differ mainly in how they manage their queues. MCS queue links point toward the tail, while CLH

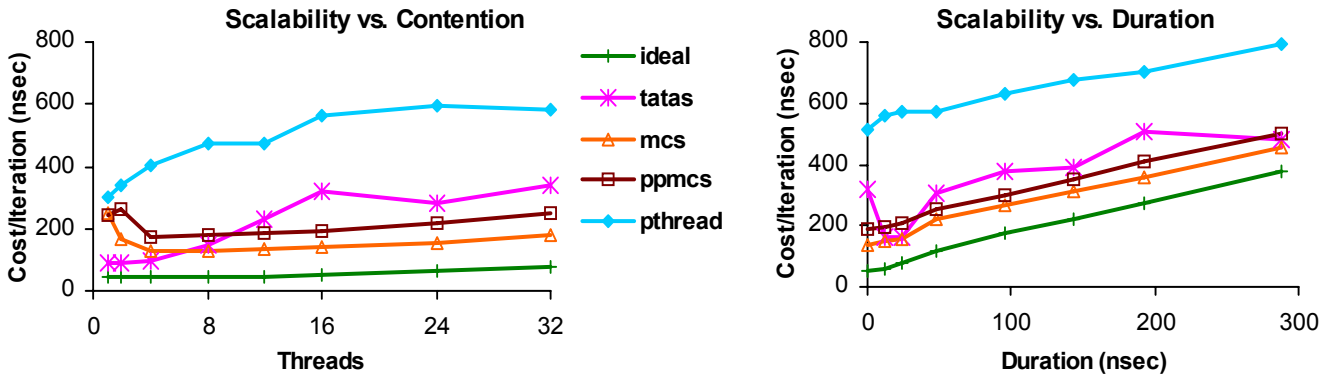



**Figure 2.***Performance of mutex locks as the contention (left) and the duration of the CS (right) vary.*

links point toward the head. Queuing improves on test-and-set by eliminating the burden on the memory system and also by decoupling lock contention from lock hand-off. Unfortunately, each thread is responsible to allocate and maintain a queue node for each lock it acquires. In our experience, memory management can quickly become cumbersome in complex code, especially for CLH locks, which require heap-allocated state.

**Reader-Writer Locks.** In certain situations, threads enter a critical section only to prevent other threads from changing the data to be read. Reader-writer locks allow either multiple readers or one writer to enter the critical section simultaneously, but not both. While operating systems typically provide a reader-writer lock, we find that the pthreads implementation suffers from extremely high overhead and poor scalability, making it useless in practice. The most straightforward reader-writer locks use a normal mutex to protect their internal state; more sophisticated approaches extend queuing locks to support reader-writer semantics [17][13].

**A Note About Convoys.** Some synchronization primitives, such as blocking mutex and queue-based spinlocks, are vulnerable to forming stable quasi-deadlocks known as convoys [3]. Convoys occur when the lock passes to a thread that has been descheduled while waiting its turn. Other threads must then wait for the thread to be rescheduled, increasing the chances of further preemptions. The result is that the lock sits nearly idle even under heavy contention. Recent work [8] has provided a preemption-resistant form of queuing lock, at the cost of additional overhead which can put medium-contention critical sections squarely on the critical path.

## 4.2 Alternatives to Locking

Under certain circumstances critical sections can be enforced without resorting to locks. For example, independent reads and writes to a single machine word are already atomic and need no further protection. Other, more sophisticated approaches such as optimistic concurrency control and lock-free data structures allow larger critical sections as well.

**Optimistic Concurrency Control.** Many data structures feature *read-mostly* critical sections, where updates occur rarely, and often come from a single writer. The reader's critical sections are often extremely short and overhead dominates the overall cost. Under these circumstances, optimistic concurrency control schemes can improve performance dramatically by assuming no writer will interfere during the operation. The reader performs the operation without enforcing any critical section, then afterward verifies that no writer interfered (e.g. by checking a version stamp). In the rare event that the assumption did not hold, the reader blocks or retries. The main drawbacks to OCC are that it cannot be applied to all critical sections (since side effects are unsafe until the read is verified), and unexpectedly high writer activity can lead to livelock as readers endlessly block or abort and retry.

**Lock-free Data Structures.** Much current research focuses on lock-free data structures [9] as a way to avoid the problems that come with mutual exclusion (e.g. [14][6]). These schemes usually combine optimistic concurrency control and atomic operations to produce data structures that can be accessed concurrently without enforcing critical sections. Unfortunately there is no known general approach to designing lock free data structures; each must be conceived and developed separately, so database engine designers are have a limited library to choose from. In addition, lock-free approaches can suffer from livelock unless they are also *wait-free*, and may or may not be faster than the lock-based approaches under low and medium contention (many papers provide only asymptotic performance analyses rather than benchmark results).

**Transactional Memory.** Transactional memory approaches enforce critical sections using database-style “transactions” which complete atomically or not at all. This approach eases many of the difficulties of lock-based programming and has been widely researched. Unfortunately, software-based approaches [23] impose too much overhead for the tiny critical sections we are interested in, while hardware approaches [10][19] generally suffer from complexity, lack of generality, or both, and have not been adopted. Finally, we note that transactions do not inherently remove contention; at best transactional memory can serialize critical sections with very little overhead.

# 5. CHOOSING THE RIGHT APPROACH

This section evaluates the different synchronization approaches using a series of microbenchmarks that replicate the kinds of critical sections found in database code. We present the performance of the various approaches as we vary three parameters: Contended vs. uncontended accesses, short vs. long duration, and read-mostly vs. mutex critical sections. We then use the results to identify the primitives which work best in each situation.

Each microbenchmark creates N threads which compete for a lock in a tight loop over a one second measurement interval (typically 1-10M iterations). The metric of interest is cost per iteration per thread, measured in nanoseconds of wall-clock time. Each iteration begins with a delay of *To* ns to represent time spent out-

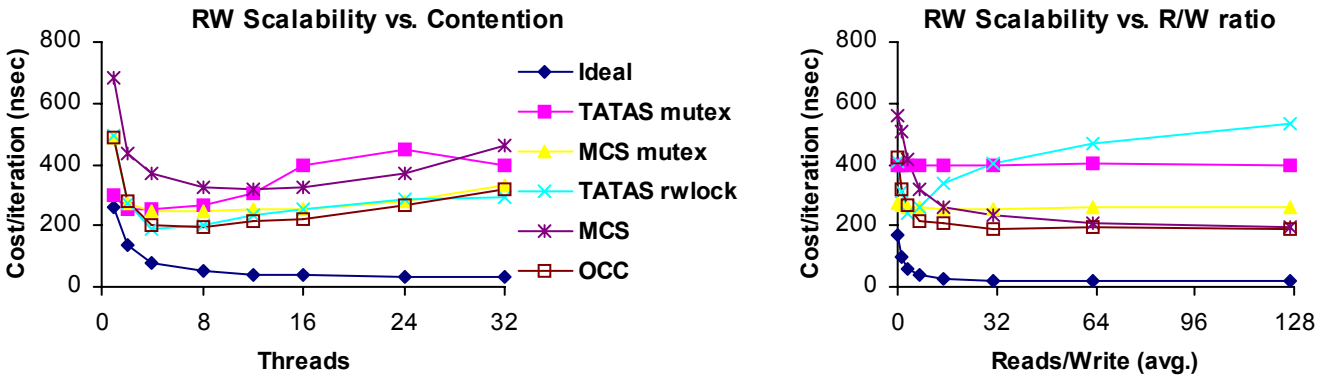



**Figure 3.***Performance of reader-writer locks as contention (left) and reader-writer ratio (right) vary.*

side the critical section, followed by an acquire operation. Once the thread has entered the critical section, it delays for *Ti* ns to represent the work performed inside the critical section, then performs a release operation. All delays are measured to 4 ns accuracy using the machine's cycle count register; we avoid unnecessary memory accesses to prevent unpredictable cache misses or contention for hardware resources.

For each scenario we compute an ideal cost by examining the time required to serialize *Ti* plus the overhead of a memory barrier, which is always required for correctness. Experiments involving reader-writers are set up exactly the same way, except that readers are assumed to perform their memory barrier in parallel and threads use a pre-computed array of random numbers to determine whether they should perform a read or write operation.

All of our experiments were performed using a Sun T2000 (Niagara [12]) server running Solaris 10. The Niagara chip is a multi-core architecture with 8 cores; each core provides 4 hardware contexts for a total of 32 OS-visible "processors". Cores communicate through a shared 3MB L2 cache.

## 5.1 Contention

Figure 2 (left) compares the behavior of four mutex implementations as the number of threads in the system varies along the x-axis. The y-axis gives the cost of one iteration as seen by one thread. In order to maximize contention, we set both *To* and *Ti* to zero; threads spend all their time acquiring and releasing the mutex. TATAS is a test-and-set spinlock variant. MCS and ppMCS are the original and preemption-resistant MCS locks, respectively, while pthread is the native pthread mutex. Finally, "ideal" represents the lowest achievable cost per iteration, assuming that the only overhead of enforcing the critical section comes from the memory barriers which must be present for correctness.

As the degree of contention of the particular critical section changes, different synchronization primitives become more appealing. The native pthread mutex is both expensive and unscalable, making it unattractive. TATAS is by far the cheapest for a single thread, but quickly falls behind as contention increases. We also note that all test-and-set variants are extremely unfair, as the thread which most recently released it is likely to re-acquire it before other threads can respond. In contrast, the queue-based locks give each thread equal attention.

## 5.2 Duration

Another factor of interest is the performance of the various synchronization primitives as the duration of the critical section varies (under medium contention) from extremely short to merely short. We assume that a long, heavily-contended critical section is a design flaw which must be addressed algorithmically.

Figure 2 (right) shows the cost of each iteration as 16 threads compete for each mutex. The inner and outer delays both vary by the amount shown along the x-axis (keeping contention steady). We see the same trends as before, with the main change being the increase in ideal cost (due to the critical section's contents). As the critical section increases in length, the overhead of each primitive matters less; however, ppMCS and TATAS still impose 10% higher cost than MCS, while pthread more than doubles the cost.

## 5.3 Reader/Writer Ratio

The last parameter we study is the ratio between the readers and the writers. Figure 3 (left) characterizes the performance of several reader-writer locks when subjected to 7 reads for every write and with *To* and *Ti* both set to 100 ns. The cost/iteration is shown on the y-axis as the number of competing threads varies along the x-axis. The TATAS mutex and MCS mutex apply mutual exclusion to both readers and writers. The TATAS rwlock extends a normal TATAS mutex to use a read/write counter instead of a single "locked" flag. The MCS rwlock comes from the literature [13]. OCC lets readers increment a simple counter as long as no writers are around; if a writer arrives, all threads (readers and writers) serialize through an MCS lock instead.

We observe that reader-writer locks are significantly more expensive than their mutex counterparts, due to the extra complexity they impose. For very short critical sections and low reader ratios, a mutex actually outperforms the rwlock; even for the 100ns case shown here, the MCS lock is a usable alternative.

Figure 3 (right) fixes the number of threads at 16 and varies the reader ratio from 0 (all writes) to 127 (mostly reads) with the same delays as before. As we can see, the MCS rwlock performs well for high reader ratios, but the OCC approach dominates it, especially for low reader ratios. For the lowest read ratios, the MCS mutex performs the best — the probability of multiple concurrent reads is too low to justify the overhead of a rwlock.

