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-

e	t	name	price	CO2
e1	t1	A	50	0.5
e1	t3	A	52	0.5
e1	t5	A	52	0.3
e2	t2	B	100	0.7
e2	t4	B	105	0.7

(a)

ts	e
1	e1
2	e1
3	e1
4	e2
5	e2

ts	t
1	t1
2	t3
3	t5
4	t2
5	t4

ts	name
1	A
2	A
3	A
4	B
5	B

ts	price
1	50
2	52
3	52
4	100
5	105

ts	CO2
1	0.5
2	0.5
3	0.3
4	0.7
5	0.7

(b)

e	t	name
e1	t1	A
e2	t2	B

e	t	price
e1	t1	50
e1	t3	52
e2	t2	100
e2	t4	105

e	t	CO2
e1	t1	0.5
e1	t5	0.3
e2	t2	0.7

(c)

e	t	name	price	CO2
e1	t1	A	50	0.5
e1	t3	A	52	0.5
e1	t5	A	52	0.3
e2	t2	B	100	0.7
e2	t4	B	105	0.7

(d)

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright 2008 ACM 978-1-60558-184-2 ...$5.00.

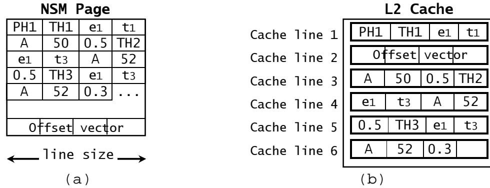

Figure 2: (a) NSM page layout. (b) Cache behavior for "find the CO_2 consumption history of A".

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

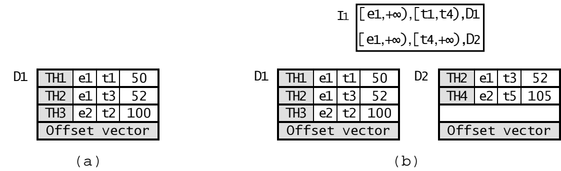

Figure 3: (a) The TSB-tree starts with one data page. (b) Time split of D_1 at t_4

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: L1, L2 {Two sorted list of resp. n1 and n2 tuples; each
  tuple T1 (resp. T2) of L1 (resp. L2) has an entity surrogate e, a
  timestamp t and an attribute a1 (resp. a2).}
Output: Lr {List of tuples resulting from the merge join of L1 and
  L2. Each tuple Tr of Lr has an entity surrogate, a timestamp and
  two attributes a1 and a2.}
  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

Step	L_p		L_c		Output		
1)	e1, t1, 50 (i)	e1, t3, 52	e1, t1, 0.5 (j)	e1, t5, 0.3	e1, t1, 50, 0.5		
2)	e1, t1, 50	e1, t3, 52 (i)	e1, t1, 0.5	e1, t5, 0.3 (j)	e1, t1, 50, 0.5	e1, t3, 52, 0.5	
3)	e1, t1, 50	e1, t3, 52	e1, t1, 0.5	e1, t5, 0.3 (j)	e1, t1, 50, 0.5	e1, t3, 52, 0.5	e1, t5, 52, 0.3
4)	e1, t1, 50	e1, t3, 52	e1, t1, 0.5	e1, t5, 0.3	e1, t1, 50, 0.5	e1, t3, 52, 0.5	e1, t5, 52, 0.3

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 θ, 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

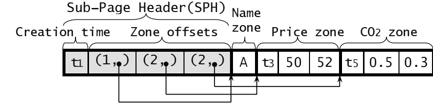

Figure 6: Sub-page layout

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

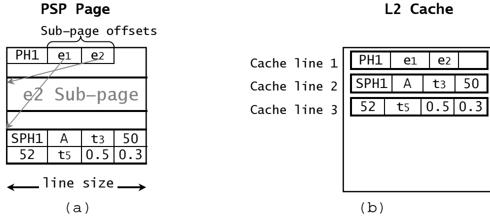

Figure 7: (a) PSP page layout. (b) Cache behavior for "find the CO_2 consumption history of A".

