Positional Update Handling in Column Stores

Sándor Héman, Marcin Zukowski
VectorWise
Amsterdam, The Netherlands
sandor@vectorwise.com,
marcin@vectorwise.com

Niels Nes, Lefteris Sidirourgos,
Peter Boncz
CWI
Amsterdam, The Netherlands
niels@cwi.nl, lsidir@cwi.nl,
boncz@cwi.nl

ABSTRACT

In this paper we investigate techniques that allow for on-line updates to columnar databases, leaving intact their high read-only performance. Rather than keeping differential structures organized by the table key values, the core proposition of this paper is that this can better be done by keeping track of the tuple *position* of the modifications. Not only does this minimize the computational overhead of merging in differences into read-only queries, but this makes the differential structure oblivious of the value of the order keys, allowing it to avoid disk I/O for retrieving the order keys in read-only queries that otherwise do not need them – a crucial advantage for a column-store. We describe a new data structure for maintaining such positional updates, called the Positional Delta Tree (PDT), and describe detailed algorithms for PDT/column merging, updating PDTs, and for using PDTs in transaction management. In experiments with a columnar DBMS, we perform microbenchmarks on PDTs, and show in a TPC-H workload that PDTs allow quick on-line updates, yet significantly reduce their performance impact on read-only queries compared with classical value-based differential methods.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications

General Terms

Algorithms, Experimentation, Performance

1. MOTIVATION

Database systems that use columnar storage (DSM [8]) have recently re-gained commercial and research momentum [22, 3, 3, 21], 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-wise disk storage (NSM). Modern columnar data warehousing systems additionally employ techniques like compression, data clustering, and replication that make them truly read-friendly and write-unfriendly. In this paper we address the question how such systems can still efficiently provide update functionality.

Analytical query workloads inspect large amounts of tuples, but typically only a small subset of the columns; hence columnar storage avoids data access (I/O, CPU cache misses) for unused columns. Three techniques are often added to further increase performance: first, columnar systems often store data in a certain order (or clustering), to further reduce data access cost when range-predicates are used. Second, by using multiple replicas of such tables in different orders, the amount of predicates in a query workload that can benefit from this increases. Data compression, finally, reduces the data access cost needed to answer a query, because the data gets smaller, and more data fits in the buffer pool when compressed, reducing the page miss ratio.

The challenge facing updates in analytical column stores is that one I/O per column (replica) is needed, hence a single-row update that can be handled in a row-store with a single disk I/O will lead to many disk I/Os in a column store. These disk I/Os are scattered (random) I/Os, even if the database users only do inserts, because of the ordered or clustered table storage. Finally, compression makes updates computationally more expensive and complex since data needs to be de-compressed, updated and re-compressed before being written back to disk. Extra complications occur if the updated data no longer fits its original location.

Differential Updates. Some analytical columnar database systems, such as C-Store [22], handle updates by splitting their architecture in a "read-store" that manages the bulk of all data and a "write-store" that manages updates that have been made recently. Consequently, all queries access both base table information from the read-store, as well as all corresponding differences from the write-store and *merge* these on-the-fly. Also, in order to keep the write-store small (it resides typically in RAM), changes in it are periodically propagated into the read-store [22].

The topic of this paper is what data structures and algorithms should be used for implementing the write-store of a column-oriented database system that aims to support generic update workloads (inserts, deletes, modifications; batch or trickle). The natural "value-based" way to implement the write-store is to keep track of which tuples were deleted, inserted or modified in a RAM-resident data structure that organizes these items in the *sort key* (SK) order of the underlying read-store table and contains these keys; for example in a RAM friendly B-tree [6]. Such a data struc-

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.
SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$5.00.

ture can easily be modified for update queries, and can also be merged efficiently with the read-store for read-queries by scanning the leaves. An important side effect of the value-based approach, however, is that all queries must scan all sort key columns in order to perform the merge (even if those queries themselves do not need the keys) which reduces an important advantage of column stores.