# 6. DISCUSSION AND OPEN ISSUES

The microbenchmarks from the previous section illustrate the wide range in performance and scalability among the different

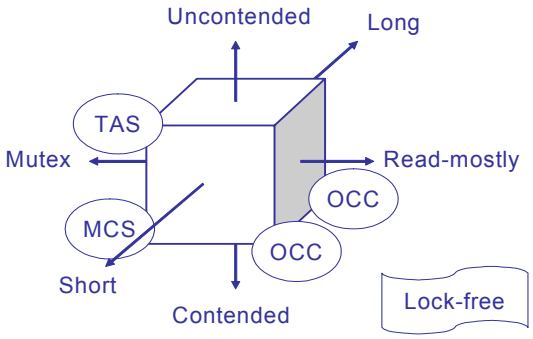



**Figure 4.***The space of critical section types. Each corner of the cube is marked with the appropriate synchronization primitive to use for that type of critical section.*

primitives. From the contention experiment we see that the TATAS lock performs best under low contention due to having the lowest overhead; for high contention, the MCS lock is superior thanks to its scalability. The experiment also highlights how expensive it is to enforce critical sections. The ideal case (memory barrier alone) costs 50 ns, and even TATAS costs twice that. The other alternatives cost 250 ns or more. By comparison a store costs roughly 10 ns, meaning critical sections which update only a handful of values suffer more than 80% overhead. As the duration experiment shows, pthread and TATAS are undesirable even for longer critical sections that amortize the cost somewhat. Finally, the reader-writer experiment demonstrates the extremely high cost of reader-writer synchronization; a mutex outperforms rwlocks at low read ratios by virtue of its simplicity, while optimistic concurrency control wins at high ratios. Figure 4 summarizes the results of the experiments, showing which of the three synchronization primitives to use under what circumstances. We note that, given a suitable algorithm, the lock free approach might be best.

The results also suggest that there is much room for improvement in the synchronization primitives that protect small critical sections. Hardware-assisted approaches (e.g. [18]) and implementable transactional memory might be worth exploring further in order to reduce overhead and improve scalability. Reader-writer primitives, especially, do not perform well as threads must still serialize long enough to identify each other as readers and check for writers.

## 7. CONCLUSION

Critical sections are emerging as a major obstacle to scalability as the number of hardware contexts in modern systems continues to grow and a large part of the execution is computation-bound. We observe that algorithmic changes and proper use of synchronization primitives are both vital to maximize performance and keep critical sections off the critical path in database engines and that even uncontended critical sections sap performance because of the overhead they impose. We identify a small set of especially useful synchronization primitives which a developer can use for enforcing critical sections. Finally, we identify several areas where currently available primitives fall short, indicating potential avenues for future research.

## 8. ACKNOWLEDGEMENTS

We thank Brian Gold and Brett Meyer for their insights and suggestions, and the reviewers for their helpful comments. This work was partially supported by grants and equipment from Intel; a Sloan research fellowship; an IBM faculty partnership award; and NSF grants CCR-0205544, CCR-0509356, and IIS-0133686.

## 9. REFERENCES

[1] R. Agrawal, M. Carey, and M. Livny. “Concurrency control performance modeling: alternatives and implications.” ACM TODS, 12(4), 1987.

[2] T. Anderson. “The performance of spin lock alternatives for shared-memory multiprocessors.” IEEE TPDS, 1(1), 1990.

[3] M. Blasgen, J. Gray, M. Mitona, and T. Price. “The Convoy Phenomenon.” ACM SIGOPS, 13(2), 1979.

[4] M. Carey, et al. “Shoring up persistent applications.” In Proc. SIGMOD, 1994.

[5] T. Craig. “Building FIFO and priority-queueing spin locks from atomic swap.” Technical Report TR 93-02-02, University of Washington, Dept. of Computer Science, 1993.

[6] M. Fomitchev, and E. Rupert. “Lock-free linked lists and skip lists.” In Proc. PODC, 2004.

[7] V. Gottemukkala, and T. J. Lehman. “Locking and latching in a memory-resident database system.” In Proc. VLDB, 1992.

[8] B. He, W. N. Scherer III, and M. L. Scott. “Preemption adaptivity in time-published queue-based spin locks.” In Proc. HiPC, 2005.

[9] M. Herlihy. “Wait-free synchronization.” ACM TOPLAS, 13(1), 1991.

[10] M. Herlihy and J. Moss. “Transactional memory: architectural support for lock-free data structures.” In Proc. ISCA, 1993.

[11] R. Johnson, I. Pandis, N. Hardavellas, and A. Ailamaki. “Shore-MT: A Quest for Scalability in the Many-Core Era.” CMU-CS-08-114.

[12] P. Kongetira, K. Aingaran, and K. Olukotun. “Niagara: A 32-Way Multithreaded SPARC Processor.” IEEE MICRO, 2005.

[13] O. Krieger, M. Stumm, and R. Unrau. “A Fair Fast Scalable Reader-Writer Lock.” In Proc. ICPP, 1993

[14] M. Maged. “High performance dynamic lock-free hash tables and list-based sets.” In Proc. SPAA, 2002.

[15] P. Magnussen, A. Landin, and E. Hagersten. “Queue locks on cache coherent multiprocessors.” In Proc. IPPS, 1994.

[16] J. Mellor-Crummey, and M. Scot. “Algorithms for scalable synchronization on shared-memory multiprocessors.” ACM TOCS, 9(1), 1991.

[17] J. Mellor-Crummey, and M. L. Scott. “Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors.” In Proc. PPoPP, 1991.

[18] R. Rajwar, and J. Goodman. “Speculative lock elision: enabling highly concurrent multithreaded execution.” IEEE MICRO, 2001.

[19] R. Rajwar and J. Goodman. “Transactional lock-free execution of lock-based programs.” SIGPLAN Notices, 37(10), 2002.

[20] D. P. Reed, and R. K. Kanodia. “Synchronization with event-counts and sequencers.” Commun. ACM 22(2), 1979.

[21] J. T. Robinson. “A fast, general-purpose hardware synchronization mechanism.” In Proc. SIGMOD, 1985.

[22] L. Rudolph and Z. Segall. “Dynamic decentralized cache schemes for MIMD parallel processors.” In Proc ISCA, 1984.

[23] N. Shavit and D. Touitou. “Software Transactional Memory.” In Proc. PODC, 1995.

# Avoiding Version Redundancy for High Performance Reads in Temporal DataBases

Khaled Jouini
khaled.jouini@dauphine.fr

Geneviève Jomier
genevieve.jomier@dauphine.fr

Université Paris Dauphine
Place du Maréchal de Lattre de Tassigny
Paris, France

## ABSTRACT

A major performance bottleneck for database systems is the memory hierarchy. The performance of the memory hierarchy is directly related to how the content of disk pages maps to the L2 cache lines, *i.e.* to the organization of data within a disk page, called the *page layout*. The prevalent page layout in database systems is the N-ary Storage Model (NSM). As demonstrated in this paper, using NSM for temporal data deteriorates memory hierarchy performance for query-intensive workloads. This paper proposes two cache-conscious, read-optimized, page layouts for temporal data. Experiments show that the proposed page layouts are substantially faster than NSM.

## 1. INTRODUCTION

Database systems (DBMS) fetch data from non-volatile storage (*e.g.* disk) to processor in order to execute queries. Data goes through the *memory hierarchy* which consists of disk, main memory, L2 cache, L1 cache [4]. The communication between the main memory and the disk has been traditionally recognized as the dominant database performance bottleneck. However, architectural research on modern platforms has pointed out that the L2 cache miss penalty has an increasing impact on response times [3]. As a result, DBMS should be designed to be sensitive, not only to disk and main memory performance, but also to L2 cache performance [2].

The mapping of disk page content to L2 cache lines is determined by the organization of data within the page, called the *page layout* [2]. Thus, the page layout highly impacts the memory hierarchy utilization of DBMS [15]. The prevalent page layout in commercial DBMS is the *N-ary Storage Model* (NSM), also called *row-store architecture* [16]. NSM stores all the attributes of a tuple contiguously within a disk page. While NSM provides a generic platform for a wide range of data storage needs, recent studies demonstrate that it exhibits poor memory hierarchy performance for query-intensive applications [17]. In contrast with OLTP-style applications, query-intensive applications require faster reads, while tolerating slower writes. Hence, they should be *read-optimized* [16]. Typical examples are data warehouses and customer relationship management systems, where relatively long periods of ad-hoc queries are interspersed with periodic bulk-loading of new data [16]. Recently, page layouts alternative to NSM have been implemented in academic and commercial read-optimized systems [20, 19, 5, 21].

This paper focuses on read-optimized, cache-conscious page layouts for temporal data. Various characteristics of temporal data make this problem novel. In temporal databases, in order to keep past, whenever a modeled entity $e$ is modified, its old version is retained and a new version of $e$ is created. Thus, an entity $e$ may be represented in a single temporal relation by a set of tuples. Each tuple contains a timestamp $t$ and records the state (or the version) of $e$ at $t$. Figure 1.a depicts a sample temporal relation *product* (*entity surrogate, timestamp, name, price, $CO_2$ consumption*) in NSM-style representation. In this example, as in the remainder of this paper, time is assumed to be linear and totally ordered: $t_i < t_{i+1}$. Let $t_i$ and $t_j$ be two timestamps such that: $(i)$ $t_i < t_j$; and $(ii)$ the state of an entity $e$ is modified at $t_i$ and at $t_j$, but is unchanged between them. As the state of $e$ remains unchanged between $t_i$ and $t_j$, it is recorded only once by the tuple identified by $(e, t_i)$. The tuple $(e, t_i)$ is said to be *alive* or *valid* for each $t \in [t_i, t_j)$; $[t_i, t_j)$ expresses the *lifespan* or the *time validity* of $(e, t_i)$.

In most cases: $(i)$ only a small fraction of the attributes of an entity are time-varying; and $(ii)$ time-varying attributes vary independently over time. With NSM, even if only one attribute is updated, all the other attributes are duplicated. For example, in figure 1.a the update of the price of prod-

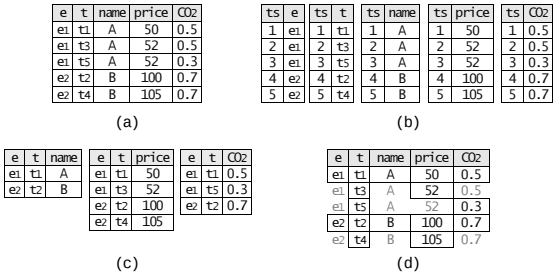



**Figure 1: (a) A sample temporal relation *product* (*entity surrogate, timestamp, name, price, $CO_2$ consumption*) in NSM-style representation. (b) Straight-forward DSM; *ts*: tuple surrogate. (c) Temporal DSM. (d) PSP only stores values written in black. Other values are implicit.**


*Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008),* June 13, 2008, Vancouver, Canada.

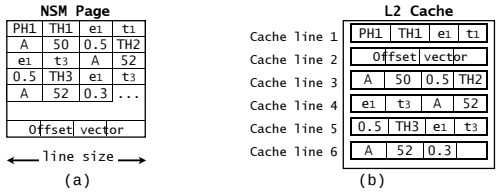



**Figure 2: (a) NSM page layout. (b) Cache behavior for "find the $CO_2$ consumption history of $A$".**

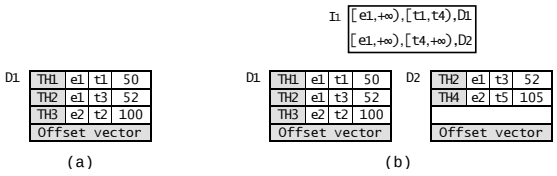



**Figure 3: (a) The TSB-tree starts with one data page. (b) Time split of $D_1$ at $t_4$**

uct $A$ at $t_3$ leads to the replication of his name and $CO_2$ consumption. We call this type of replication *version redundancy*. The important issue here is not the disk space consumed by version redundancy, as disk space costs virtually nothing nowadays. The issue is that loading the memory hierarchy several times with the same data: ($i$) wastes disk and main memory bandwidths; ($ii$) pollutes the main memory and the L2 cache; and ($iii$) increases the amount of CPU cycles wasted in waiting for data loading.

