Positional Delta Trees to reconcile updates with read-optimized data storage

Sándor Héman, Niels Nes, Marcin Zukowski, Peter Boncz
Firstname.Lastname@cwi.nl
Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

***Abstract*— We investigate techniques that marry the high read-only analytical query performance of compressed, replicated column storage ("read-optimized" databases) with the ability to handle a high-throughput update workload. Today's large RAM sizes and the growing gap between sequential vs. random IO disk throughput, bring this once elusive goal in reach, as it has become possible to buffer enough updates in memory to allow background migration of these updates to disk, where efficient sequential IO is amortized among many updates. Our key goal is that read-only queries always see the latest database state, yet are not (significantly) slowed down by the update processing. To this end, we propose the *Positional Delta Tree* (PDT), that is designed to minimize the overhead of on-the-fly merging of differential updates into (index) scans on stale disk-based data. We describe the PDT data structure and its basic operations (lookup, insert, delete, modify) and provide an in-detail study of their performance. Further, we propose a storage architecture called *Replicated Mirrors*, that replicates tables in multiple orders, storing each table copy mirrored in both column- and row-wise data formats, and uses PDTs to handle updates. Experiments in the MonetDB/X100 system show that this integrated architecture is able to achieve our main goals.**

I. Introduction

Read-optimized databases, that use columnar data storage (DSM [5]) in combination with compression and replication have recently re-gained commercial and research momentum [14], [2], especially for performance intensive read-mostly application areas such as data warehousing, as they can significantly reduce the cost of disk I/O with respect to row-clustered disk storage (NSM or PAX [1]), at the price of more expensive updates.

At the same time, application areas like data warehousing experience user pressure to shorten or even fully eliminate data refresh times, raising the golden question whether it is possible after all to marry the benefits of read-optimized databases with the update throughput of row-stores.

The C-Store and Vertica systems try to address this question by splitting their architecture in a read-store and a write-store, where changes in the write-store are periodically merged into the read-store [14]. In this paper, however, we focus on solutions where read-queries always see on the most recent state of the database.

Read-Optimized Databases. Columnar storage allows scan queries that need a subset of all columns to access data blocks that only contain useful data, reducing disk I/O requirements for a given query with respect to row storage. The disadvantage of columnar storage is that each update or delete to an C-column table leads to C disk block writes, as opposed to just one in case of NSM and PAX.

Modern column stores often keep tuples in some user-specified order, which helps to restrict scans also in the horizontal dimension, if the predicate is a range condition along this order (i.e. allows to scan only those blocks from a column that are relevant for the range predicate). The flip-side of the coin of maintaining a particular order is that bulk insert operations, typical in data warehousing workloads, are no longer localized at the end of each table, but may cause scattered disk I/O over the entire table.

Data compression, when combined with columnar storage or PAX, benefits from subsequent data items being of the same type and belonging to the same value distribution, allowing for faster and more effective data compression than in row-storage [17], [14], and hence further reduces disk I/O. The disadvantage in case of updates is that a full disk block needs to be read, de-compressed, updated and re-compressed (introducing significant CPU cost), before being written back to disk. Extra complications occur if the updated data no longer fits its original location.

Adding replication to this picture allows to maintain table replicas in R different orders, which can strongly increase the query percentage in a given workload that benefits from clustered range scans, reducing disk I/O for read queries. The obvious disadvantage of replication during updates is that the amount of update I/O also increases by a factor R.

Differential Updates. We argue that the key to providing good update performance in read-optimized databases is to avoid updating the read-optimized (columnar, compressed, replicated) data structures immediately. Rather, updates should be batched into a RAM-based *differential structure*. This structure grows over time until a background checkpointing process migrates (parts of) it to disk, combining the effects of many updates in each single bulk disk write, thus amortizing I/O maintenance cost. Durability in such an approach is achieved by directly logging updates in a disk-based Write Ahead Log (WAL), which extends sequentially.

The idea of using differential structures is quite old. Apart from the original proposal of differential files [12], a well-developed data structure exploiting this idea is the Log Structured Merge tree (LSMT) [9]. The LSMT is actually a stack of trees, that differ in size by a fixed ratio, where each tree is a

(insert,delete) delta on top of the underlying data. The topmost, smallest, tree is typically cached in RAM, whereas the layout of the other read-only, disk-resident, trees is optimized for sequential access (100% full disk blocks of a large size). While the LSMT offers enhanced throughput with respect to insert- and delete-intensive workloads, the increased lookup and range scan cost (in the worst case, all trees have to be checked, leading to at least one disk I/O per tree) has held back its acceptance as a general-purpose indexing structure.

PDT. Providing transactional consistency without this affecting the speed advantage of read-optimized databases, is the key goal of this work.

The previously proposed differential files and LSMT store differences as $(key, type, kind)$ information. Applying deltas then implies using a merge-join algorithm on key *values* (in case of composite key being the conjunction of equality tests on all corresponding key columns), which may make the normal Scan operation considerably more CPU intensive. We call this operator the *MergeScan*, as it provides the functionality of a (range) Scan operator, but has the additional task of merging in deltas. Note that such classical value-based differential delta merging has the disadvantage in DSM systems that it forces to scan along all key columns in *all queries*, which can significantly increase their disk IO needs, and impact performance.

In this paper, we contribute a new data structure called the Positional Delta Tree (PDT). The PDT is a tree that contains the differential updates with $type \in$\{insert,delete,modify\}. The PDT is designed to make merging in differential updates extremely fast by pre-computing the tuple *positions* where deltas have to be applied. Therefore, instead of performing a value-based MergeScan on tree key, the range scan can simply count down to the next position where a difference has to be applied, and apply it blindly when the scan cursor arrives there.

Thus, the key advantages of the PDT over value-based merging are (i) PDT-based MergeScan is less CPU intensive than value-based MergeScan, and (ii) queries need to perform less IO as the key columns are not strictly needed.

Unlike the LSMT, the PDT is primarily designed for use in RAM. The lowest level in the data hierarchy is not a PDT, but read-optimized disk storage (i.e. columnar, compressed and/or replicated data storage). Search operations in the PDT involve at most searching three PDTs (of which the top-most two are so small that they should fit in the L1 CPU cache). If PDT search in RAM is unsuccessful, it escalates to an access method supported by the disk storage layer (depending on what is used).

Replicated Mirrors. Besides the PDT data structure, a second contribution of this paper is to outline a physical database design architecture for high update throughput data warehouses, based on Fractured Mirrors. The original Fractured Mirrors proposal [10] suggested as future work the use of a differential technique to handle updates. We propose PDTs as exactly this contribution, and combine it a with indexing and replication.