PDT. In this paper, we propose a new data structure called the Positional Delta Tree (PDT). PDTs are similar to counted B-trees [23] but contain differential updates. They are designed to make merging in of these updates fast by providing the tuple *positions* where differences have to be applied at update time. Thanks to that, read queries that merge in differences do not need to look at sort key values. Instead, the merge operator can simply count down to the next position where a difference occurs, and apply it blindly when the scan cursor arrives there.

The key advantages of the PDT over value-based merging are *(i)* positional merging needs less I/O than value-based merging, because the sort keys do not need to be read, and *(ii)* positional merging is less CPU intensive than value-based merging, especially when the sort-key is a compound key (common in clustering approaches) and/or non-numerical attributes are part of the sort-key.

While the concept of positional differences sounds simple, its realization is complex, since positions in an ordered table are highly volatile: after an insertion or deletion, all positions of subsequent tuples change. The core of the problem is managing a mapping between two monotonically increasing numbers associated with tuples in a table, called the Stable ID (SID) and current Row ID (RID). The SIDs conceptually correspond to the consecutively ascending tuple positions found in the read-store, but are in the current tuple order not necessarily consecutive nor unique, just non-decreasing.

As PDTs capture updates, they also form an important building block in transaction management. We show that three layers of PDTs can be used to provide *lock-free* isolation. This lock-free property is a great advantage for an analytical DBMS designed not to compromise read-query performance for updates. We define the basic notions on which to base ACID properties of PDT based transactions, and provide two PDT transformation algorithms (Propagate and Serialize), pivotal for transaction management.

All in all, the main contribution of the paper is the PDT data structure and its application in column stores for providing efficient transactional updates without compromising read performance. We have fully implemented PDT-based update management in VectorWise[1] (originally based on the MonetDB/X100 prototype [3]), which we use for our experimental evaluation.

Outline. The basic concepts of PDTs are introduced in Section 2, and their working is illustrated by examples. Detailed algorithms for PDT update and MergeScan operators are provided in Section 3, as well as the Propagate and Serialize algorithms for three-layer PDT-based transaction management. We evaluate the performance of PDT based update management in Section 5, both using microbenchmarks, as well as in a TPC-H 30GB comparison between read-only, value-based and PDT-based query processing under an update load. Finally, in Section 5 we describe related work before concluding in Section 6.

[1]See www.vectorwise.com and www.ingres.com/vectorwise.

2. DIFFERENTIAL COLUMNAR UPDATES

We first introduce the basic assumptions, concepts and notations underlying PDTs, then explain them by example.

Ordered Tables. A column-oriented definition of the relational model is as follows: each column col_i is an ordered sequence of values, a TABLE$<\text{col}_1, \cdots, \text{col}_n>$ is a collection of related columns of equal length, and a *tuple* τ is a single row within TABLE. Tuples consist of values aligned in columns, i.e., the attribute values that make up a tuple are retrievable from a column using a single positional index value. This index we call the *row-id*, or RID, of a tuple.

Though the relational model is order-oblivious, column-stores are internally conscious of the physical tuple ordering, as this allows them to reconstruct tuples from columns cheaply, without expensive value-based joins [19]. The physical storage structures used in a column-store are designed to allow fast lookup and join by position using either B-tree storage with the tuple position (RID) as key in a highly compressed format [10] or dense block-wise storage with a separate sparse index with the start RID of each block.

Columnar database systems not only exploit the fact that all columns contain tuples in some physical order that is the same for all of them, but often also impose a particular physical order on the tuples, based on their value. That is, tuples can be ordered according to sequence of sort attributes S, typically determined by the DBA [22]. A sequence of attributes that defines a sort order, while also being a key of a table we call SK (the sort key). The motivation for such ordered storage, is to restrict scans to a fraction of the disk blocks if the query contains range- or equality-predicates on any prefix of the sort key attributes. As such, explicitly ordered storage is the columnar equivalent of index-organized tables (clustered indices) also used in a row-oriented RDBMS.

Ordering vs. Clustering. Multi-column sort orders are closely related to table clustering, that organizes a large (fact) table in groups of tuples, each group representing a cell in a multi-dimensional cube, where those dimensional values are typically reachable over a foreign-key link. Such multi-dimensional table clustering is an important technique to accelerate queries in data warehouses with a star- or snowflake-schema [17, 7]. In a column-store, multi-dimensional clustering translates into ordering the tuples in a table in such a way that tuples from the same cell are stored contiguously. This creates the same situation as in ordered columnar table storage, where physical tuple position is determined by the tuple values; and the machinery to handle table updates needs to respect this value-based tuple order.