This paper introduces the *Temporal Decomposition Storage Model* (TDSM) and the *Per Surrogate Partitioning* storage model (PSP), two page layouts specifically tailored for temporal data. TDSM and PSP aim at avoiding version redundancy to achieve: ($i$) reasonable performance for writes; and ($ii$) high-performance reads. The remainder of this paper is organized as follows. Section 2 illustrates the use of conventional page layouts for temporal data. Section 3 introduces TDSM and PSP. Section 4 compares the performance of PSP, NSM and TDSM. Section 5 reviews related work. Section 6 concludes the paper.

# 2. CONVENTIONAL PAGE LAYOUTS

## 2.1 N-ary Storage Model

Each NSM page has a *Page Header* (PH) containing information such as the page identifier and the total remaining free space [13]. Each tuple in an NSM page is preceded by a *Tuple Header* (TH) providing metadata about the tuple, such as the length of the tuple and offsets of variable-length attributes [13]. To locate tuples within a page, the starting offsets of tuples are kept in an *offset vector* [13]. Typically, the tuple space grows downwards while the offset vector grows upwards (figure 2.a).

Consider the query: "find the $CO_2$ consumption history of product $A$" and assume that the NSM page of figure 2.a is already in main memory and that the cache line size is smaller than the tuple size. As shown in figure 2.b, to execute the query, the page header and the offset vector are first loaded in the cache in order to locate product $A$ tuples (cache lines 1 and 2). Next, each $A$ tuple is loaded in the cache (cache lines 3 to 6). Product $A$ name and price, which are useless for the query, are brought more than once in the cache, leading to the waste of main memory bandwidth, L2 cache space and CPU cycles.

## 2.2 Decomposition Storage Model

An alternative storage model to NSM is the *Decomposition Storage Model* (DSM) [6], also called *column-store architecture* [16]. As illustrated in figure 1.b, DSM partitions vertically a relation $R$ with arity $n$, into $n$ sub-relations. Each sub-relation holds: ($i$) the values of an attribute of $R$; and ($ii$) the tuple surrogates identifying the original tuples that the values came from. The trade-offs between DSM and NSM are still being explored [10, 1]. The two most cited strengths of DSM are: ($i$) *improved memory hierarchy utilization*: with DSM, a DBMS needs only read the values of attributes required for processing a given query [2]; ($ii$) *improved data compression* [16]: as the values of an attribute are stored contiguously, DSM enables further compression opportunities.

The most cited drawbacks of DSM are: ($i$) *increased seek time*: "disk seeks between each read might be needed as multiple attributes are read in parallel" [1]; ($ii$) *increased cost of insertions*: DSM performs poorly for insertions because multiple distinct pages have to be updated for each inserted tuple [1]; and ($iii$) *increased tuple reconstruction cost*: for queries involving several attributes, DSM needs to join the participating sub-relations together [2]. In addition to the drawbacks aforementioned, using DSM for temporal data does not avoid version redundancy.

# 3. READ-OPTIMIZED PAGE LAYOUTS

## 3.1 Temporal Decomposition Storage Model

### 3.1.1 Principle

TDSM is a temporal extension of DSM. As illustrated in figure 1.c, TDSM does not store the timestamp attribute in a separate sub-relation as in the straight-forward DSM. Rather, the timestamp attribute is stored with each of the other attributes. With this approach, TDSM has the following advantages when compared to DSM: ($i$) TDSM avoids version redundancy and hence improves memory hierarchy utilization; and ($ii$) TDSM reduces the insertion cost when the attributes of an entity are updated, because only the pages storing the updated values are modified.

For queries involving several attributes, TDSM needs to join the participating sub-relations as in DSM. However, unlike DSM, where equi-joins on tuple surrogate are performed, TDSM joins two tuples only if their entity surrogates are equal and their lifespans intersect (*i.e. Temporal Equi-join* [7, 8]). Such a temporal equi-join is known to be more expensive to process than a conventional equi-join [7].

At the current state of TDSM implementation, we use an indexed join scheme to reduce tuple reconstruction cost. With this approach, each sub-relation is implemented as a clustering Time-Split B-tree (TSB-tree) [12] and a slightly modified merge join is used to connect sub-relations tuples selected by the TSB-trees. Obviously, more elaborate join techniques could be used [7, 8]. However, as demonstrated in [7], the adopted approach provides a reasonable simplicity-efficiency tradeoff. The following subsection reviews the TSB-tree and details tuple reconstruction in TDSM.

### 3.1.2 TSB-Tree and Tuple Reconstruction

The TSB-tree is a variant of the B+tree. Leaf nodes contain data and are called *data pages*. Non-leaf nodes, called *index pages*, direct search from the root and contain only

**Input:** $L_1$, $L_2$ {Two sorted list of resp. $n_1$ and $n_2$ tuples; each tuple $T_1$ (resp. $T_2$) of $L_1$ (resp. $L_2$) has an entity surrogate $e$, a timestamp $t$ and an attribute $a_1$ (resp. $a_2$).}
**Output:** $L_r$ {List of tuples resulting from the merge join of $L_1$ and $L_2$. Each tuple $T_r$ of $L_r$ has an entity surrogate, a timestamp and two attributes $a_1$ and $a_2$.}

```
i ← 0;   j ← 0;
while i < n1 or j < n2 do
  if i < n1 and j < n2 then
    T1 ← L1[i];      T2 ← L2[j];
    if T1.e = T2.e then
      Tr.e ← T1.e;
      if T1.t = T2.t then
        Tr.t ← T1.t;   Tr.a1 ← T1.a1;   Tr.a2 ← T2.a2;
        i++;      j++;
      else if T1.t < T2.t then
        Tr.t ← T1.t;   Tr.a1 ← T1.a1;   i++;
        {Tr.a2 keeps its old value}
      else
        Tr.t ← T2.t;   Tr.a2 ← T2.a2;   j++;
        {Tr.a1 keeps its old value}
      end if
    else if T1.e < T2.e then
      Tr.e ← T1.e;   Tr.t ← T1.t;   Tr.a1 ← T1.a1;   i++;
    else
      Tr.e ← T2.e;   Tr.t ← T2.t;   Tr.a2 ← T2.a2;   j++;
    end if
  else if i < n1 then
    T1 ← L1[i];   Tr.t ← T1.t;   Tr.a1 ← T1.a1;   i++;
  else
    T2 ← L2[j];   Tr.t ← T2.t;   Tr.a2 ← T2.a2;   j++;
  end if
  Lr.push_back(Tr);
end while
return Lr;
```

**Figure 4: Merge Join in TDSM**

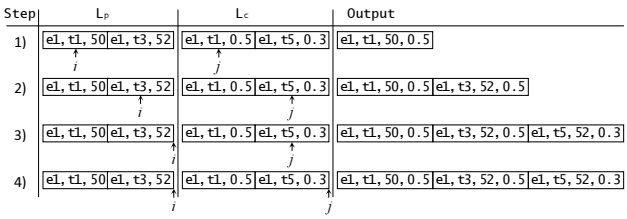



**Figure 5: Merging of two sorted temporal lists**

search information. TSB-tree pages at a given level partition the surrogate-time space. An entry of an index page is a triple ($[e_{min}, e_{max})$, $[t_{start}, t_{end})$, $I$), where $[e_{min}, e_{max})$ is a surrogate interval, $[t_{start}, t_{end})$ is a time interval and $I$ the identifier of a child page. Such entry indicates that the data pages of the subtree rooted at $I$ contain tuples ($e$,$t$), such that $e \in [e_{min}, e_{max})$ and $t \in [t_{start}, t_{end})$.

Tuples within a data page are ordered by entity surrogate and then by timestamp. If the insertion of a tuple causes a data page overflow, the TSB-tree uses either, *time split*, *surrogate split* or a combination of both. A surrogate split occurs when the overflowing data page only contains current tuples (*i.e.* tuples alive at the current time). It is similar to a split in a B+tree: tuples with surrogate greater than or equal to the split surrogate are moved to the newly allocated data page. A time split occurs when the overflowing data page, $D$, contains both current and historical tuples. The time split of $D$ separates its tuples according to the current time $t$: (1) a new data page $D'$ with time interval $[t, +\infty)$ is allocated; (2) tuples of $D$ valid at $t$ are copied in $D'$ (figure 3.b). After a time split, if the number of tuples copied in $D'$ exceeds a threshold $\theta$, a surrogate split of $D'$ is performed. An index page split is similar to a data page split.

Consider the query "find the price and the $CO_2$ consumption history of product $A$" and assume that sub-relations

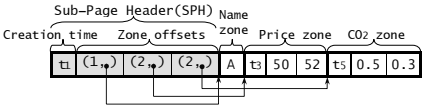



**Figure 6: Sub-page layout**

price and $CO_2$ are, respectively indexed, by $TSB_p$ and $TSB_c$. The query is processed as follows. First, $TSB_p$ and $TSB_c$ are searched in order to locate product $A$ prices and $CO_2$ consumptions. Two list of tuples, $L_p$ and $L_c$, sorted on entity surrogate and on timestamp, are created to hold tuples respectively selected by $TSB_p$ and $TSB_c$. Finally, $L_p$ and $L_c$ are merged. The algorithm of merge join used in TDSM is shown in figure 4. Figure 5 illustrates the merge process.

## 3.2 Per Surrogate Partitioning Model

As illustrated in figure 1.d, the goal of PSP is: ($i$) to store each information only once; and ($ii$) to allow easy tuple reconstructions. In order to allow easy tuple reconstructions, PSP keeps all the attribute values of a tuple in the same page, as in NSM. Unlike NSM, PSP organizes attribute values within a page, so that, version redundancy is avoided.

Within a page, PSP packs tuples into *sub-pages*, so that tuples of distinct sub-pages have distinct entity surrogates and tuples of any sub-page have the same entity surrogate. Thus, a sub-page records the history of an entity. Within a sub-page, to be able to avoid version redundancy, PSP packs the values of each attribute contiguously in an *attribute zone*. The remainder of this section details the design of PSP.

### 3.2.1 Attribute Zone

Let $s$ be a sub-page recording the history of an entity $e$. An attribute zone is an area within $s$, storing the history of an attribute $a$ of $e$: the values taken by $a$ over time. For instance, in figure 6, the price zone of product $A$, stores together its successive amounts. Each zone of an attribute $a$ is prefaced by a timestamp vector holding the timestamps of updates on $a$. The values of $a$ are put at the end of the timestamp vector in the same order as timestamps. When a variable-length attribute is also time-varying, its values are preceded by an offset vector.

Let $t_c$ be the lowest timestamp identifying a tuple recording a state of an entity $e$; $t_c$ is called the *creation time* of $e$: *e.g.* in figure 1.a the creation time of entity $e_1$ is $t_1$. To avoid redundancy, $t_c$ is not stored in the timestamp vector of each attribute zone; rather, $t_c$ is stored only once at the sub-page level. Thus, if an attribute is time-invariant, the timestamp vector of its attribute zone is empty. For example, in figure 6, the timestamp vector of the name zone of product $A$ is empty.

### 3.2.2 Sub-Page and Page Layouts

As shown in figure 6, each sub-page corresponding to an entity $e$ is preceded by a *Sub-Page Header* (SPH) containing: the creation time of $e$ and a vector of pairs ($v$,$z$), where $z$ is the starting offset of the zone of an attribute $a$ of $e$ and $v$ is the number of $a$ distinct values.

As an entity may have tens or even hundreds of versions, a vector of sub-page offsets in PSP is expected to be much smaller than a vector of tuple offsets in NSM. In addition, the page header size is typically smaller than an L2 cache line size. Thus, for PSP it makes sense to store the page header and the offset vector contiguously, so that, loading the page

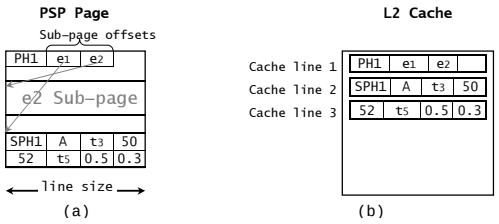



**Figure 7: (a) PSP page layout. (b) Cache behavior for "find the $CO_2$ consumption history of $A$".**

header also loads the offset vector or a large part of it. As illustrated in figure 7.a, in a PSP page, the sub-page space grows upwards while the offset vector grows downwards.