Replicated Mirrors store each relational table replica twice, in a columnar-representation and a row-wise representation, which end up on different sets of disks, thus obviating the need for additional RAID protection. Additionally, each relational table may be replicated multiple times, each using a different order criterion. At least one replica contains all columns, other replicas may only contain a subset of all columns. Each replica has a *sparse index* on its order columns, that can be used to perform range scans on both the column- and row-wise mirrors. Updates to both mirrors are amortized using the PDT. Read-mostly OLAP queries can exploit the columnar layout using MergeScan to retrieve (ranges of) columns with the latest updates from the PDT merged in, while a simultaneous workload of small OLTP queries can exploit the row-wise mirror.

We evaluate our proposed Replicated Mirror set-up in the MonetDB/X100 vectorized database engine, developed as a prototype at CWI [2], and show that it can fulfill our goal of unaffected high read-only performance combined with update throughput that competes with a row-store.

Outline. This paper is organized as follows. In Section II, we describe in detail the PDT data structure, and outline its basic operations in Section III. In Section IV, we then give a detailed definition of the proposed Replicated Mirrors scheme that combines PAX-DSM mirroring with sparse indices and PDTs, and in Section V provide a performance model that motivates its efficiency. In Section VI we describe related work before concluding in Section VII.

II. Positional Differential Updates

A. Terminology

A *column* is a sequence of relational attribute values. A *table*, T, is a collection of related columns, all of equal length. A *tuple*, or *record* is a single row within T. Tuples should be aligned over columns, i.e., the attribute values that make up a tuple should be retrievable using a single positional index value, for all the columns that make up a table. This index we call the *row-id*, or RID, of a tuple.

Tuples in a table can be ordered along a subset of attributes, S, and can have a possibly distinct set of key attributes, K, that uniquely identify a single tuple. A subset of attributes that defines a sort order while also being a key of a table we call SK.

An update on a table is one of *insert*, *delete*, and *modify*. An $insert(T, t, i)$ adds a full tuple t to table T, at RID i, thereby incrementing the RIDs of existing tuples at RIDs $i \cdots N$ by one. A $delete(T, i)$ deletes the full tuple at RID i from table T, thereby decrementing the RIDs of existing tuples at RIDs $i \cdots N$ by one. A $modify(T, i, j, v)$ changes attribute j of an existing tuple at RID i to value v.

Assuming we maintain updates against a table in a differential structure Δ, we can define the following table hierarchy:

$$UpdateTable = (\Delta, StableTable, timestamp) \quad (1)$$

$$StableTable = DiskTable | UpdateTable \quad (2)$$

where a $DiskTable$ is an immutable, disk-resident, read-optimized (e.g. columnar, compressed) table, and $UpdateTable$ is an immutable $StableTable$, with differential updates against it stored in Δ. The $timestamp$ of an

$UpdateTable$ represents the time it was instantiated, starting out with an empty differential file, $\Delta \leftarrow \emptyset$. Once an $UpdateTable$ is used as a $StableTable$ to create a new $UpdateTable$, it becomes immutable, and future updates against the table should go into the differential structure of the newly created $UpdateTable$. This leads to a hierarchy with a $DiskTable$ at the bottom, and an arbitrary number of differential structures stacked on top of it, where only the topmost differential structure can be modified.

We define *stable-id*, or SID, to be the position of a tuple within the a $StableTable$. The *row-id*, or RID, is the position of a tuple within the table image produced by applying all differential updates.

B. Value-based Deltas

To be able to merge differential updates into a scan stream efficiently, the updates themselves can be kept in the sort order of the table they apply to. If that is the case, interleaving the differential updates into the scan stream boils down to a two-way merge with at most linear complexity. Maintaining differential updates ordered by their sort order attribute values has been assumed in all previous work [12], [8], [10]. Storing both the main table as well as the deltas in order, is usually done using tree data structures, such as the aforementioned LSMT [9]. We refer to the idea of identifying and maintaining deltas using (ordered) attribute values by *Value-based Deltas*, and the concept of using a tree to store these as a *Value-based Delta Trees* (VDT).

Note that, in case the sort order attributes of interest, S, are not a key for T, both the table and the differential file should be maintained ordered on SK, an extension of the sort order attributes that make it a key. Although such a maintenance of updates is easy, the price has to be paid during merging of the updates, as we need to locate the exact tuple each update applies to by comparing its values on SK to those of the tuples in the original table, which can be a costly process, in terms of CPU overhead, even when assuming that both streams are sorted. Furthermore, in case we are dealing with DSM, we would always need to scan along the SK attributes to be able to identify updated tuples, even if the query itself does not require access to these columns. In case the sort order is not a key by itself, something not uncommon in analytical scenarios, especially if tables are replicated in different sort orders, this boils down to scanning along at least two columns. For tables without a key, value-based differential updates require addition of an artificial key, like a tuple-id (TID). Clearly for DSM, value based differential updates do not only induce CPU overhead, but disk I/O overhead as well. As these overheads violate our starting requirement that read-only performance should not be compromised, we investigated different solutions.

C. Positional Deltas

An alternative way to maintain differential updates, is by means of *position* rather than by value. This has the advantage that updates can be merged efficiently, as it is trivial to compute the gap till the next update, which allows sequences in the original data that do not have any updates to be processed without any additional CPU overhead. Tuple position, however, is a dynamic concept, as soon as tuples can be inserted and deleted at arbitrary locations. To be able to maintain differential updates by means of a dynamic position, we introduce the *Positional Delta Tree* (PDT).

The PDT resembles a B^+-Tree, that stores differential information of the form $(SID, type, value)$, where SID is the location within the underlying $StableTable$ where the update applies to, $type$ is one of insert/delete/modify, and the $value$ associated with the update. If the PDT is associated with a table, $value$ is a a tuple, if it is associated with a single column, $value$ is atomic. Note that this also stores a value with tuple deletions. We will elaborate on these issues later.

The PDT can be searched by either SID, RID, or (SID, RID), to locate existing updates or add new ones. As RIDs are changing continuously all over the table, materializing and maintaining them is unacceptable. Instead of materializing RIDs, we define $RID = SID + \delta$, with δ being determined by the number of preceding inserts and deletes, and defined as:

Definition 1: During a sequential scan of a table, or, alternatively, a sequential walk of the leaves of a PDT, δ is defined as the the total number of newly inserted tuples, minus the number of deleted tuples, up till the scan position.

Each insert contributes an increment of δ by one, and each delete contributes a decrement of δ by one. Modifications do not have impact on the value of δ.