Positional Updates. An update on an ordered table is one of (insert, delete, modify). A TABLE.insert(τ, i) adds a full tuple τ to the table at RID i, thereby incrementing the RIDs of existing tuples at RIDs $i \cdots N$ by one. A TABLE.delete(i) deletes the full tuple at RID i from the table thereby decrementing the RIDs of existing tuples at RIDs $i+1 \cdots N$ by one. A TABLE.modify(i, j, v) changes attribute j of an existing tuple at RID i to value v. Transactions may consist of a BATCH of multiple updates.

Differential Structures. We focus on the situations where positional updates happen truly scattered over the table. Note that in ordered table storage, even append-only warehouse update workloads drive column-stores into worst-case

territory if an in-place update strategy would be used, creating an avalanche of random writes, one for each affected row *and* column (modifies and deletes behave similarly). For this reason, and as argued previously, column-stores must use some differential structure DIFF, that buffers updates that have not yet been propagated to a stable table image.[2] As a consequence, all queries must not just scan stable table data from disk, but also apply the updates in DIFF as tuples are produced, using a Merge operator.

Checkpointing. Differential updates need to be eventually propagated to the stable storage to reduce the memory consumption and merging cost. While many strategies for this process can be envisioned, the simplest one is to detect a moment when the size of DIFF starts to exceed some threshold and to create a new image of the table with all updates that happened before applied. When this is ready, query processing switches over to this new image with the applied updates pruned from DIFF.

Stacking. Previous work on differential structures, e.g. differential files [20] and the Log-Structured Merge Tree [16] suggests *stacking* differential structures on top of each other, typically with the smallest structure on top, increasing in size towards the bottom. One reason to create such a multi-layer structure is to represent multiple different database snapshots, while sharing the bulk of the data structures (the biggest, lowest layers of the stack). Another reason is to limit the size of the topmost structure, which is the one being modified by updates, thus providing a more localized update access pattern, e.g. allowing to store different layers of the stack on different levels of the memory hierarchy (CPU caches, RAM, Flash, Disk). Each structure $\text{DIFF}_{t_2}^{t_1}$ in the stack contains updates from a time range $[t_1, t_2\rangle$:

$$\text{TABLE}_{t_2} = \text{TABLE}_{t_1}.\text{Merge}(\text{DIFF}_{t_2}^{t_1}) \quad (1)$$

If one keeps not one, but a stack of DIFFs (most recent on top), we can see the current image of a relational table as the result of merging the DIFFs in the stack one-by-one bottom-up with the initial TABLE_0. Here, TABLE_0 represents the "stable table", i.e. the initial state of a table on disk when instantiated (empty), bulk-loaded, or checkpointed.

DEFINITION 1. *Two differential structures are* **aligned** *if the table state they are based on is equal:*
Aligned$(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_a = t_c$

DEFINITION 2. *Two differential structures are* **consecutive** *if the time where the first difference ends equals the time the second difference starts:*
Consecutive$(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_b = t_c$

DEFINITION 3. *Two differential structures are* **overlapping** *if their time intervals overlap:*
Overlapping$(DIFF_{t_b}^{t_a}, DIFF_{t_d}^{t_c}) \Leftrightarrow t_a < t_d \leq t_b \vee t_c < t_b \leq t_d$

The time t rather than absolute time identifies the moment a transaction started, and could be represented by a monotonically increasing logical number, such as a Log Sequence Number (LSN). If a query that started at t works with just a single DIFF_t^0 structure, we shorten our notation to DIFF_t.

[2] Just like row-stores, at each commit column-stores need to write information in a Write-Ahead-Log (WAL), but that causes only sequential I/O, and does not limit throughput.