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 δ 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, δE entities, randomly selected, are updated. The data agility δ 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 2×2 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

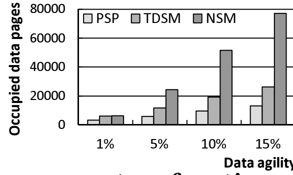

Figure 8: Storage cost as function of data agility

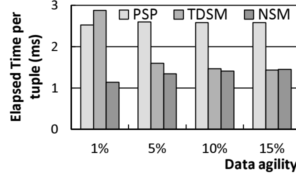

Figure 9: Insertions: average elapsed time per tuple as function of data agility

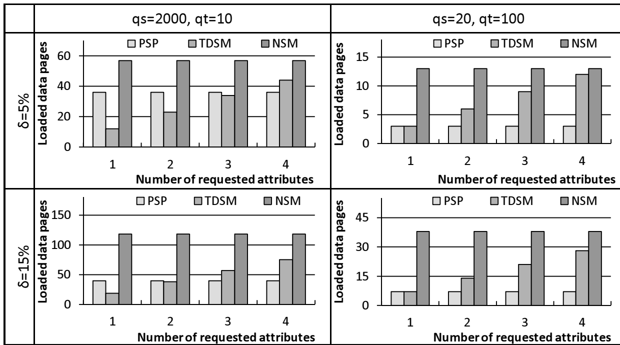

Figure 10: Queries: average loaded disk pages

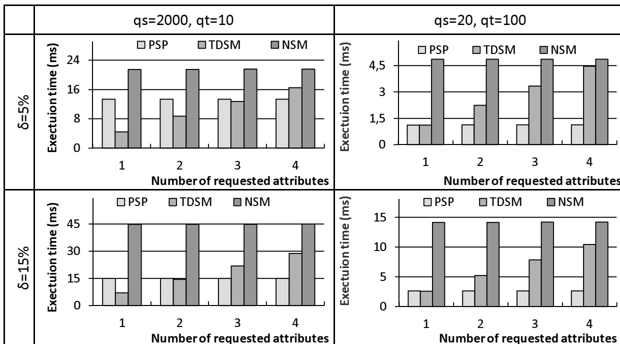

Figure 11: Queries: average execution time when data are fetched from disk

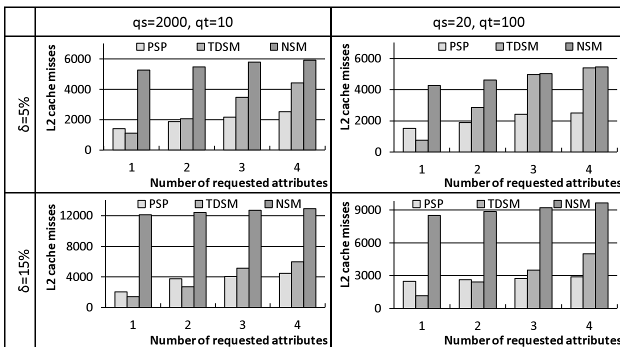

Figure 12: Queries: average L2 cache misses when data are fetched from main memory

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 ≈ 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 ≈ 2.74 times faster than NSM; PSP is on average ≈ 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

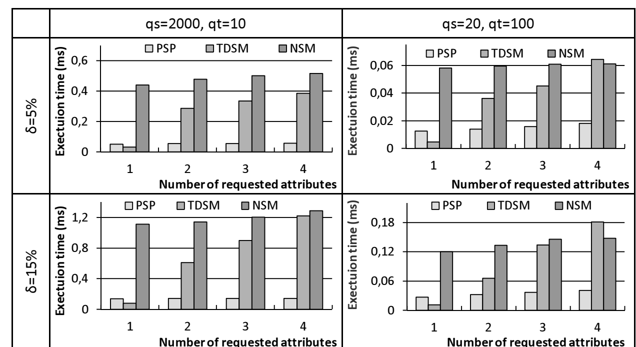

Figure 13: Queries: average execution time when data are fetched from main memory

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 ≈ 6.54 times faster than NSM and ≈ 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 ith minipage stores all the values of the ith 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.