An example PDT can be found in Figure 1. Internal nodes maintain F child pointers, where F is the fan-out of the tree, and two aligned lists of length $F-1$, the first of which stores SIDs, and the second storing δ values:

```
PDT_internal_node {
  sid[F],
  delta[F],
  child[F],
  parent
}
```

The delta field `delta[`i`]` represents the relative number of inserts and deletes in the subtree rooted at child `delta[`i`]`. We maintain delta on a per-subtree basis to avoid high maintenance cost (i.e. when adding an insert or delete we only need to modify delta fields along the path from the leaf to the parent). When searching the tree on RID, or (SID, RID), we therefore need to compute the cumulative sum of deltas, until we find that `sid[`i`]` + $\sum_{\forall j \in lookup(i)}$ `delta[`j`]` is bigger than the RID we are searching for, in which case we take the branch to child i, as illustrated in Algorithm 1.

Leaf nodes of the PDT simply store update triples ordered on (SID, RID). in a structure like:

```
PDT_leaf {
  sid[F],
  type[F],
  value[F],
  parent,
  next
```

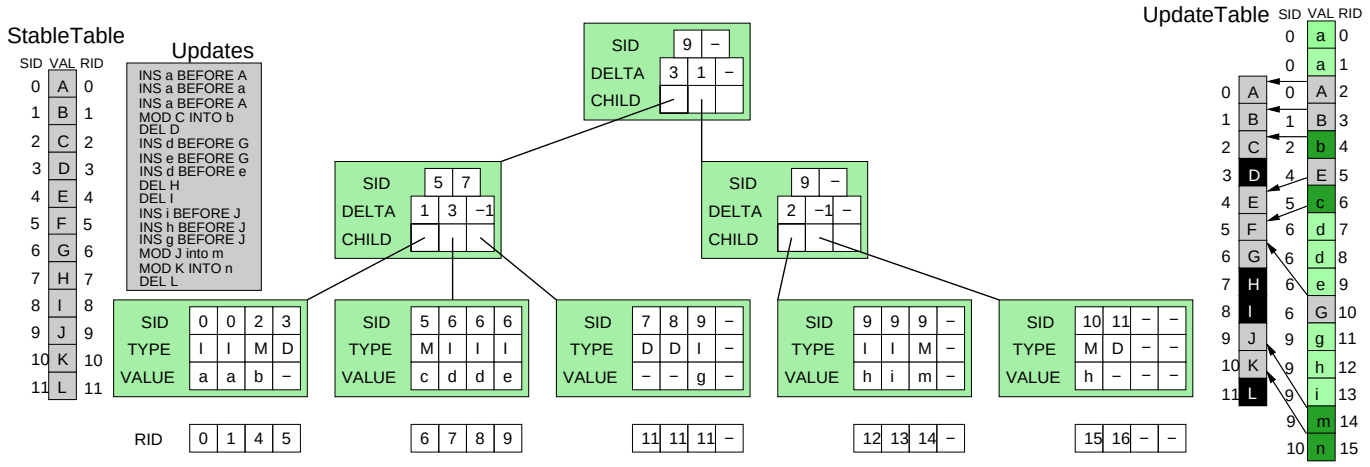

Fig. 1. Example of the Positional Delta Tree (PDT), and its use in creating a virtual UpdateTable on top of an unmodified StableTable

```
}
```

Per-Tuple PDTs. In this example, the PDT leaf nodes contain a simple `value` from a single column; while in a typical database implementation its stores delta information for entire tuples, such that the `value` field would then be a pointer or offset into some tuple-space or -heap, as illustrated in Figure 2. As for storage overhead in that case, the important aspects are the size of the `value`, `sid`, and `type`; as even with moderately low F these fields dominate. A typical implementation, the `sid` could be a 48-bits integer field. The `type` takes three basic values (*insert,delete,modify*), for which just 2 bits are required. In a tuple-based PDT, however, for modify we also need the information which column was updated. This could be handled by a 16-bit `type`, that uses 2 values for insert/delete and leaves all other values to indicate a modification of a certain column (identifying one out of maximum 65534 columns). This sets the PDT overhead per update to 12 bytes, assuming a 32-bits `value` (offset). Note that when a table is replicated in multiple orders, a separate PDT must be kept for each replica. However, the updated values in the various replicas are identical, so it makes sense to share the tuple-space between replicas. This way, PDT RAM consumption scales sublinearly with replication degree.

Figure 2 shows in detail how various table replicas, for each of which a separate PDT is maintained, can share a *single* value-space. The value-space is an memory-resident data structure that stores, separately, inserted rows (a value for all columns) and modifications (a value for one column only). This split insert/modification storage minimizes PDT memory utilization, since the PDT leaves that point to an insert can use a single offset into the value-space that leads to all inserted values, while modifications on a single column store only a single value. The use-counts maintained for inserts and for each updated column are for garbage collection (as will be discussed in the Checkpointing section).

D. Basic Properties and Algorithms.

To be able to maintain the tree, each tuple, being it a stable one, a deleted one, or even a newly inserted one, should have a

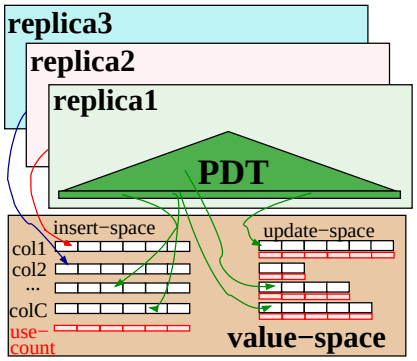

Fig. 2. Saving PDT space across replicas

SID. Each tuple furthermore automatically has a RID, defined as $RID = SID + \delta$. This means that even deleted tuples stay around as ghost records, sharing the RID of their direct successor, as a deletion decrements the cumulative δ by one. Newly inserted tuples are assigned the same SID as the tuple *before* (according to table sort order) which they are inserted. Clearly, neither SID nor RID are guaranteed to be unique within a PDT. Their concatenation however, is a unique per-tuple key.

Theorem 1: The concatenation of SID and RID, (SID, RID) is a unique key.

Proof: We prove by contradiction. Assume we have two tuples with equal SID. The first of these is always a newly inserted tuple, which increments δ by one. Given that $RID = SID + \delta$, the second tuple can never have an equal RID. Next, assume we have two tuples with equal RID. The first of these is always a deleted tuple, which decrements δ by one. The second tuple can never have an equal SID, as $SID = RID - \delta$. ■

Corollary 2: If updates within a PDT are ordered on (SID, RID), they are also ordered on SID and on RID.

Corollary 3: Within a PDT, a chain of N updates with equal SID is always a sequence of $N - 1$ inserts, followed by either another insert, or a modification or deletion of an underlying stable tuple.