SID	store	prod	new	qty
0	London	chair	N	30
1	London	stool	N	10
2	London	table	N	20
3	Paris	rug	N	1
4	Paris	stool	N	5

Figure 1: TABLE_0

RID	
0	INSERT INTO inventory VALUES ('Berlin','table','Y',10)
1	INSERT INTO inventory VALUES ('Berlin','cloth','Y',5)
2	INSERT INTO inventory VALUES ('Berlin','chair','Y',20)

Figure 2: BATCH_1

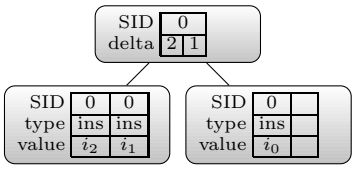

Figure 3: PDT_1

ins	store	prod	new	qty
i_0	Berlin	table	Y	10
i_1	Berlin	cloth	Y	5
i_2	Berlin	chair	Y	20

del store prod

new new qty qty

Figure 4: VALS_1

RID vs. SID. We define *stable-id* $\text{SID}(\tau)$, to be the position of a tuple τ within the TABLE_0. (i.e. the "stable" table on disk) starting the tuple numbering at 0, and define the *row-id* $\text{RID}(\tau)_t$, to be the position of τ at time t; thus $\text{SID}(\tau) = \text{RID}(\tau)_0$. $\text{SID}(\tau)$ never changes throughout the lifetime of tuple τ (except for checkpoints). We actually define $\text{SID}(\tau)$ to also have a value for newly inserted tuples τ: they receive a SID such that it is larger than the SID of all preceding stable tuples (if any) and equal to the first following stable tuple (if any). Conversely, we also define a RID value for a stable tuple that was deleted ("ghost tuples") to be one more than the RID of the preceding non-ghost tuple (if any) and equal to the first following non-ghost tuple (if any).

If the tuple τ is clear from the context, we abbreviate $\text{SID}(\tau)$ to just SID (similar for RID), and if the time t is clear from the context (e.g. the start time of the current transaction) we can also write just RID instead of RID_t. In general, considering the stacking of PDTs and thus having the state of the table represented at multiple points in time, we define Δ as the RID difference between two time-points:

$$\Delta_{t_2}^{t_1}(\tau) \quad = \text{RID}(\tau)_{t_2} - \text{RID}(\tau)_{t_1} \quad (2)$$

which in the common case when we have one PDT on top of the stable table ($t_1 = 0$) is RID minus SID:[3]

$$\Delta_t(\tau) \quad = \text{RID}(\tau)_t - \text{SID}(\tau) \quad (3)$$

If we define the SK-based Table time-wise difference as:

$$\text{MINUS}_{t_2}^{t_1} \quad = \{\tau \in \text{TABLE}_{t_1} : \nexists \gamma \in \text{TABLE}_{t_2} : \tau.\text{SK} = \gamma.\text{SK}\} \quad (4)$$

then we can compute Δ as the number of inserts minus the number of deletes before τ:

$$\begin{aligned} \Delta_{t_2}^{t_1}(\tau) \quad = \quad & |\{\gamma \in \text{MINUS}_{t_1}^{t_2} : \text{RID}(\gamma)_{t_2} < \text{RID}(\tau)_{t_2}\}| - \\ & |\{\gamma \in \text{MINUS}_{t_2}^{t_1} : \text{SID}(\gamma) < \text{SID}(\tau)\}| \end{aligned} \quad (5)$$

2.1 PDT by Example

We introduce the Positional Delta Tree (PDT) using a running example of a data warehouse table inventory, with sort key (store,prod), shown in Figure 1. This TABLE_0 is already populated with tuples, using an initial bulk load, and persistently stored on disk. Initially its RIDs are identical to the SIDs. Note that RIDs and SIDs are conceptual sequence numbers; they are not stored anywhere.

[3] When the context $\text{PDT}_{t_2}^{t_1}$ is clear, we sometimes slightly imprecisely refer to RID_{t_1} as SID and RID_{t_2} simply as RID, even in case of a stacked PDT ($t_1 > 0$).