Situations occur where the whole history of an entity does not fit in a single page, *i.e.* a sub-page is too *large* to fit in a single page. PSP copes with this large sub-page problem, as follows. Let $s$ be a large sub-page storing the history of an entity $e$ and $t$ be the time of the last update of $e$. The solution consists in creating a new sub-page $s'$ and initializing it with the version of $e$ valid at $t$. This solution introduces some (limited) redundancy but has an important advantage: the search for $e$ tuples alive at a timestamp $t_i$, such that $t_i \geq t$, is only performed within $s'$ and the search for $e$ tuples alive at a timestamp $t_j$, such that $t_j < t$, is only performed within $s$.

## 3.3 Discussion

Consider the query "find the $CO_2$ consumption history of product $A$". As shown in figure 7.b, PSP improves the cache space and the main memory bandwidth consumed by this query, because it avoids fetching the same value several times (as opposed to NSM and DSM). In addition, PSP improves the data spatial locality, because the requested values are stored contiguously. PSP also requires less storage space than NSM, because: $(i)$ it stores unchanged values only once; and $(ii)$ it factorizes common entity metadata, whereas NSM stores a header for each tuple. Thus, PSP also improves disk bandwidth and main memory space utilization, as a PSP page is expected to contain more informations than the corresponding NSM page.

In case of time-invariant data, each attribute zone within a PSP page stores a single value and has an empty timestamp vector. Thus, in such case a PSP sub-page has a layout similar to a typical tuple format in NSM. As a result, a PSP page and an NSM page have similar layouts and behaviors when used for non temporal data.

For queries involving several attributes, PSP only needs to perform joins among attribute zones stored contiguously within a single sub-page (as opposed to DSM and TDSM).

# 4. PERFORMANCE EVALUATION

This section compares the performance of the different storage models. For vertical decomposition, as TDSM is expected to outperform the straight-forward DSM and due to the lack of space, only TDSM is considered.

NSM and DSM systems often use their own sets of query techniques that can provide additional performance improvements [10]. As this paper only focuses on the differences between NSM, TDSM and PSP related to the way data are stored in pages, we have implemented a TDSM, an NSM and a PSP storage managers, in C++ from scratch (our code is compiled using GCC). The performance of these storage managers are measured with identical datasets and query workloads, generated following the specifications of the cost models presented in [18, 11]. To provide a fair comparison, the implmented storage managers use clustering TSB-trees.

## 4.1 Workload and Assumptions

The cost models proposed in [11, 18] model a temporal relation $R$ by a set of $E$ entities and $T$ timestamps. Each entity $e$ is subject to updates; each update occurring at timestamp $t$, generates a new entity version $(e,t)$, whose value is recorded in a tuple of $R$. The proportion $\delta$ of entities updated at each timestamp, called *data agility* [18], is assumed to be constant. Thus, the total number of tuples in $R$ is: $E + \delta E(T-1)$. $R$ is assumed to be indexed by TSB-trees, using NSM, TDSM (one TSB-tree per attribute) or PSP. The goal is to evaluate the storage, insertion and query costs. The storage cost is measured by the number of occupied data pages. The insertion cost is measured by the average time elapsed during the insertion of a tuple. The cost of a query $q$ is measured by the following parameters: $(i)$ the average number of data pages loaded in main memory; $(ii)$ the execution time when data are fetched from disk; $(iii)$ the average number of L2 cache misses when the requested pages are main-memory resident; and $(iv)$ the execution time when the requested pages are main-memory resident. To be as general as possible, we follow [18] and assume that a temporal query $q$ has the following form:

**select** $q_a$ attributes **from** $R$ **where** $e \in [e_i, e_j)$ **and** $t \in [t_k, t_l)$

where $q_a$ is the number of involved time-varying attributes, $[e_i, e_j)$ an interval of entity surrogates containing $q_s$ surrogates and $[t_k, t_l)$ a time interval containing $q_t$ timestamps.

## 4.2 Settings and Measurement Tools

A large number of simulations have been performed to compare NSM, TDSM and PSP. However, due to the lack of space, only few results are presented herein. For the presented simulations, data are generated as follows. A temporal relation $R$ is assumed to have ten 4-byte numeric attributes, in addition to an entity surrogate and a timestamp. Four attributes of $R$ are time-varying. Time-varying attributes are assumed to vary independently over time, with the same agility. $E$ and $T$ are respectively set to 200K entities and 200 timestamps. At the first timestamp, 200K tuples are inserted in $R$ (one tuple per entity). Then, at each of the following 199 timestamps, $\delta E$ entities, randomly selected, are updated. The data agility $\delta$ is varied in order to obtain different temporal relations. For example, if $\delta = 15\%$, $R$ contains 6.17 millions tuples ($200K + 200K \times 15\% \times (200-1)$=6.17 millions).

Simulations are performed on a dual core 2.80 GHz Pentium D system, running Windows 2003 Server. This computer features 1GB main memory (DDR2-667MHz), 800 MHz Front Side Bus, and $2 \times 2$ MB L2 cache. The cache line size is 64B. The storage managers were configured to use a 8KB page size. The execution time is measured by function *QueryPerformanceCounter* provided by the API Win32. L2 cache events are collected using Intel VTune.

## 4.3 Results

*Storage Cost.* Figure 8 depicts the storage costs as function of data agility. PSP requires up to $\approx$ 7.4 times less storage space than NSM and on average $\approx$ 2 times less storage space than TDSM. The superiority of PSP and TDSM against NSM increases as the agility increases, because, the

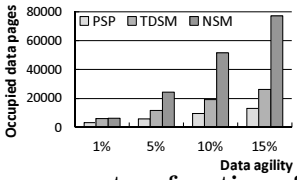



**Figure 8: Storage cost as function of data agility**

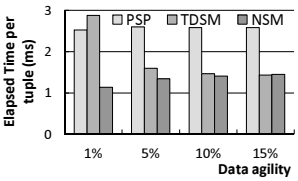



**Figure 9: Insertions: average elapsed time per tuple as function of data agility**

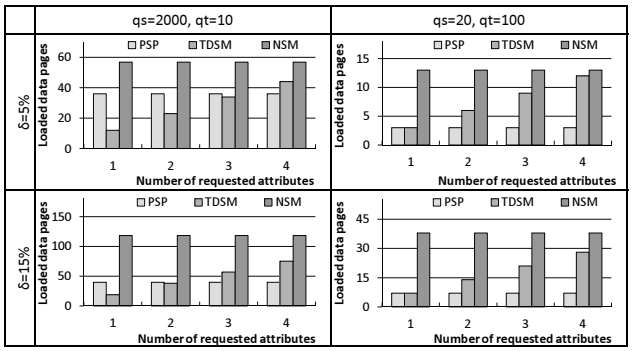



**Figure 10: Queries: average loaded disk pages**

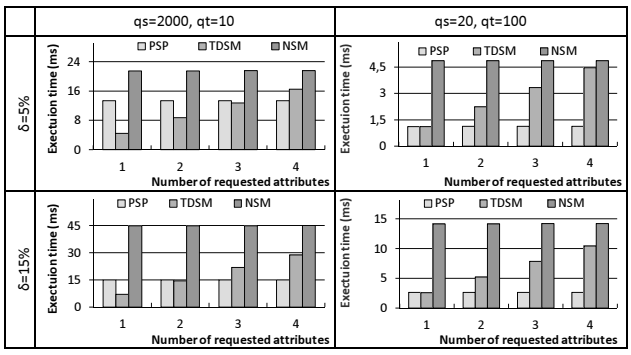



**Figure 11: Queries: average execution time when data are fetched from disk**

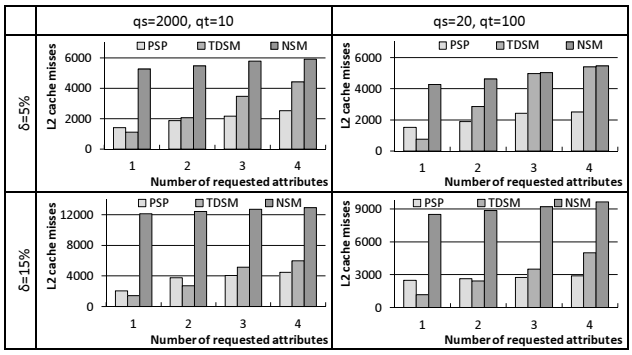



**Figure 12: Queries: average L2 cache misses when data are fetched from main memory**

larger is the agility and the larger is the number of tuples per entity, hence, the larger is the disk space saved by PSP and TDSM.

*Insertion Cost.* To provide fair comparison of insertion costs, we assume that all data requests are served from disk. Figure 9 depicts the insertion costs as function of data agility. Insertions in NSM are on average $\approx 2$ times faster than in PSP. The main reason is that with NSM a single write suffices to push all the attributes of a tuple, while with PSP, an additional effort is needed for sub-page reorganizations. Although TDSM performance are penalized by the fact that the operating system scheduler handles write requests for multiple relations, TDSM exhibits good performance. Note that TDSM performance should be worst if more than one attribute is updated at a time.

*Query Cost.* To evaluate the query cost, we consider a moderately agile temporal relation, $\delta = 5\%$, and a highly agile temporal relation, $\delta = 15\%$. For each temporal relation, eight query workloads are considered; each one consisting of 100 queries. Queries in a workload involve the same of number of consecutive entity surrogates, $q_s$, the same number of consecutive timestamps, $q_t$, and the same number of temporal attributes, $q_a$. The surrogate intervals and the time intervals of queries of a given workload are uniformally distributed in the surrogate-timestamp space. The reported results for a given workload are the average of performance achieved by the 100 queries composing it. The first four query workloads involve a relatively large surrogate interval, $q_s = 2000$, a small time interval, $q_t = 10$, and different numbers of time-varying attributes: $q_a$ varies from 1 to 4. The latter ones involve a small $q_s = 20$, a relatively large $q_t = 100$, and different $q_a$: $q_a$ varies from 1 to 4.

Figure 10 depicts the average numbers of data pages loaded in main memory to answer the query workloads described above. As expected, TDSM outperforms PSP and NSM when a single attributes is involved ($q_a = 1$). In all cases, TDSM and PSP outperform NSM. Figure 11 depicts the average query execution time when the requested pages are fetched from disk. As expected TDSM and PSP outperform NSM in all casses: TDSM is on average $\approx 2.74$ times faster than NSM; PSP is on average $\approx 3.6$ times faster than NSM. Figure 12 depicts the average numbers of L2 cache misses, when the requested pages are main-memory resident. As shown in figure 12, in all cases, PSP and TDSM generate less L2 cache misses than NSM. In particular, PSP generates up to 9 times less L2 cache misses than NSM. Figure 13 depicts the average execution time, when the requested

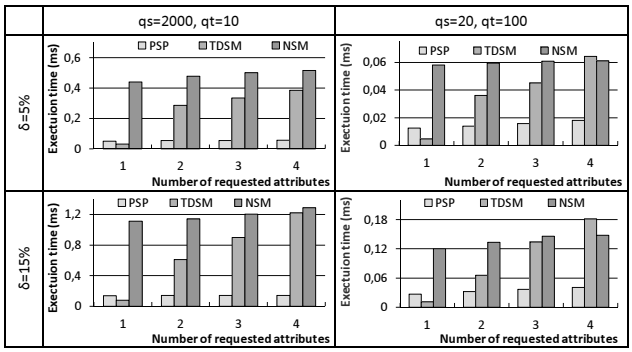



**Figure 13: Queries: average execution time when data are fetched from main memory**

pages are main-memory resident. TDSM has a better behavior than PSP when a single attribute is involved. However, TDSM performance deteriorates very quickly as $q_a$, the number of involved attributes in a query, increases. This is due to temporal joins. As shown in figure 13, PSP is in general faster than either NSM and TDSM: on average $\approx 6.54$ times faster than NSM and $\approx 3.66$ faster than TDSM, when data are already in main memory.