Algorithm 1 FindLeafByRid(rid)
Given a RID, finds the rightmost leaf containing updates on given rid. Versions that find the leftmost leaf, or search by SID or (SID, RID) are omitted

```
1: node = root_node
2: δ = 0
3: while is_leaf(node) ≠ true do
4:    for i = 0 to node_count(node) do
5:       δ = δ + node.delta[i]
6:       if rid < node.sids[i] + δ then
7:          δ = δ − node.delta[i]
8:          break from inner loop
9:       end if
10:   end for
11:   node = node.child[i]
12: end while
13: return (node, δ)
```

Algorithm 2 AddModify(PDT, rid, new_value)
Finds the rightmost leaf containing updates on a given rid. Within that leaf, we either add a new modification triplet at index pos, or modify in-place.

```
1: (leaf, δ) = FindLeafByRid(rid)
2: (pos, δ) = SearchLeafForRid(leaf, rid, δ)
3: while leaf.sid[pos] + δ ≡ rid and leaf.type[pos] ≡ −1
   do {Skip deletes with conflicting RID}
4:    pos = pos + 1
5:    δ = δ − 1
6: end while
7: if leaf.sid[pos] + δ ≡ rid then {In-place update}
8:    leaf.value[pos] = new_value
9: else {add new update triplet to leaf}
10:   ShiftLeafEntries(leaf, pos, 1)
11:   leaf.type[pos] = 0
12:   leaf.sid[pos] = RID − δ
13:   leaf.value[pos] = new_value
14: end if
```

Corollary 4: Within a PDT, a chain of N updates with equal RID is always a sequence of $N-1$ deletions, followed by either another deletion, or a modification of the subsequent underlying stable tuple, or a newly inserted tuple.

Adding a new modification or deletion update to a PDT only requires a RID, as deleted ghost records are not present in the final table image, which is also the image seen by the modify and delete operators. We only need to make sure that the new update goes to the end of an update chain that conflicts on RID, within the PDT. If the final update of such a chain is an existing insert or modify, we need to either modify or delete that update in-place, within the PDT. The procedures for adding a new modification or deletion update to a PDT are outlined in Algorithms 2 and 3, respectively. Tree specific details, like splitting full leaves and jumping from one leaf to its successor, are left out for brevity.

For insert updates, inserting by RID only is not sufficient, as we can not determine where in the PDT to store the insert, with respect to deleted ghost records. Inserting by SID only does not work as well, as we can not determine where to store it with respect to conflicting inserts on the same SID. As we want such conflicting insert chains to be stored according to the underlying tables value based sort order, we need to translate from the values on attributes SK to a unique (SID, RID) combination, identifying the existing tuple before which the newly inserted tuple should go. How we achieve this mapping is discussed below. The fact that we maintain values for deleted tuples is related. For now, assuming we have (SID, RID) to insert at, we can add an insert update using Algorithm 4.

To keep PDT updates ordered on both (SID, RID) and on their values on the underlying tables sort order attributes, SK, both concepts should be correlated. For modifications and

Algorithm 3 AddDelete(PDT, rid, old_value)
Finds the rightmost leaf containing updates on a given rid. Within that leaf, we either add a new deletion triplet at pos, or delete in-place.

```
1: (leaf, δ) = FindLeafByRid(rid)
2: (pos, δ) = SearchLeafForRid(leaf, rid, δ)
3: while leaf.sid[pos] + δ ≡ rid and leaf.type[pos] ≡ −1
   do {Skip deletes with conflicting RID}
4:    pos = pos + 1
5:    δ = δ − 1
6: end while
7: if leaf.sid[pos] + δ ≡ rid then {In-place update}
8:    if leaf.type[pos] ≡ 1 then {Delete existing insert}
9:       ShiftLeafEntries(leaf, pos, −1)
10:   else {Change existing modify to delete}
11:      leaf.type[pos] = −1
12:   end if
13: else {add new update triplet to leaf}
14:   ShiftLeafEntries(leaf, pos, 1)
15:   leaf.type[pos] = −1
16:   leaf.sid[pos] = RID − δ
17:   leaf.value[pos] = old_value
18: end if{Decrement deltas along the path from leaf to root
   by 1}
19: IncrementNodeDeltas(leaf, −1)
```

Algorithm 4 AddInsert(PDT, sid, rid, new_value)
Finds the leaf where updates on (sid, rid) should go. Within that leaf, we add a new insert triplet at index pos.

```
1: (leaf, δ) = FindLeafBySidRid(sid, rid)
2: while leaf.sid[pos] < sid or leaf.sid[pos] + δ < rid do
   {Skip updates with lower (SID, RID)}
3:    δ = δ + leaf.type[pos]
4:    pos = pos + 1
5: end while{Insert update triplet in leaf}
6: ShiftLeafEntries(leaf, pos, 1)
7: leaf.type[pos] = 1
8: leaf.sid[pos] = RID − δ
9: leaf.value[pos] = new_value {Increment deltas along
   the path from leaf to root by 1}
10: IncrementNodeDeltas(leaf, 1)
```

Algorithm 5 SKtoSidRid(tuple[SK])
This routine takes a partial tuple of sort key attribute values as input, and returns the position where these values belong in terms of (SID, RID) ordering.

1: $sid = FindSID(tuple[SK])$ {By means of traditional search}
2: $(rid_{lo}, rid_{hi}) = GetRidRange$

deletions there are no dangers of violating this correlation: deletions triplets within a PDT always refer to an underlying stable tuple, which is assumed to adhere to ordering on SK, and modifications can only occur on attributes that are not in SK (i.e. a modification of an SK attribute would result in a deletion plus an insertion of the modified tuple, which changes its position). Only inserts need special attention to retain an exact ordering on SK. This can be achieved by first locating the SID of the tuple in the stable table before which the new tuple should go. With this SID we search the PDT for any updates on that SID. As all updates in the PDT contain a value, even deletions, we can search for RID of the tuple before which to insert.

Correlating (SID, RID) ordering and SK value ordering is important for random tuple lookups. For example, when performing an index scan that involves range predicates, we need to be able to translate the predicate values to (SID, RID), so that we can produce the correct range of tuples, including tuples introduced by PDT inserts and leaving out all that got deleted. The procedure to translate SK values to (SID, RID), is also used for determining the exact position of insert updates, and is listed in Algorithm 5.

1) Complexity: Although it depends heavily on the workload, we could come up with at least average case complexity under a uniform load (it's similar to B-tree, except that we cannot binary search within internal nodes, due to cumulative delta summation. We do have a small fan-out in general though). Space complexity should not be hard either.

III. Database Operators

A. MergeScan

In systems that use block-oriented processing, where a Scan does not produce a single next tuple, but a block or *vector* of V next tuples, the check to see whether there is a change within the next V tuples can be done once per block; reducing the checking overhead. Since the percentage of updates tuples tends to be very low (<1%) this will often be the case, such that the more expensive merging code path can be avoided in the common case.

A brief outline of the optimized merging algorithm is in Algorithm 6.