SID	store	prod	new	qty	RID
0	Berlin	chair	Y	20	0
0	Berlin	cloth	Y	5	1
0	Berlin	table	Y	10	2
0	London	chair	N	30	3
1	London	stool	N	10	5
2	London	table	N	20	4
3	Paris	rug	N	1	6
4	Paris	stool	N	5	7

Figure 5: TABLE$_1$

```
UPDATE inventory SET qty=1
WHERE store='Berlin' and prod='cloth'
UPDATE inventory SET qty=9
WHERE store='London' and prod='stool'
DELETE FROM inventory
WHERE store='Berlin' and prod='table'
DELETE FROM inventory
WHERE store='Paris' and prod='rug'
```

Figure 6: BATCH$_2$

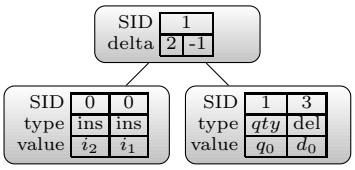

Figure 7: PDT$_2$

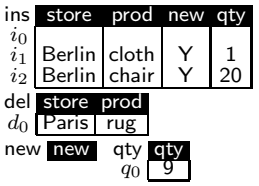

Figure 8: VALS$_2$

SID	store	prod	new	qty	RID
0	Berlin	chair	Y	20	0
0	Berlin	cloth	Y	1	1
0	London	chair	N	30	2
1	London	stool	N	9	3
2	London	table	N	20	4
3	Paris	rug	N	1	5
4	Paris	stool	N	5	5

Figure 9: TABLE$_2$

```
INSERT INTO inventory
  VALUES ('Paris','rack','Y',4)
INSERT INTO inventory
  VALUES ('London','rack','Y',4)
INSERT INTO inventory
  VALUES ('Berlin','rack','Y',4)
```

Figure 10: BATCH$_3$

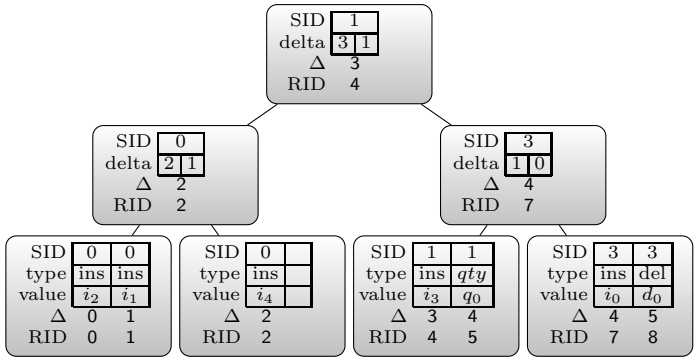

Figure 11: PDT$_3$ with annotated Δ,RID=SID+Δ

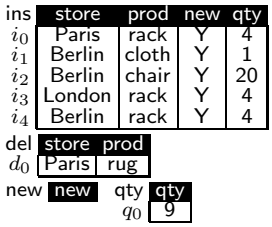

Figure 12: VALS$_3$

SID	store	prod	new	qty	RID
0	Berlin	chair	Y	20	0
0	Berlin	cloth	Y	1	1
0	Berlin	rack	Y	4	2
0	London	chair	N	30	3
1	London	rack	Y	4	4
1	London	stool	N	9	5
2	London	table	N	20	6
3	Paris	rack	Y	4	7
3	Paris	rug	N	1	8
4	Paris	stool	Y	4	8

Figure 13: TABLE$_3$

Inserts. We now execute the insert statements from Figure 2 and observe the effects to the PDT structure in Figures 3 and 4. Because the inventory table is kept sorted on (store,prod), the new Berlin tuples get inserted at the beginning. Rather than touching the stable table, the updates are recorded in a PDT shown in Figure 3, which is a B+ tree like structure that holds its data in the leaves.[4] The non-leaf nodes of the PDT (here only the root node) contain a SID as separator key, and a delta (explained below) that allows to compute the RID⇔SID mapping, because after the inserts these will no longer be identical. The separator SID in the inner nodes is the minimum SID of the right subtree. Figure 3 shows that the inserts all get the same SID 0, which is thus not unique in the PDT. Among them, the left-to-right leaf order in the PDT determines their order in the final results, which is displayed in Figure 5.