# 5. RELATED WORK

Several approaches have been proposed in order to achieve high performance for read operations. In [9], [2] and [15] cache-conscious page layouts have been proposed, namely Data Morphing, Partition Attributes Across (PAX) and Clotho. Among these, PAX is closest to PSP in design philosophy. Given a relation $R$ with arity $n$, PAX partitions each page into $n$ *minipages*. The $i^{th}$ minipage stores all the values of the $i^{th}$ attribute of $R$. PAX provides a high degree of spatial locality for sequential access to values of one attribute. Nevertheless, PAX stores data in entry sequence (probably to achieve good performance for updates). Thus, using PAX for temporal data does not avoid version redundancy.

A number of academic and commercial read-optimized systems implement DSM: Sybase IQ [20], Fractured Mirrors [14], Monet [5], C-Store [16, 21], etc.. These systems reduce the tuple reconstruction cost of DSM using techniques such as join indexes and chunk-based reconstructions [14, 16]. However, except C-Store and Fractured Mirrors, as they store data in entry sequence order, their performance suffer from the same problems as NSM.

Fractured Mirrors [14] stores two copies of a relation, one using DSM and the other using NSM. The read requests generated during query execution are appropriately scheduled between mirrors. With this approach, Fractured Mirrors provides better query performance than either storage model can provide separately. However, as Fractured Mirrors have not been designed to handle temporal data the problem of version redundancy has not been considered.

C-Store [16] implements a relation as a collection of materialized overlapping projections. To achieve high performance reads, C-Store: (*i*) sorts projections from the same relation on different attributes; (*ii*) allows a projection from a given relation to contain any number of other attributes from other relations (*i.e.* pre-joins); (*iii*) stores projections using DSM; and (*iv*) uses join indexes to reconstruct tuples. The implementation of C-Store implicitly assumes that projections from the same relation have the same number of tuples. Thus, C-Store is unable to avoid version redundancy.

# 6. CONCLUSION

This paper compares the conventional page layout, NSM, to TDSM and PSP, two read-optimized, cache-conscious, page layouts specifically tailored for temporal data. TDSM exhibits interesting features that need to be further explored. In particular, TDSM performance can be substantially improved if join techniques, more adapted to vertical decomposition than those commonly used for temporal data, are designed. PSP optimizes performance at all levels of the memory hierarchy: (*i*) it avoids version redundancy (as opposed to NSM and DSM); and (*ii*) it allows easy tuple reconstructions, (as opposed to vertical decomposition). In addition to the advantages aforementioned, PSP can be used for non temporal data with the same performance as NSM.

# 7. ACKNOWLEDGMENTS

We would like to thank Claudia Bauzer Medeiros for many helpful discussions and reviews that improved this paper.

# 8. REFERENCES

[1] D. Abadi. Column Stores for Wide and Sparse Data. In *CIDR*, pages 292–297, 2007.

[2] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[3] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999.

[4] E. Baykan. Recent Research on Database System Performance. Technical report, LBD-Ecole Polytechnique Fédérale de Lausanne, 2005.

[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[6] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In S. B. Navathe, editor, *SIGMOD*, pages 268–279. ACM, 1985.

[7] Z. Donghui, V. Tsotras, and B. Seeger. Efficient Temporal Join Processing Using Indices. In *ICDE*, page 103. IEEE Computer Society, 2002.

[8] D. Gao, S. Jensen, T. Snodgrass, and D. Soo. Join Operations in Temporal Databases. *The VLDB Journal*, 14(1):2–29, 2005.

[9] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, pages 417–428, 2003.

[10] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized DataBases. In *VLDB*, pages 487–498, 2006.

[11] K. Jouini and G. Jomier. Indexing Multiversion DataBases. In *CIKM*, pages 915–918. ACM, 2007.

[12] D. B. Lomet and B. Salzberg. The Performance of a Multiversion Access Method. In *SIGMOD*, pages 353–363. ACM, May 1990.

[13] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, chapter 3 Data Storage and Indexing. McGraw-Hill, 2000.

[14] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. *The VLDB J.*, 12(2):89–101, 2003.

[15] M. Shao, J. Schindler, S. W. Schlosser, and Al. Clotho: Decoupling Memory Page Layout from Storage Organization. In *VLDB*, pages 696–707, 2004.

[16] M. Stonebraker, D. J. Abadi, A. Batkin, and Al. C-store: A Column-Oriented DBMS. In *VLDB*, pages 553–564, 2005.

[17] M. Stonebraker, S. Madden, D. Abadi, and Al. The End of an Architectural Era: (it's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[18] Y. Tao, D. Papadias, and J. Zhang. Cost Models for Overlapping and Multiversion structures. *TODS*, 27(3):299–342, 2002.

[19] www.sensage.com/.

[20] www.sybase.com/products/datawarehousing/sybaseiq.

[21] www.vertica.com/v zone/product_documentation.

# DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing

Marcin Zukowski Niels Nes Peter Boncz

CWI, Amsterdam, The Netherlands
{Firstname.Lastname}@cwi.nl

## ABSTRACT

Comparisons between the merits of row-wise storage (NSM) and columnar storage (DSM) are typically made with respect to the persistent storage layer of database systems. In this paper, however, we focus on the CPU efficiency tradeoffs of tuple representations inside the query execution engine, while tuples flow through a processing pipeline. We analyze the performance in the context of query engines using so-called "block-oriented" processing – a recently popularized technique that can strongly improve the CPU efficiency. With this high efficiency, the performance trade-offs between NSM and DSM can have a decisive impact on the query execution performance, as we demonstrate using both microbenchmarks and TPC-H query 1. This means that NSM-based database systems can sometimes benefit from converting tuples into DSM on-the-fly, and vice versa.

## 1. INTRODUCTION

As computer architecture evolves, and the "make the common case fast" rule is applied to more and more CPU features, the efficiency of an application program can no longer be measured by the number of instructions it executes, as instruction throughput can vary enormously due to many factors, among which: *(i)* CPU cache and TLB miss ratio, resulting from the data access patterns; *(ii)* the possibility of using SIMD operations (e.g. SSE) to process multiple data items with one instruction; *(iii)* the average amount of in-flight instructions unbound by code- or data-dependencies, thus available to keep the instruction pipelines filled.

While such factors and their (significant) impact on performance may be well-understood, even in the specific context of data management tasks, and a range of so-called *architecture-conscious* query processing algorithms has been proposed, our goal is to investigate how such ideas can be integrated in real database systems. Therefore, we study how architectural-conscious insights can be integrated into the (typical) architecture of query engines.

The central question addressed in this research is how tuple layout in a *block-oriented* query processor impacts performance. This work is presented from the context of the MonetDB/X100 prototype [5], developed at CWI. MonetDB/X100 uses the standard open-next-close iterator execution model, but its most notable characteristic is the pervasive use of block-oriented processing [15, 5], under the moniker "vectorized execution". In block-oriented processing, rather than processing a single tuple per next() call, in each iteration the operator returns a *block* of tuples. This block can contain from a few tens to hundreds of tuples, thereby striking middle ground between tuple-at-a-time processing and full table materialization. Typically, performance increases with increasing block size, as long as the the cumulative size of tuple-blocks flowing between the operators in a query plan fits in the CPU cache. The main advantage of block-oriented processing is a reduction in the amount of method calls (i.e., query interpretation overhead). Additional benefit comes from the fact that the lowest level *primitive* functions in the query engine now expose independent work on multiple tuples (arrays of tuples). This can help compiler and CPU – and sometimes the algorithm designer – to achieve higher efficiency at run-time.

While MonetDB/X100 is known as a column-store[1], our focus here is not persistent storage, rather the representation of tuples as they flow through a block-oriented query processing engine, which can be different from the storage format. In particular, we experiment with both horizontal tuple layout (NSM) and vertical layout (DSM) and also discuss indirect value addressing (to avoid tuple copying).

Our main research questions are: *(i)* what are the advantages and disadvantages of DSM and NSM for tuple representations during query execution? *(ii)* what specific opportunities and challenges arise when considering tuple layout in the context of block-oriented processing (SIMD, prefetching, block size)? *(iii)* can query executors be made to work on both representations, and allowed to (dynamically) switch between them, given that depending on the situation, and even depending on the query sub-expression, either DSM or NSM can be better?

### 1.1 Outline and Findings.

In Section 2 we first describe the NSM and DSM layouts considered. Section 3 starts with a number of microbenchmarks contrasting the behavior of DSM and NSM in sequential and random-access algorithms. DSM is significantly faster in sequential scenarios thanks to simpler ad-

[1]In fact, it also supports the hybrid PAX layout [4].


*Proceedings of the Fourth International Workshop on Data Management on New Hardware (DaMoN 2008),* June 13, 2008, Vancouver, Canada.

dressing, larger vector sizes fitting in the L2 cache, whereas the higher spatial locality of NSM makes it the method of choice when operators access memory areas larger than the L1 cache randomly. SIMD instructions give an advantage to DSM in all sequential operators such as Project and Select, whereas in Aggregation only NSM allows to exploit SIMD (in some cases). Therefore, ideally, a query processing engine should be able to operate on data in both formats, even allowing tuple blocks where some columns are in NSM, and others in DSM. Further micro-benchmarks demonstrate that thanks to block-oriented processing, converting NSM tuples into DSM (and vice versa) can be done with high efficiency.

The consequences of these findings can be startling: we show in case of TPC-H query 1, that systems with NSM storage turn out to benefit from converting tuples on-the-fly to DSM, *pulling up* the selection operator to achieve SIMDized expression calculation, then followed by conversion back into NSM, to exploit SIMD Aggregation.

We wrap up by discussing related work in Section 4 and outlining conclusions and future work in Section 5.

# 2. TUPLE REPRESENTATIONS

To analyze different aspects of the DSM and NSM data organization, for the experiments presented in this paper we try to isolate the actual data access functionality from unnecessary overheads. This is achieved with following the block-oriented execution model, and analyzing the computationally simplest DSM and NSM data representations, presented in this section.

## 2.1 DSM tuple-block representation

Traditionally, the Decomposed Storage Model [9] proposed for each attribute column to hold two columns: a *surrogate* (or *object-id*) column and a *value* column. Modern column-based systems [5, 16] choose to avoid the former column, and use the natural order for the tuple reorganization purposes. As a result, the table representation is a set of binary files, each containing consecutive values from a different attribute. This format is sometimes complicated e.g. by not storing NULL values and other forms of data compression [21, 1]. In this case, some systems keep the data compressed for some part of the execution [1], and some perform a fully-transparent decompression, providing a simple DSM structure for the query executor [21]. Here, we choose a straightforward DSM representation, with columns stored as simple arrays of values. This results in the following simple code to access a specific value in a block:

```
value = attribute[position];
```

## 2.2 Direct vs. Indirect Storage

Variable-width datatypes such as strings cannot be stored directly in arrays. A solution is to represent them as memory pointers into a heap. In MonetDB/X100, a tuple stream containing string values uses a list of heap buffers that contain concatenated, zero-separated strings. As soon as the last string in a buffer has left the query processing pipeline, the buffer can be reused.