The idea here is that `gap` represents the distance in position till the next insert or delete, *not* modify. If the gap is large, we `merge_patch` these gap tuples. `merge_patch` first checks whether there are modifies for this column within distance gap. If not, we return a hard data pointer. If there are, we first memcpy input data, and then patch modifications. If `in_place` is true, input data is already in private buffer

Algorithm 6 SKtoSidRid(tuple[SK])
This routine takes a partial tuple of sort key attribute values as input, and returns the position where these values belong in terms of (SID, RID) ordering.

```
1: for each column c
2:   res[c] = fetch(next V tuples)
3:
4: if (gap > V) // fast code
5:   for each column c
6:     pdt[c].merge_patch(res[c],V,in_place)
7:     gap -= V
8: else
9:   for each column c // slow code
10:    pdt[c].mini_merge(res[c],V)
11:  gap = check_ins_and_del(key_col_pdt)
```

(i.e. decompressed) and patching can be done in place. The `mini_merge` has to deal with insert/delete, and thus copies always. It tries to exit as soon as it finds that the gap till the next insert/delete is bigger than 63 (to avoid huge fragmentation of vectors).

B. Lookup

A lookup query that selects tuples based on some predicate is handled in the general case by a MergeScan followed by a ScanSelect. However, equality lookup to the SK attributes can be much faster (sub-linear) by taking advantage of indexing structures. In the Replicated Mirrors implementation in MonetDB/X100, for instance, we use a sparse index on the SK attributes that identifies a single 64KB PAX block where such tuples are found (within the block, binary search lookup can be used). Recall that both the read-optimized storage, as well as the indices on it are stale (i.e. the differences after the last checkpoint are in the PDT). A lookup in the index thus yields (if successful) the stable ID (SID).

The second step is to use the SID to access the PDT. Note that multiple leaf nodes in the PDT may have the same SID, so looking up a SID also involves a range traversal among sibling leaves. The thus found changes should be analyzed to see whether or not a tuple exists: initial success in lookup on the stable data may be overruled by a corresponding delete in the PDT, while failure to find a tuple in the stable image may be overturned by the presence of an insertion.

In case there are multiple, stacked PDTs, the PDT lookup process should be repeated for all PDTs starting at the lowest one.

C. Insert

Insert queries involve a table name and a list of attribute values that represents the tuple to be inserted into that table. They are easy in case there is no sort order on the table being updated, or, if the sort order is some auto-incremented tuple ID. In this case, the newly inserted tuple can be simply appended at the end of the table.

In case there is a table SK order, the first step is to determine the insert position (SID) in the StableTable; as

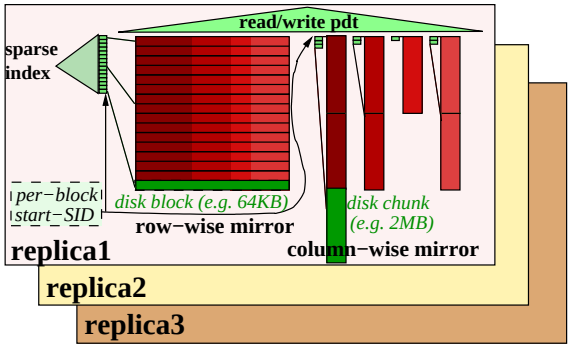

Fig. 3. Replicated Mirrors: replicated ordered table storage using a Fragmented Mirror plus PDT and Sparse Index

described above. The second step is to insert the new tuple in the PDT (the top-most PDT, if there is a stack of PDTs). Note that a PDT insert starts as a PDT lookup, until the correct leaf node is found/created for the insert. Along the way, the delta values in the traversed internal nodes are incremented by one.

D. Delete

Delete queries involve some predicate that identifies the tuples to be deleted. For each tuple, if its a (SID,RID) is present in the PDT as an insert or a modify, it is removed from the PDT. If the (SID,RID) is not present as an insert in the PDT, it is inserted as a delete in the PDT.

E. Modify

Modification queries also rely on a predicate that describes tuples to be modified. However, modifications involve only a subset of the attributes participating in a table. If a modified attribute is present as an insert or modification in the PDT, we can simply modify the value in the PDT (note that in case of one PDT per tuple, this modifies the PDTs for all replicas at once). Otherwise, a new modification entry is added to the PDT.

Modifications are more complicated in case a SK attribute is being changed, as this means the tuple will change its position. Such modifications therefore have to be seen as a deleted followed by an insert.

IV. Replicated Mirrors

We now discuss the proposed Replicated Mirroring scheme, depicted in Figure 3.

Mirroring. The gist of the original Fractured Mirror [10] paper is to store data both column- and row-wise, such that the query optimizer can choose which representation to use for which query (and even mix representations in the same query). The observation that database installations often already use physical mirroring in the form of RAID, is exploited to argue that recovery could exploit the fact that data is mirrored logically, thus making additional physical mirroring unnecessary, such that physical storage cost is not increased.

Fractured Mirrors divide (”fracture”) the columnar and row-wise mirrors across all disks, to spread out the load. The expected load to the column- and row-wise is different: column-wise data is accessed only sequentially in large chunks that consist of multiple disk blocks (we use a 2MB chunk size), whereas the row-wise data is accessed using random read IO (at a disk block granularity of 64KB) for handling OLTP queries, yet also sustains a low-intensity sequential checkpointing load at the chunk granularity.

In our MonetDB/X100 implementation, the row-wise mirror is stored using PAX [1], not NSM. Like NSM, PAX stores all data of a single row in the same disk block. However, inside the disk block, column values are laid out consecutively. This storage layout better fits the column-oriented MonetDB/X100 storage access methods and allows the Scan operator to retrieve vectors of data (small columnar slices that contain data of around 100-100 tuples) used in its *vectorized* query processing model, without any CPU investment to change the representation. Also, PAX allows to apply column-wise compression techniques. The Replicated Mirrors proposal works independent of the particular row-wise storage format used (e.g. NSM, PAX).

Indexing. In the Replicated Mirrors approach, each table is assumed to be stored in a certain order. The tuple order in each column- and row-wise mirror is identical among them and indexed by a *shared sparse index*, that can be used to access both mirrors.

The sparse memory-resident B+-tree index contains one key per row-wise block. The leaves store a value, a block number (pointing into the row-wise mirror), as well as a tuple number, that is a SID (StableID). As it stores around 20 bytes of data per 64KB NSM/PAX block, the size overhead is around 1/2700; making it possible to cache all sparse indices in RAM. Random-lookup single-record OLTP queries access the sparse index to get a block number. After fetching the NSM (or PAX) block, the individual tuple (and thus SID) is found using binary search on the ordered columns.