Value Space. The leaf nodes of the PDT store the SID where the update applies, as well as the type of update, as well as a reference (or pointer) to the new tuple values. Because the type of information to store for the various updates types (delete,insert,modify) differs, these are stored in separate "value tables". In terms of our notation, we have:

$$\text{DIFF} = (\quad \text{PDT}, \text{VALS} \quad) \tag{6}$$

$$\text{VALS} = (\quad \text{ins}\langle \text{col}_1, \cdots, \text{col}_n\rangle, \text{del}\langle \text{SK}\rangle, \; col_1\langle \text{col}_1\rangle, \cdots, col_n\langle \text{col}_n\rangle \quad) \tag{7}$$

Thus, each PDT has an associated "value space" that contains multiple value tables: one insert-table with new tuples, one delete-table with deleted key values, and for each column a single-column modify-table with the modified values. The value space resulting from the insert statements in Figure 4 has all tables empty except the insert table.

Delete. Moving to Figures 5-8 we show the effects of delete and modify statements. Deletions produce a leaf node with the "del" type and a reference to a table in the value space that contains the sort key values of the deleted stable "ghost" tuples (d_0 refers to (Paris,rug) in Figure 8). We still display tuples deleted from TABLE$_0$ but greyed out (that is, (Paris,rug)); as these are not visible anymore to queries. Note that the other deleted tuple, (Berlin,table), is not stable (i.e. in TABLE$_0$) therefore really disappeared. The reason we keep track of ghost tuples, is that the SIDs of newly inserted tuples, always respect the original SK order of the stable table, as explained later ("Respecting Deletes").

Modify. The type field of modify leaves contains a reference to the modified column. In the value space, each column has a separate single-column table that holds modified values. In our PDTs, we indicate modifications using the column name in italics (*qty*). Thus, the first value of the right leaf states that stable tuple SID=1 (i.e. (London,stool,N,10)) had its qty column set to 9, per the q_0 reference to the qty modification table in the value space of Figure 8. Note that the key surrogates of the value space tables e.g. i_1, d_0, q_0 are not drawn as part of those tables, as simple numerical offsets can be used to save memory in the value space.

Handling of modify and delete is different if the updated data already resides in the PDT; it can then be changed there directly. That is, a deletion of an inserted tuple removes all traces of it from the PDT. One may observe this in case of the the i_0 insert of (Berlin,table,Y,10). In case column values of a stable tuple have modify entries in the PDT, all these are removed and get replaced by a single deletion entry (not in this example). Finally, a modification of a value that was already modified or inserted, changes the value stored in PDT. In our example, this happened in case of the i_1 insert of (Berlin,cloth,Y,5), which got its qty changed into 1 in Figure 8.

Finally note that when modifying the SK columns of a tuple τ, this is handled as a deletion of τ followed by an insert of the updated tuple.

[4] The tree fan-out is 2 for presentation purposes. Given its use as a cache/memory-resident data structure, node sizes should be a few cache lines long, e.g. a fan-out of 8 or 16.

RID ⇔ SID. Figures 9-12 show the effect of three additional inserts, producing a final table state of Figure 13. Of course, what is stored on disk is still $TABLE_0$ shown in Figure 1. At this stage, the PDT has grown to a tree of three levels. To illustrate the way RIDs are mapped into SIDs, the PDT_3 in Figure 11 is annotated with the running Δ, as well as with the RID. Note that Δ, which is the number of inserts minus the number of deletes so far, as defined in equation(5), takes into account the effect of all *preceding* updates, hence it is still zero at the leftmost insert. For the inner nodes, the annotated value is the separator value; the lowest RID of the right subtree. The PDT maintains the delta field in the inner nodes, which is the contribution to Δ of all updates in the subtree below it. Note that this number can be negative, as in Figure 7. It is thus easy to see that by summing the respective delta values on a root-to-leaf path, we obtain the Δ value for the first update entry in that leaf. By counting inserts and deletes from left-to-right in the leaf, we can compute Δ, and thus RID, for any leaf value. Since lookup as well as updates to the PDT only involve the nodes on a root-to-leaf path, cost is logarithmic in PDT size.