Indirect storage can also be used to reduce value copying between the operators in a pipeline. For instance, in MonetDB/X100, the Select operator leaves all tuple-blocks from the data source operator intact, but just attaches an array

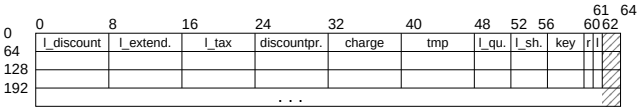



**Figure 1: Diagram of the access-time optimized NSM data organization during computation of TPC-H Query 1**

of selected offsets, called the *selection vector*. All primitive functions support this optional index array:

```
value = attribute[selection[position]];
```

Other copy-reduction mechanisms are also possible. For instance, MonetDB/X100 avoids copying result vectors altogether if an operator is known to leave them unchanged (i.e. columns that just pass through a Project or the left side of an N-1 Join). Note that the use of index arrays (selection vectors) is not limited to the Select operator. Other possibilities include e.g. not copying the build-relation values in a HashJoin, but instead storing references to them. In principle each column could have a different (or no) selection vector, which brings multiple optimization opportunities and challenges. In this paper, however, we focus on a simple, direct data storage.

## 2.3 NSM tuple-block representation.

Typically, database systems use some form of a slotted page for the NSM-stored tuples. The exact format of the tuples in this model can be highly complex, mostly due to storage considerations. For example, NULL values can be materialized or not, variable-width fields result in non-fixed attribute offsets, values can be stored explicitly or as references (e.g. dictionary compression or values from a hash table in a join result). Even fixed-width attributes can be stored using variable-width encoding, e.g. length encoding [17] or Microsoft's Vardecimal Storage Format [3].

Most of the described techniques have a goal of reducing the size of a tuple, which is crucial for disk-based data storage. Unfortunately, in many cases, such tuples are carried through into the query executor, making the data access and manipulation complex and hence expensive. In traditional tuple-at-a-time processing, the cost of accessing a value can be an acceptable compared to other overheads, but with block processing handling complex tuple representations can consume the majority of time.

To analyze the potential of NSM performance, we define a simple structure for holding NSM data, that results in a very fast access to NSM attributes. Figure 1 presents the layout of the tuples used for the processing of TPC-H Q1, visualized in Figure 4, and analyzed in Section 3.3. Tuples in a block are stored continuously one after another. As a result, tuple offset in a block is a result of the multiplication of the tuple width and its index. The attributes are stored in an order defined by their widths. Assuming attributes with widths of power of 2, this makes every value naturally aligned to its datatype within the tuple. Additionally, the tuple is aligned at the end to make its width a multiple of the widest stored attribute. This allows accessing a value of a given attribute at a given position with this simple code:

```
value = attribute[position * attributeMultiplier];
```

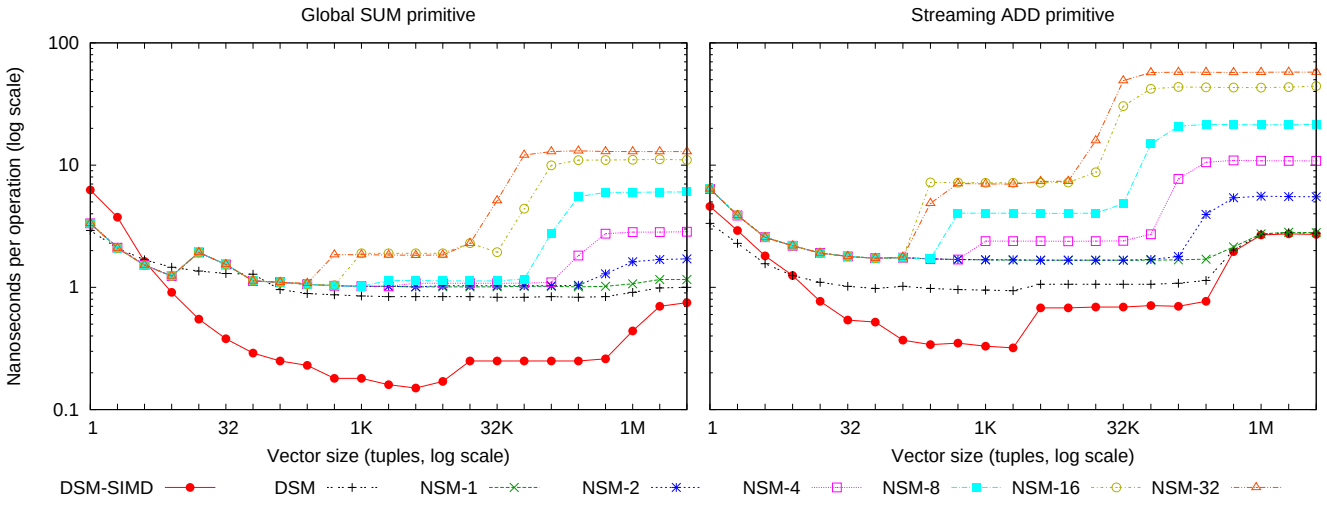



**Figure 2: Sequential access: performance of the SUM and ADD routines with DSM and NSM and varying tuple widths.**

# 3. EXPERIMENTS

In this section we analyze the performance of NSM and DSM data organization schemas on database performance. We start with a series of micro-benchmarks, presenting the baseline performance on some basic data access and manipulation activities. Then we demonstrate how these microbenchmark results are confirmed during the processing of the TPC-H Query 1. There, we also discuss some optimization techniques, that depend heavily on data organization, as well as on-the-fly data conversion.

## 3.1 Experimental setup

The experimental platform used is a Core2 Quad Q6600 2.4GHz with 8GB RAM running Linux with kernel 2.6.23-15. The per-core cache sizes are: 16KB L1 I-cache, 16KB L1 D-cache, 4MB L2 cache (shared among 2 cores). All experiments are single-core and in-memory. We used 2 compilers, GCC 4.1.2[2] and ICC 10.0[3].

We have performed similar experiments on a set of other CPUs: Athlon 64 X2 3800+ (2GHz), Itanium 2 and Sun Niagara. For Athlon and Itanium the results were mostly in line with the Core2 results. On Niagara the performance benefit of DSM was typically higher, and the impact of the data location was lower. This is caused by lower performance of Niagara in terms of sequential execution: it has a lower clock speed and in-order execution pipeline. Since Niagara was designed with multi-threaded processing in mind, it would be interesting to see how the presented, currently single-threaded, benchmarks perform when running in parallel. This might be a topic for future research.

## 3.2 Microbenchmarks

In this section we analyze the baseline performance of the DSM and NSM models in typical data-processing operations: sequential computations, random-access, and data copying.

### 3.2.1 *Sequential data access*

The left part of Figure 2 present the results of the experiment in which a SUM aggregate of a 4-byte integer column is computed repeatedly in a loop over a fixed dataset. The size of the data differs, to simulate different block sizes, which allows identifying the impact of the interpretation overhead, as well as the location (cache, memory) in block-oriented processing. We used GCC, using standard processing, and additionally ICC to generate SIMD-ized DSM code (NSM did not benefit from SIMD-ization). In the NSM implementation, we use tuples consisting of a varying number of integers, represented with *NSM-x*.

To analyze the impact of the data organization on CPU efficiency, we look at the performance of *NSM-1*, which has exactly the same memory access pattern and requirements as the DSM implementation. The GCC results show that for a single-integer table the performance of the DSM and NSM-1 is very close. The small benefit of DSM, ca. 15% in the optimal case, comes from the fact that thanks to a simpler data access the compiler is able to generate slightly more efficient code. However, with ICC-generated SIMD instructions, DSM is a clear winner, being almost 5 times faster in the optimal case. Note that SIMD can only be applied if the same operation is executed on adjacent memory locations, therefore it can only be used in DSM.

The other aspect of this benchmark is the impact of the interpretation overhead and data location. While for small block sizes the performance is dominated by the function calls[4], for larger sizes, when the data does not fit in the L1 cache anymore, the data location aspect becomes crucial. Performance of NSM-1 and DSM without SIMD is relatively flat, since even for main-memory sized data (1M+ tuples), the sequential bandwidth is close enough to balance the CPU activity. However, with the highly efficient (sub-cycle cost) SIMD DSM implementation, it operates fastest while the block still fits in the L1 CPU cache, then goes to an intermediate plateau when its fits L2, to become mem-

[2]compilation: `gcc -O6 -Wall -g -mtune=core2`
[3]compilation: `icc -O3 -Wall -axT`

[4]In a real DBMS, function call overhead is significantly larger [5] – this was a hard-coded micro-benchmark.

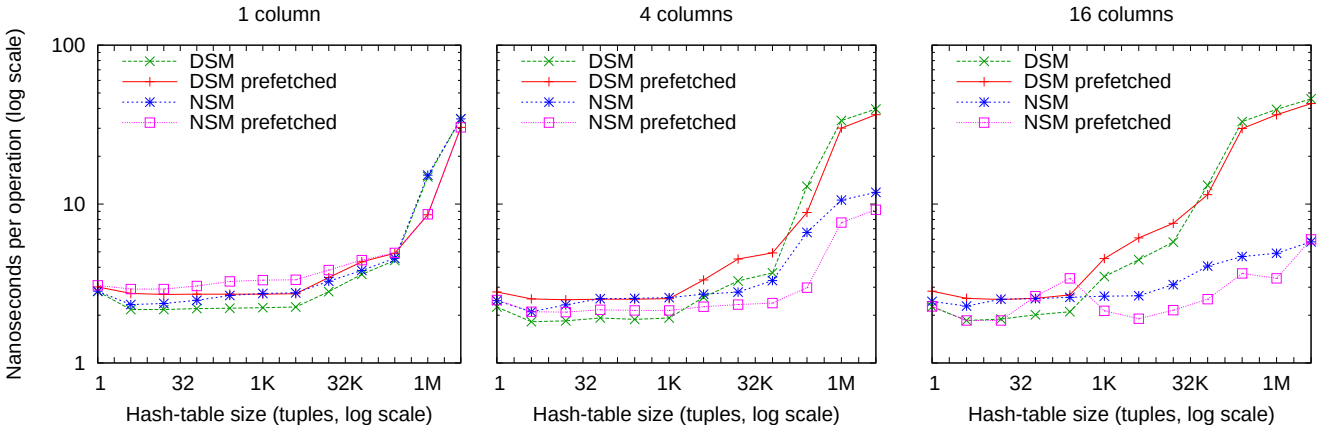



**Figure 3: Random access: running a grouped SUM aggregate on DSM input data, using a DSM or NSM hash table, with or without prefetching, and a varying number of GROUP BY keys (X-axis)**

ory bandwidth limited for larger sizes (i.e. L1 bandwidth exceeds L2 bandwidth which exceeds RAM bandwidth).

Looking at the performance of wider NSM tuples, we see that the performance degrades with the increasing tuple width. As long as the tuples are in L1, all widths are roughly equal. However, for NSM-16 and higher (64 byte tuples or longer) once the data shifts to L2, the impact is immediately visible. This is caused by the fact, that only a single integer from the entire cache-line is used. For NSM-2 to NSM-8, the results show that the execution is limited by the L2 bandwidth: when a small fraction of a cache-line is used (e.g. NSM-8) the performance is worse than when more integers are touched (e.g. NSM-2). Similar behavior can be observed for the main-memory datasets.

The SUM primitive has a relatively low memory demand compared to the CPU activity, as it only consumes a single attribute. The right part of Figure 2 presents a similar experiment, that uses an ADD routine which consumes two attributes and produces a new result attribute. Results follow the trends of the SUM operation, but there are some important differences. First, the higher number of parameters passed to the NSM routine (pointers + tuple widths VS only pointers) results in a higher interpretation overhead. Secondly, comparing DSM and NSM-1 for L1-resident data, shows that multiple more complex value-access computations in NSM have a higher impact on the CPU performance. Finally, with a higher memory demand, the impact of data locality on performance is significantly bigger, making even the DSM implementation fully memory-bound and in par with the NSM-1 version.

Concluding, we see that if access is purely sequential, DSM outperforms NSM for multiple reasons. First, the array-based structure allows simple value-access code. Second, individual primitive functions (e.g. SUM,ADD) use cache lines fully in DSM, and L2 bandwidth is enough to keep up. As mentioned before, during query processing, all tuple blocks in use in a query plan should fit the CPU cache. If the target for this is L2, this means significantly larger block sizes than if it were L1, resulting in reduced function call overhead. Finally, the difference in sequential processing between DSM and NSM can be huge if the operation is expressible in SIMD, especially when the blocks fit in L1, and is still significant when in L2.