The SID can also be used to identify chunks in the DSM partitions. Given their large size (and thus low number) as well as their static nature (chunks are only updated during a checkpoint), we do not use a B-tree for this this sparse DSM index. Rather, a simple sorted array kept for each DSM column holds its start SID; and binary search is is used to find chunks by SID. Apart from the side, the simple array also holds the minimum and maximum value of each chunk. These minimum and maximum values can be used to limit the chunks read by unclustered scans, in case a range-selection query is being processed. This technique is especially effective in case of correlated columns (such as dates), were a table stored on one column, is also (almost) ordered on another column. In such cases, range selections on that other column can prune avoid processing many chunks.

Replicating. Our proposal of Replicated Mirrors re-uses the idea of a column- and row-wise mirror of table, and combines it with table replication in multiple orders. Only one replica (the “’master”) is required to store all columns, other replicas may only contain a subset of the table columns. The concept

of multiple replicas roughly coincides with that of clustered indices, however it recognizes that read-optimized databases better store such indices in their native memory-tight update-unfriendly compressed and columnar forms.

Each replica is sorted on a different set of order keys, and each replica comes with its own row- and column-wise mirrors, its sparse index. Additionally, each replica maintains a PDT data structure for each column (once, as PDTs are shared between the row- and column-wise mirrors).

Processing an OLTP single-row update proceeds as follows: (i) use the order keys to lookup up the relevant PAX block (ii) fetch this PAX block using one 64KB read I/O (iii) look up the tuple inside the block with binary search, so we get its SID (iii) update the PDTs of the affected columns in this replica. If there are other table replicas that have affected columns, these also need to be processed by repeating these steps. Note that there is only one single-block read IO on the critical path. If there are R replicas; $R-1$ subsequent IOs are needed to establish the SIDs of the affected tuple in these replicas (these IOs can be processed in parallel, though).

Checkpointing. In the very simplest solution, a checkpoint is a Scan query that materializes all tuples in a new table copy. Note that Replicated Mirrors keeps multiple replicas of the same table (in different orders) and each replica itself consist of a NSM (/PAX) and DSM table mirror. Each replica keeps a separate PDT, but all PDTs of the same table reference data in a single value-space (as illustrated in Figure 2). Therefore, checkpointing a table, implies checkpointing all its replicas, each of which consist of two mirrors. After checkpointing both mirrors, the PDT can be emptied fully (all data in it de-allocated).

However, in the value-space, data can not yet be deallocated, since other replicas may not have finished checkpointing, Therefore, reference counts should be added on the values in the value-space: only when the last replica that references an updated value is checkpointed, it can be cleared out. Note that it is reasonable to assume a limited number of replica's, such that these counters can be fitted into small (e.g. 8-bit) integers; therefore, these reference counts do not introduce much overhead.

Within a single replica, it may also be desirable to perform a *partial checkpoint*, that e.g. only checkpoints a single column, or even only a subset of the disk blocks (or chunks) of the column. The idea behind partial checkpoints is to optimize the balance of PDT RAM reduction with checkpointing I/O. However, if an insert (or delete) is checkpointed already to one column, but not to another column, the `delta` field in the internal nodes of the PDT that tells how much the RID and SID differ at that point, would need to be different for both columns. Therefore, partial checkpointing is only possible if a separate PDT is kept for each column (as opposed to one per tuple), which implies additional RAM consumption in proportion to the number of columns, due to PDT node storage overhead. In this study, we have not investigated partial checkpointing in depth, and kept it out of scope for our performance evaluation.

Snapshot Isolation. In the MonetDB/X100 system, we chose

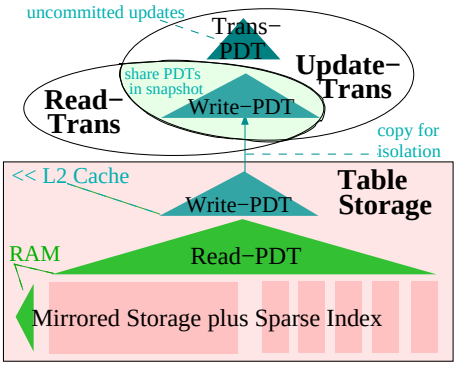

Fig. 4. Snapshot Isolation with Layered PDTs: only the small Write-PDT needs to be copied to achieve isolation

to implement transactions using snapshot isolation and optimistic concurrency control. This choice aligns well with the overall goal of adding transactional facility to the system without slowing down read-only query performance, as in snapshot isolation reads do not have to be locked.

The Log Structured Merge (LSMT) Tree [9] principle of using a stack of trees can also used in the PDT, to provide cheap snapshot isolation. A very small (L1 cache sized) *Write-PDT* is the one that gets updated by committing update queries. This small data structure can be copied in a quick memory copy operation, and is tied to a specific database snapshot. A second-tier larger *Read-PDT* is shared by all queries and contains all remaining changes. Update transactions use an additional top-tier *Trans-PDT*, that starts empty, and in which they insert their own (uncommitted) changes. If the transaction commits, the changes in the Trans-PDT are merged into a copy of the Write-PDT, that then becomes the new master Write-PDT seen by incoming queries. This mechanism provides snapshot isolation while practically avoiding any latching/checking overhead during query processing.

As the Write-PDT keeps growing as more transactions commit, its size will start to grow significant (outgrowing the CPU cache), and isolation cost (due to memory copying) will become noticeable. Therefore, the master Read-PDT is replaced from time to time by a new copy which incorporates all changes in the Write-PDT (which is then emptied). The current simple solution for doing so is to scan the leaves of the Write-PDT and apply all changes to a copy of the Read-PDT, which becomes the new master afterwards.

Recent analysis of Snapshot Isolation anomalies with respect to Serializability [6] has provided a simple and cost-effective method of concurrency control in snapshot isolated databases systems to guarantee serializability [3]. This idea can also be applied in our Replicated Mirrors approach. The idea is to keep for each transaction T two initially false booleans $T.in$ and $T.out$, and abort T as soon as both booleans become true. The variables are maintained as follows: if (1) T reads data modified by an overlapping transaction S, we set $T.out = S.in = true$, and if (2) T modifies data read by an overlapping S we set $T.in = S.out = true$. To implement (1), each time a new disk chunk is read, we should check the PDTs of all overlapping transactions. As chunks consist of

millions of tuples, such checks will not affect the performance large scan queries that ready multiple chunks, as they are well amortized. To check (2), we need to keep an administration table that for each chunk read inserts a SID ranges. Update queries need to check this table to detect conflicts.

V. Comparative Performance Model

We now provide a cost model to better understand the quantitative properties of Replicated Mirrors. We proceed in two steps: first we focus on OLTP and consider the properties of an OLTP system that processes a high throughput stream of single-tuple updates both on a traditional (single clustered B+-tree) row-store, and compare it with ordered row-wise storage with a sparse index, and updates flowing through a PDT (with checkpointing).

As a second step, we then consider the full Replicated Mirrors approach, where there also is a column-wise mirror and potentially multiple replicas, that serve a concurrent stream of OLAP queries.

The below table shows the parameters used in our model:

N	number of tuples in a table
C	number of columns
H	update throughput (inserts/updates/deletes per sec)
W	average column width (bytes)
D	number of disks (an even number)
R	number of table replicas (in Replicated Mirrors)
Z	average compression factor (in in Replicated Mirrors)
T	PDT per tuple size (uncompressed tuple width), $T = RC(W + 9)$
S	total disk storage size (bytes)
P	total RAM pool available (bytes)
B	large sequential read/write single disk bandwidth (bytes/sec)
I	maximum disk IOs/sec that a single disk delivers

We assume that the database system can use a RAM buffer that is 1/100 in size with respect to disk storage: $M = S/100$. This is motivated by the observed trends in cost per byte of respectively RAM and magnetic disk, where we basically allocate equal amounts of money to disk and RAM. In the year of this writing (2008), on the high end, a fast Seagate Cheetah NS SCSI drive costs $400 ($S$=400G, B=80M, I=300) while a 4GB stick of registered ECC DDR2 server memory also goes for $400. On the low end, for $200 we get a SATA drive that delivers (S=400G, B=70M, I=100) and as well as 4GB of common DDR2 RAM.

OLTP Only. We examine the worst-case where the OLTP load consists of updates only. In a row-store, an OLTP query that inserts, modifies or deletes a single record is assumed to perform just two IOs, for reading and writing the disk block where the row is stored (i.e. the table is stored in a B+-tree with all but the leaf level cached). However, note that writing a disk block typically involves two writes (if some RAID level is used). Concurrent updates are assumed to be independent and not to cause locking conflicts, hence throughput scales with the amount of disks. Thus, throughput is limited to $H = \frac{DI}{3}$ queries/sec. We assume a scattered non-clustered update load, such that even using all RAM for caching disk blocks in the buffer pool will only eliminate 1/100 of the IOs, not improving performance noticeably.

We now compare this with updating a single read-optimized row-wise mirror with its sparse index and PDTs. Both the approaches are assumed to use a Write Ahead Log (WAL) for durability and consistency, which is not modeled here explicitly. Disk IO for a WAL can be turned into bulk sequential IO by batching WAL records and delaying the reporting of transaction commit until a large WAL block has been flushed. WAL IO is thus not seen as a bottleneck.

By ignoring the WAL and focusing on record IO, the PDT approach on first sight always outperforms the traditional row-store as it performs just a single read IO for each OLTP query, to fetch the tuple and compute its SID. The updates are not written back to disk, rather added to the PDT data structure. However, the PDT data structure thus keeps growing, and therefore background checkpointing is necessary to keep its size under control. We assume as upper limit that we sacrifice at most half of the RAM to PDT storage ($P_{pdt} = \frac{P}{2} = \frac{S_{row}}{200}$). The question is thus when checkpointing gets in balance with the time in which the update workload fills up PDT memory P_{pdt}:

$$\begin{aligned} CheckPointTime &= MemFillTime \Leftrightarrow \\ \frac{S_{row}}{B_{checkpoint}} &= \frac{P_{pdt}}{H*T} \Leftrightarrow \\ \frac{S_{row}}{B_{checkpoint}} &= \frac{3S_{row}}{200*DI*T} \Leftrightarrow \\ B_{checkpoint} &= 66DTI \end{aligned}$$

where $B_{checkpoint}$ is the IO bandwidth (bytes/sec) taken up by checkpointing traffic. The percentage of bandwidth dedicated is thus:

$$CheckpointOverhead = \frac{B_{checkpoint}}{DB} = \frac{66TI}{B}$$

If we look at the TPC-H schema [15], the uncompressed tuple width ($C * W$) of the main LineItem table is 66 bytes. Substituting the parameters of the high- and low-end disks then put the checkpointing overhead at 1.5% resp. 0.6%, which is very low.

Moreover, the above calculation assumed that the PDT based system runs at the same throughput as the row-store ($H = \frac{DI}{3}$); however, since the PDT based OLTP setup needs to perform only a single read I/O per query, its maximum throughput without checkpointing is DI. At that speed, checkpointing overhead increases threefold to just 4.5% resp. 2.4%; therefore checkpointing and PDTs can outperform the B+-tree powered row-store by a factor close to three.

It is clear though that checkpointing efficiency decreases with larger per-tuple PDT memory consumption T, which depends on uncompressed tuple width CW. If we set 33% as the maximum checkpointing overhead, we can derive a maximum $T_{max} = \frac{B}{200I}$, which translates to records of 3KB resp. 7KB for high- and low-end disks respectively. These bounds are quite generous, and an undisputed trend in magnetic disk hardware is that bandwidth (B) is improving faster than IO operations per second (I), therefore, we may expect T_{max} to rise in future years, favoring our approach.

We therefore conclude that today's large RAM sizes, and the trend that disk bandwidth improvements outpace IO latency improvements, have created an opportunity for differential update processing to be used in OLTP DBMS engines as a viable alternative for in-place block writes.

Including the DSM mirror. If a DSM mirror is present, the

above calculations change a bit, as twice more data needs to be checkpointed. Going back to a desire of equaling row-store update throughput (i.e. $H = \frac{DI}{3}$ updates/sec) using Replicated Mirrors, we can thus roughly double the checkpointing overhead to 3.0% resp 1.2%. In this case, the disks are still only 33% busy, leaving 66% (minus 3.0% resp. 1.2%) for processing DSM queries.

In addition, if we consider a *replicated* table (e.g. $R = 2$), three things change: (i) if the update rate H remains the same at $H = \frac{DI}{3}$, then the disks will be 66% busy, since a random PAX lookup must be performed for each replica (i.e. R per update query). (ii) all replicas on disk need to be checkpointed, hence checkpointing cost increases linearly with R. (iii) additionally, the memory consumption per updated value per PDT leaf node is 12 bytes, assuming a per-tuple PDT implementation (thanks to the shared value-space between replicas, memory consumption is independent of column width C). Therefore, increasing R leads to a memory consumption factor of $1 + \frac{12R}{CW}$. Note that such a memory consumption increase, reduces the memory fill time, causing the need for checkpointing to complete $O(R)$ times faster.

These three factors combined cause increasing R to reduce the time that disks are available to handle OLAP queries on the DSM mirror in the order $O(R^3)$. For example, in the high-end scenario with $R = 2$, the disks will be 66% busy handling random PAX lookups (R per query to establish the SID of the affected tuple in each replica), and checkpointing must run twice as fast while touching twice the volume, hence checkpointing overhead increases four-fold to 12%, leaving only 22% for handling queries to the DSM mirror. Therefore, in cases where the OLAP load is higher than 22% (assuming an update-idle system), overall performance equilibrium on a system that sustains the above described concurrent OLTP and OLAP load will show a slowdown in OLAP as well as OLTP throughput.