### 3.2.2 *Random data access*

For this experiment, we use a table that consists of a single *key* column and multiple *data* columns. The table contains 4M tuples, is stored in DSM for efficient sequential access, number of *data* columns varies, and the range of the *key* column differs from 1 to 4M. We perform an experiment equivalent to this SQL query:

```
SELECT SUM(data1), ..., SUM(dataN)
FROM TABLE GROUP BY key;
```

To store the aggregate results, we use an equivalent of a hash-table in the hash-aggregation, but instead of the hash-value processing, we use the *key* column as a direct index. In DSM, the result table it is just a collection of arrays, one for each *data* attribute. In NSM, it is a single array of a size equal to the number of tuples multiplied by the number of *data* attributes. We apply block-oriented processing, using a block size of 256 tuples. In each iteration, all values from different *data* attributes are added to the respective aggregates, stored at the same vertical position in the table.

Figure 3 presents the results of this experiment for 1, 4 and 16 *data* columns. For a single column, the faster access code of the DSM version makes it slightly (up to 10%) faster than NSM as long as the aggregate table fits in the L1 cache. Once it enters L2 or main memory, the results of DSM and NSM are equal as they are memory-latency limited.

For wider tuples, DSM maintains its advantage for L1-based *key* ranges. However, once the data expands into L2 or main-memory, the performance of DSM becomes significantly worse than that of NSM. This is caused by the fact, that in DSM every memory access is expected to cause a cache-miss. In contrast, in NSM, it can be expected that a cache-line accessed during processing of one *data* column, will be accessed again with the next *data* column in the same block, as all the columns use the same key position. With the increasing number of computed aggregates, the same cache-line is accessed more often, benefiting NSM over DSM.

Figure 3 also shows experiments that use software prefetching, that is, we interspersed SUM computations with explicit prefetch instructions on the next tuple block. On Core2 we used the `prefetcht0` instruction. We also made sure the ag-

gregate table was stored using large TLB pages, to minimize TLB misses. In general, our experience with prefetching is that it is highly sensitive to the platform, and prefetch distances are hard to get right in practice. The end result is that prefetching does improve NSM performance when the aggregate table exceeds the CPU caches, however in contrast to [8] we could not obtain a straight performance line (i.e. hide all memory latency).

These simple random and sequential DSM vs. NSM micro benchmarks echo the debate on cache-conscious join and aggregation between partitioning and prefetching. In partitioning [7], the randomly accessed (e.g. aggregate) table is partitioned into chunks that fit the L1 cache. This table can be stored in DSM and probed very efficiently. The disadvantage of this approach is the cost of partitioning, possibly needing multiple sequential passes to achieve a good memory access pattern. The alternative is to have a NSM hash table exceed the CPU cache sizes, and pay a cache miss for each random probe. Unlike DSM, where random access generates a huge bandwidth need that cannot be sustained using prefetching, random probing in NSM benefits from prefetching.

In the following, we study on-the-fly conversion between DSM and NSM tuples. Using tuple conversion, it would e.g. become possible for a DSM-based system like MonetDB/X100 to use NSM (and prefetching) inside certain random-access query processing operators.

### 3.2.3 Data conversion

Many column stores use a traditional query processor based on NSM, calling for on-the-fly format conversion during the Scan operator. In case of C-Store [16][5], this is done using (slow) tuple-at-a-time conversion logic. We rather perform conversion using block-oriented processing, avoiding loop and function call overhead, where a single function call copies all values from one column in a block of NSM tuples into DSM representation (and vice versa):

```
NSM2DSM(int input[n], int width) : output[n]
 for(pos=0; pos<n; pos++]
  output[pos] = input[pos * width]
```

We performed micro-benchmarks, in which an NSM/DSM layout conversion is performed for datatypes of different widths. Table 1 shows that this can be done very efficiently, typically below 1 nanosecond per data value (ca. 2 CPU cycles on our test machine).

Therefore, given the different strengths and weaknesses of DSM and NSM, it becomes conceivable for a query optimizer to select the most appropriate storage format for certain sub-expressions in the query plan, and insert conversion operators to change the representation on-the-fly, potentially even multiple times. This could even lead to a situation where a query processing operator gets some input columns in DSM, and some in NSM (and the same for produced columns), similar as the persistent data is organized in the data-morphing technique [12].

We also measured the performance of copying a full NSM tuple into a different NSM representation. Such situation can occur e.g. during the merge join, where rows from both inputs need to be combined[6]. In this situation there are two choices: value-by-value copying and full-tuple copying (e.g. with `memcpy` equivalent). In tuple-at-a-time processing, the first choice will be typically significantly slower, due to high overheads of function calls and attribute-list iteration. However, Table 1 shows that in block-oriented processing, with the overheads amortized over a set of tuples, value-by-value copying can be very efficient. Full tuple copying, while fast, still suffers from overheads, as seen with a minimal performance difference between copying 8-byte and 128-byte wide tuples. As a result, for many types of tuples, attribute-by-attribute copying can be more efficient. This is especially useful, if copying includes only a subset of attributes, or if the field order in the result tuple needs to be different than in the source.

| Data unit | conversion speed (ns / operation) | | |
|---|---|---|---|
| | NSM⇒DSM | DSM⇒NSM | NSM⇒NSM |
| 8-byte tuple, block size 1024 | | | |
| 1-byte char | 0.85 | 0.85 | 0.65 |
| 4-byte int | 0.74 | 0.66 | 1.06 |
| full tuple | - | - | 8.42 |
| 16-byte tuple, block size 512 | | | |
| 1-byte char | 0.86 | 0.87 | 0.67 |
| 4-byte int | 0.82 | 0.65 | 1.07 |
| full tuple | - | - | 9.62 |
| 32-byte tuple, block size 512 | | | |
| 1-byte char | 0.93 | 0.87 | 0.72 |
| 4-byte int | 0.79 | 0.71 | 1.11 |
| full tuple | - | - | 9.76 |
| 64-byte tuple, block size 256 | | | |
| 1-byte char | 0.90 | 0.90 | 1.57 |
| 4-byte int | 0.89 | 0.74 | 1.62 |
| full tuple | - | - | 10.09 |
| 128-byte tuple, block size 128 | | | |
| 1-byte char | 0.97 | 0.90 | 1.43 |
| 4-byte int | 0.87 | 0.76 | 1.45 |
| full tuple | - | - | 13.17 |

**Table 1: Data conversion speed for different tuple widths and different conversion units. For each tuple width, the best block size was chosen.**

## 3.3 TPC-H Query 1

To see the impact of data organization in a more realistic scenario, we have evaluated the performance of the TPC-H Query 1 with different settings. A sketch containing the main primitives used in its query plan is presented in Figure 4. For simplicity, it does not include selection computation, connections between `count` and `sum` primitives to the aggregates table, and post-processing of the aggregate results. As Figure 4 shows, the computation in Query 1 consists mostly of 2 phases: sequential computation of input for aggregation, and random-access computation of aggregates. The microbenchmarks presented above suggest, that the best data organization for the first phase is DSM, and for the second phase it is NSM.

In this query plan we exploit the fact that `l_returnflag` and `l_linestatus` are `char` datatypes. This makes the possible key combinations limited to 65536 values (in fact, there are only 4 used). In this situation, instead of following the

[5]see `Operators/TupleGenerator.cpp` in C-Store 0.2

[6]Naturally, with more complex tuple representation the copying can be avoided, but we assume simple (hence fast) NSM organization

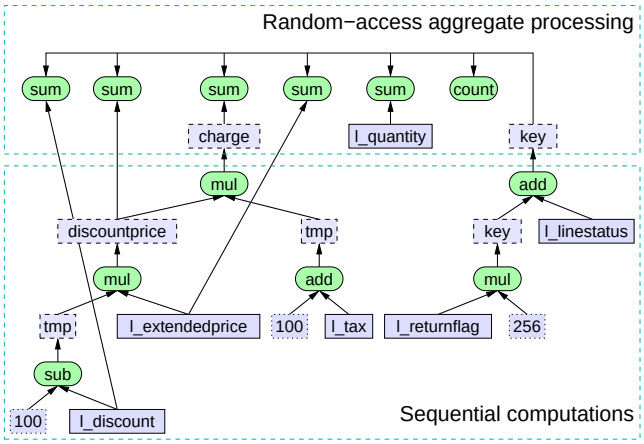



**Figure 4: Simplified plan of TPC-H Query 1 (omitted date selection, links to the aggregates table, and post-aggregation computations)**

traditional hash-table based processing, we use *direct aggregation* [5], in which the position in the aggregates table is computed directly from the key attributes. Since the original Query 1 only uses 4 `GROUP BY` key combinations, we also tested a slightly modified version of it that adds a 3rd `extrakey` column, and artificially fills all three key columns to simulate different numbers of `GROUP BY` combinations.

### 3.3.1 Pull Selections Up

In DSM, calculations on simple directly addressed arrays (i.e. without selection vector) are amendable for SIMD-ization, hence execute significantly faster. Therefore, if a Select does not eliminate many tuples and is followed by computation (e.g. a Project), it becomes beneficial to first do the calculations with SIMD, and the selection only afterwards. This counter-intuitive "pull selections up" strategy is in fact applicable to TPC-H Q1. Note that for this optimization, it is not strictly necessary to put the Select on top of the Project. In MonetDB/X100, the Select is still executed first, but for each tuple-block by looking at the selectivity (the length of the selection vector `m`) Project primitives may choose to ignore the selection vector `sel` and compute results for all `n` tuples, benefiting from SIMD:

```
ADD(long a1[n],a2[n]; int sel[m]): int output[n]
 if (m > n/2) // if many selected, compute on all
   for(pos=0; pos<n; pos++)
     output[pos] = ADD(a1[pos],a2[pos]) // SIMD
 else
   for(idx=0; idx<n; idx++)
     pos = sel[idx]
     output[pos] = ADD(a1[pos],a2[pos])
```

In fact, this performance boost makes it beneficial for a plan that starts with NSM tuples to switch to DSM.

### 3.3.2 SIMD Aggregation

Another SIMD optimization concerns grouped aggregation in NSM. If multiple identical aggregate functions must be computed (e.g. TPC-H Q1 has 5 grouped SUMs), we can SIMD-ize the aggregate update operation. This means that we have a primitive SUM2 function, that sums two adjacent NSM columns of 64-bit longs (its start pointer is denoted `col2` here) with two adjacent 64-bits aggregate totals (`tot2`):

```
SUM2(long tot2[m],col2[n]; int grp[n],w1,w2)
 for(pos=0;pos<n;pos++)
   simd_t* dst = (simd_t*) (tot2 + w2*group[pos])
   simd_t* src = (simd_t*) (col2 + w1*pos)
   *dst = SIMD_ADD2_LONG(*dst, *src)
```

As grouped aggregation takes more than half of the execution time in TPC-H Q1, applying SIMD here significantly affects performance. In fact, SIMD Aggregation makes it beneficial to switch back from DSM to NSM after the calculations to profit from SIMD.

### 3.3.3 On-the-Fly NSM/DSM Conversions

We run TPC-H on data that is in both NSM and DSM storage layouts, but consider switching layout before and after doing the calculations (i.e. Select and Project). Also, the format of the aggregate table can be DSM or NSM.

The results of the experiment, presented in Figure 5, confirm the trends from the micro-benchmarks. The DSM-formatted input (A,B,C) achieves significantly better performance. However, the DSM-formatted hash table suffers from random memory accesses (A,D,E). Using an NSM hash table removes this problem (B,G), and converting the DSM data on the fly into NSM allows to perform SIMD-based aggregation, further improving the performance (C,H). For NSM input we see that converting it into DSM allows faster sequential computation (E,G). For additional analysis of the performance, Table 2 presents the profiling of different scenarios for a case with 32K unique `GROUP BY` keys. It shows, that the extra data conversion before doing the projection and the aggregation phase can be in some cases more than balanced by the performance improvement gained in the following computation.

The performance benefits presented in this section are limited due to a fact that most of the computation is based on 8-byte integers. The currently available SSE3 SIMD instruction set provides only 128-bit SIMD operations, allowing just 2 operations to be executed at once. Since SIMD functionality is continuously improving, we expect these gains to become more significant in the future.

## 4. RELATED WORK

Block-oriented processing [15, 5] recently gained popularity as a technique to improve query processor efficiency. Traditionally, its main goal was to reduce the number of function calls [15]. Further research demonstrated that it also can result in a much higher CPU instruction cache hit-ratio [19]. Block-oriented processing is also an enabling technique for different performance optimizations that require multiple tuples to work on: exploiting SIMD instructions [18], memory-prefetching [8], and performing efficient data (de)compression [20].

The trade-offs between NSM and DSM as disk storage formats were analyzed in [13], where it is demonstrated that DSM performs better when only a fraction of the attributes is accessed. In contrast to what our paper proposes, the system described in [13] forces a conversion of DSM data on disk into NSM before entering the block-oriented iterator pipeline, allowing DSM layout only in the early scan-select stages. Avoiding early materialization of NSM tuples in column stores also was the topic of [2], but this work still requires forming NSM tuples at some moment in the

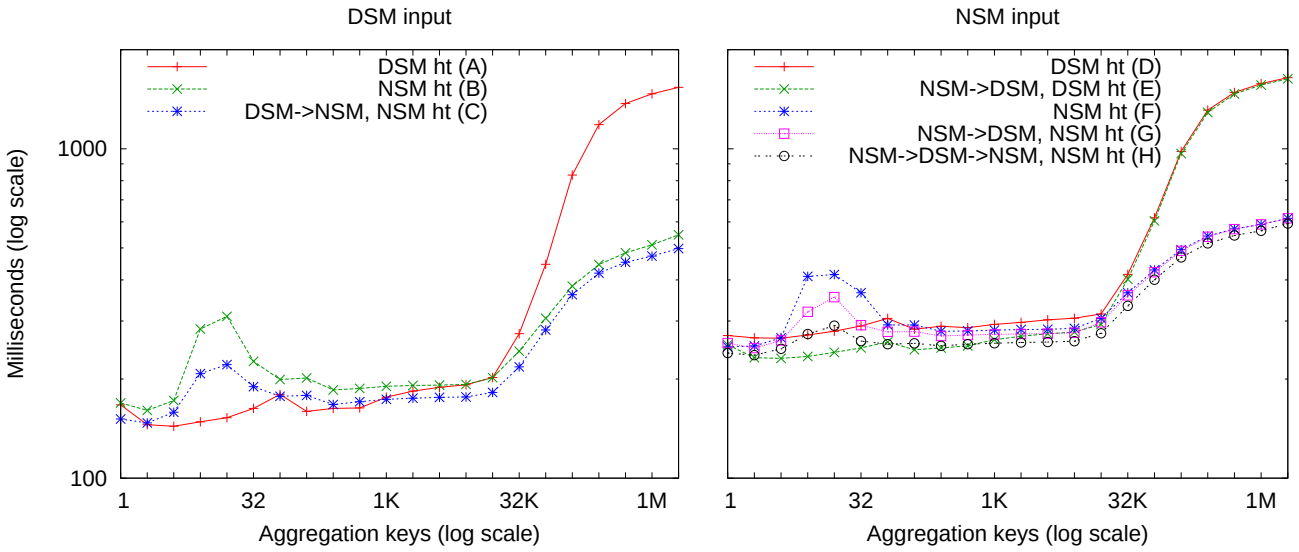



**Figure 5: TPC-H Q1, with a varying number of keys and different data organizations (ht – hash table)**

| | Time (millions of CPU cycles) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Source data | DSM | DSM | DSM | NSM | NSM | NSM | NSM | NSM |
| Projection phase | DSM | DSM | DSM | NSM | NSM | *DSM | *DSM | *DSM |
| Aggregation input | DSM | DSM | *NSM | NSM | NSM | DSM | DSM | *NSM |
| Aggregation table | DSM | NSM | NSM | DSM | NSM | DSM | NSM | NSM |
| Primitive | (A) | (B) | (C) | (D) | (E) | (F) | (G) | (H) |
| nsm2dsm_discount | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 325.07 | 340.47 | 337.72 |
| nsm2dsm_extendedprice | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 17.73 | 18.24 | 17.99 |
| nsm2dsm_tax | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 25.17 | 20.03 | 19.64 |
| nsm2dsm_quantity | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 16.84 | 17.14 | 16.93 |
| nsm2dsm_shipdate | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 19.70 | 20.02 | 19.45 |
| nsm2dsm_linestatus | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 22.30 | 19.21 | 19.10 |
| nsm2dsm_returnflag | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 22.76 | 19.31 | 19.12 |
| select | 28.40 | 27.98 | 27.96 | 330.37 | 338.77 | 38.93 | 39.77 | 39.00 |
| tmp = 100 - discount | 53.57 | 52.36 | 52.07 | 30.31 | 30.17 | 14.85 | 14.14 | 13.89 |
| discountprice = tmp * l_extendedprice | 47.52 | 47.17 | 47.33 | 40.67 | 40.99 | 18.52 | 17.56 | 17.80 |
| tmp = 100 + l_tax | 50.08 | 50.18 | 49.89 | 27.76 | 27.83 | 13.64 | 13.93 | 13.70 |
| charge = tmp * discountprice | 18.04 | 17.10 | 17.35 | 40.08 | 44.63 | 20.89 | 18.18 | 17.83 |
| key = 256 * l_returnflag | 20.66 | 20.09 | 20.12 | 28.85 | 28.69 | 19.43 | 19.77 | 19.39 |
| key = key + l_linestatus | 22.43 | 21.84 | 21.92 | 39.42 | 39.42 | 21.11 | 21.54 | 21.42 |
| key = 256 * key + extrakey | 33.66 | 32.95 | 33.07 | 64.39 | 64.46 | 32.38 | 32.78 | 32.24 |
| dsm2nsm_charge | 0.00 | 0.00 | 18.73 | 0.00 | 0.00 | 0.00 | 0.00 | 20.41 |
| dsm2nsm_discountprice | 0.00 | 0.00 | 20.07 | 0.00 | 0.00 | 0.00 | 0.00 | 20.09 |
| dsm2nsm_discount | 0.00 | 0.00 | 17.50 | 0.00 | 0.00 | 0.00 | 0.00 | 17.54 |
| dsm2nsm_extendedprice | 0.00 | 0.00 | 17.54 | 0.00 | 0.00 | 0.00 | 0.00 | 17.55 |
| COUNT() | 50.71 | 51.75 | 50.56 | 56.80 | 72.81 | 52.67 | 58.65 | 50.24 |
| SUM( charge ) | 55.05 | 80.24 | 0.00 | 59.49 | 0.00 | 56.98 | 104.75 | 0.00 |
| SUM( l_discount ) | 52.49 | 56.47 | 0.00 | 62.92 | 0.00 | 51.69 | 62.81 | 0.00 |
| SUM( discountprice ) | 52.24 | 48.02 | 0.00 | 58.46 | 0.00 | 52.01 | 47.47 | 0.00 |
| SUM( extendedprice ) | 55.43 | 48.65 | 0.00 | 69.11 | 0.00 | 56.97 | 48.71 | 0.00 |
| SIMD-SUM | 0.00 | 0.00 | 107.68 | 0.00 | 226.44 | 0.00 | 0.00 | 130.98 |
| SUM( l_quantity ) | 64.05 | 52.23 | 53.87 | 70.64 | 68.01 | 58.03 | 50.20 | 48.66 |
| TOTAL | 603.98 | 608.17 | 553.65 | 981.47 | 981.47 | 956.30 | 1006.63 | 931.14 |

**Table 2: Primitive function profile of a modified TPC-H Q1 (SF=1) with different data organizations. Star (*) denotes an explicit format conversion phase. Block size 128-tuples, 32K unique aggregation keys.**

query plan. Comparing DSM and NSM execution is also the focus of a recent paper [10], though the methodology is a high-level systems comparison without investigating the interaction with computer architecture.

Some performance analysis of DSM and NSM data structures has been presented in [11] where the authors propose "super-tuples" for both rows and columns. Related research is PAX [4], a storage format that combines low NSM costs for getting a single tuple from disk with good cache behavior of "thinner" DSM storage. In memory PAX is almost equivalent to DSM (the only difference is a possible impact on the possible block sizes), it has the same properties during the processing. The PAX idea has been generalized in [12] in the *data-morphing* technique, that allows part of the attributes in a given disk page to be stored in DSM and part in NSM, depending on the query load. This research, focused only on persistent data reorganization based on the query load. Our technique goes further, by proposing dynamic reorganization of transient, in-flight data.

On many architectures, SIMD instructions expect the input data to be stored in simple arrays, as in DSM. Since most database systems work on NSM, the potential of SIMD can often not be used. Notable exceptions include [18], as well as the family of MonetDB processing kernels [6, 5]. SIMD instructions are also becoming more relevant due to appearance of architectures such as Cell that provide SIMD only [14]. Interestingly, in this context it was already shown that grouped aggregates can only be SIMD-ized when using a NSM-like data organization (*array of structures*).

# 5. CONCLUSIONS AND FUTURE WORK

We have shown how different tuple layouts in the pipeline of a block-oriented query processor can strongly influence performance. For sequential access, DSM is the best representation, usually as long as the tuple blocks fit L2; DSM also wins for random access inside L1. If a sequential operator is amendable for SIMD-ization, this causes DSM to strongly outperform NSM; the difference sometimes even making it profitable to pull selections *upwards* to keep data densely organized for SIMD. NSM, on the other hand, is more efficient for random access operations (hash join, aggregation) into memory regions that do not fit L1. Unlike DSM, random access memory latency to NSM can be hidden using software prefetching. Finally, grouped Aggregation allows SIMD calculations only in case of NSM.

This means that it depends on the query which data layout is the most efficient in a given part of the plan. With the conversion between NSM and DSM being relatively cheap, we show that query plans such as TPC-H Q1 can be accelerated by using both formats with on-the-fly conversions. Therefore, we think that this work opens the door for future research into making tuple layout planning a query optimizer task. Additionally, more complex data representations should be investigated, including mixing NSM and DSM in one data block, as well as using indirect data storage.

# 6. REFERENCES


[1] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, 2006.

[2] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.

[3] S. Agarwal and H. Daeubler. *Reducing Database Size by Using Vardecimal Storage Format*. Microsoft, 2007.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, Rome, Italy, 2001.

[5] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[6] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, May 2002.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, pages 54–65, Edinburgh, 1999.

[8] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007.

[9] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, Austin, TX, USA, 1985.

[10] N. H. Daniel J. Abadi, Samuel R. Madden. Column-Stores vs. Row-Stores: How Different Are They Really? In *SIGMOD*, Vancouver, Canada, 2008.

[11] A. Halverson, J. L. Beckmann, J. F. Naughton, and D. J. DeWitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University of Wisconsin-Madison, 2006.

[12] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, pages 417–428, Berlin, Germany, 2003.

[13] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, 2006.

[14] S. Heman, N. Nes, M. Zukowski, and P. Boncz. Vectorized Data Processing on the Cell Broadband Engine. In *DAMON*, 2007.

[15] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, Heidelberg, Germany, 2001.

[16] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, Trondheim, Norway, 2005.

[17] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, September 2000.

[18] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, Madison, USA, 2002.

[19] J. Zhou and K. A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *VLDB*, Berlin, Germany, 2003.

[20] M. Zukowski, S. Heman, and P. Boncz. Architecture-Conscious Hashing. In *DAMON*, Chicago, IL, USA, 2006.

[21] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, Atlanta, GA, USA, 2006.