Given the above, we argue that replication with degree R should be offset by scaling disk resources D of the system by an equal factor R. In that case (i) the PAX random lookup load stays equal as there are R times more disks to handle requests concurrently and (ii) R times increased checkpointing volume is properly offset by similarly increased disk bandwidth. What remains is a need to also increase the available RAM size to hold the PDT trees of the replicas. However, with an average column width $W = 4$, and assuming $R = C$ as an upper limit on the replication degree, PDT RAM is independent of C and increases at most by factor 3, while disk storage increases by R.

A. Possible Extensions

Flash PDT. Our performance model showed that there is a price of at least a couple of percent of read-only bandwidth to paid up (and a limit imposed on tuple width and replication) for the per-update memory consumption of the PDT for it to fit in RAM. While it is left out of scope for this paper, we argue that it may well be beneficial to introduce a layer of PDTs in Flash memory.

The price per GB of raw Flash memory is around a factor 8 lower than RAM, thus it would be logical to make the Flash PDTs 8 times larger than the RAM PDTs, and this would allow to push back the checkpointing overhead by a factor 8. Note that these lower-layer Flash PDTs would experience random read access during search operations, sequential read access from checkpointing and sequential read and write access from the Apply() operator when RAM-based PDT data is migrated to Flash. These access patterns are all highly efficient in Flash memory, where only random writes are problematic.

We therefore view Flash PDTs as an extremely interesting opportunity to make merging differential updates cheaper, and to help PDTs cope with very wide or highly replicated tables.

VDT+PDT. The advantage of the PDT approach is that DSM queries do not need to read the SK columns and CPU merging cost is lower than a VDT approach. The advantage of the VDT approach is that an OLTP update query only needs a single IO, whereas the PDT approach needs to establish the affected (SID,RID) for each replica, leading to R IOs. By combining both approaches, low-cost CPU merging and single IO updates can be obtained. In the so-called VDT+PDT approach, updates are in principle handled using VDTs, but while read-only queries read the SK columns on some replica, as a piggybacking side effect, the (RID,SID)s of the updates that are merged in are established, removed from the VDT and inserted into the PDT. Therefore, most of the updates will be in the PDTs, such that queries profit from low-cost positional merging. However, the VDT drawback of having to read the SK in all queries remains in VDT+PDT (avoiding to read chunks that do not have a VDT update will not help much, as with the large DSM chunks the probability that at least one tuple has a VDT update remains high in many query loads).

Cooperative Checkpointing. Both DSM read-queries as well as checkpointing perform bulk reading and writing of large disk blocks (chunks, of e.g. 2MB). Intelligent scheduling of concurrent requests that do not have a strict ordering, has been shown to provide strong benefits for data warehousing workloads (Cooperative Scans [17]). Thus, an interesting avenue for further exploration is to see whether checkpointing can tolerate input chunks (to which PDT changes must be applied) to arrive out-of-order, allowing the checkpointing operator to be incorporated in the Cooperative Scans framework. This would allow checkpointing overhead to disappear as a performance factor, as it would process DSM chunks as they are brought in by DSM read-only queries.

VI. Related work

A number of papers investigated the differences between the DSM and NSM. DSM storage [5] has been identified to be beneficial not only for I/O benefits, but also for in-memory processing [1] To get these benefits without significant changes to the system architecture, PAX storage model [1] proposed using DSM data inside a traditional N-ary disk page, allowing good cache-performance. Multi-resolution block storage model (MBSM) [16] investigates the physical placement of the DSM-based data. In this approach, the good scan performance of DSM is preserved, while the cost of tuple reconstruction

is reduced, since values of different attributes from the same tuple are stored closer on disk than in the naive DSM implementation. The fractured-mirrors approach [10] suggests keeping two copies of data on two disks, one copy in NSM and another in DSM. This allows using the best model depending on the task, as well as combining both mirrors in a single query for even better performance. However, research on handling of updates in these papers is limited.

The idea of maintaining updates against large databases in a differential file for efficiency reasons, was introduced by Severance and Lohman in 1976 [12]. Copeland elaborated on this idea in a DSM context [5]. C-Store Log-structured merge-tree [9] defers and batches index updates to reduce index maintenance costs in insert/delete-heavy query workloads. Similarly, [7] proposes multi-level indexing in a warehousing environment. C-Store [14] borrows on these ideas by proposing to let updates go to a separate write-store (WS), while queries are being ran against an immutable read-store (RS). Updates from the WS are propagated to the RS on a temporary basis, and are therefore not immediately visible to subsequent queries. The original papers on fractured mirrors [10] also proposes using a differential file to buffer updates in memory. Comparing to our work, it uses a standard value-based approach. Also, authors do not discuss how to propagate the changes to the mirrors, while this paper proposes a number of optimization strategies, including partial checkpointing. Deferring of the updates has also been proposed in other areas, including view maintenance [4], [11] and updating inverted lists in information retrieval applications [13].

VII. Conclusions

This paper proposed a novel approach of handling the updates in a database, based on two fundamental innovations. First, merging of in-memory updates with disk-resident stable data has been significantly improved with a proposed *Positional Delta Tree* data structure, that reduces the need to perform expensive value-based checks. Secondly, storage model based on the idea of *Replicated Mirrors* allows to have quick NSM lookups necessary for update handling, while providing multiple DSM replicas for efficient scans. As a result, the proposed architecture achieves the read performance of DSM systems, only marginally impacted by the update merging process, while at the same time providing update facilities that can match those of standard row-based systems. We also presented that this architecture allows efficient, possibly partial, checkpointing and can provide transaction isolation based on the snapshot isolation model and layered PDTs.

References

[1] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, 2001.

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

[3] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proc. SIGMOD*, Vancouver, Canada, 2008.

[4] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. *SIGMOD Rec.*, 25(2):469–480, 1996.

[5] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, 1985.

[6] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[7] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 16–25, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[8] S. J. O'Connell and N. Winterbottom. Performing joins without decompression in a compressed database system. *SIGMOD Rec.*, 32(1):6–11, 2003.

[9] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4), 1996.

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

[11] A. Segev and J. Park. Updating distributed materialized views. *IEEE Trans. on Knowl. and Data Eng.*, 1(2):173–184, 1989.

[12] D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.*, 1(3), 1976.

[13] C. L. A. C. Stefan Büttcher. Hybrid Index Maintenance for Contiguous Inverted Lists. *Information Retrieval*, 11(3), 2008.

[14] M. Stonebraker et al. C-Store: A Column-oriented DBMS. In *Proc. VLDB*, 2005.

[15] Transaction Processing Performance Council. *TPC Benchmark H version 2.1.0*, 2002.

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

[17] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE*, 2006.