Respecting Deletes. One of the subtleties in the PDT structure is the interaction between deletions of stable tuples (that become "ghost" tuples) and subsequent insertions of new tuples. We see this interaction in the SID=3 that the newly inserted (Paris,rack) received; its place in the table is exactly before a deleted tuple (Paris,rug). If we would not consider ghost tuples, the new tuple could have received SID=4, as that is the first subsequent valid stable tuple. We chose to respect the order of ghost tuples, mainly to reduce index maintenance cost; because its effect is that any SK ⇔ SID mapping (index) created on $TABLE_0$ remains valid in future versions of the table. Analytical database systems use a variety of sparse indexing techniques (e.g. Zone Maps [11], Knowledge Grid [21] and Small Materialized Aggregates [14]) to exploit exact and non-exact clustering in data, allowing table scans to skip large tuple ranges that cannot contain valid answers.

store	prod	SID
London	stool	1
Paris	rug	3

Sparse Index

The table to the right is a classical and simple variant of such, namely a sparse index, created on $TABLE_0$, that tells that all tuples with SK ≤(London,stool) can be found in SID≤1, all tuples with SK ≤(Paris,rug) have SID≤3, and SID> 3 holds for the rest. When handling a query with range-predicate on SK, the sparse index can be used to obtain a SID range (or potentially multiple SID-ranges) to pass on to a table scan. The table scan will fetch blocks for these SID ranges in order, for all needed columns, and additionally merge in differences using the PDT. Consider the following query, posed against $TABLE_3$:

```
SELECT qty FROM inventory
WHERE store = 'Paris' and prod < 'rug'
```

it will obtain from the sparse index a SID range ⟨1,3] to restrict the scan with. Respecting ghost tuples makes sure the qualifying (Paris,rack) tuple is in range, whereas disregarding it (SID=4) would cause problems. If ghosts were disregarded, one could avoid storing key values in the delete-table, but sparse indices would have to be kept up-to-date. The ghost-respecting semantics we use here, allows sparse indices to be kept "stale", without update overhead.

VDTs. Other column-stores that use differential update processing, such as MonetDB [2] use a simpler "value-based" approach, that consists of an insert table that contains all columns and holds all inserted and modified tuples, and a deletion table, that only holds the sort key values of deleted or modified tuples.

This is illustrated to the right with the insertion table on top, and the deletion table at the bottom. Both tables are kept organized in sort key order, to facilitate merging these delta tables with the stable table. Therefore, it is natural to implement such tables as B-trees, hence we call this approach the Value-based Delta Tree (VDT).

store	prod	new	qty
Berlin	chair	Y	20
Berlin	cloth	Y	1
Berlin	rack	Y	4
London	rack	Y	4
London	stool	N	9
Paris	rack	Y	4

store	prod
Paris	rug
London	stool

VDT: ins+del table

A VDT based DBMS should replace each table (range) scan of our example inventory table by:

```
SELECT * FROM ins
 UNION (SELECT * FROM inventory WHERE NOT EXISTS
(SELECT * FROM del
 WHERE inventory.store = del.store AND
       inventory.prod = del.prod))
```

while the above seems intimidating, the fact that all three tables are kept organized in the order of the sort key, allows to execute this on the physical relational algebra level as an (efficient) linear merge-union and -difference:

```
MergeUnion[store,prod](Scan(ins),
 MergeDiff[store,prod](Scan(inventory),Scan(del)))
```

Merging: PDT vs VDT. The main advantage of the PDT over the VDT approach is that merging in of updates involves Merge-Union/-Diff comparisons on the sort keys (here (store,prod)), which is *(i)* computationally intensive and *(ii)* forces the database system to read those sort keys off disk for every query, even in cases where the query does not involve these keys. The positional-only merging that PDTs offer, on the other hand, avoids reading key columns in these cases, an important advantage in a column-store.

Some systems (e.g. Vertica) employ optimizations improving the performance of value-based delta structures. For example, it is often possible to perform (parts of) a query on the stable and delta data separately, and only combine the results at the end. Also, deletions can be handled by using a boolean column marking the "alive" status of a given tuple, stored in RAM using some updatable variant of the compressed bitmap index (a non-trivial data structure, see e.g. [5]). However, in cases where the order of tables needs to be maintained, e.g. for queries using merge joins, these solutions still need a CPU-intense key-based `MergeUnion`, and require scanning of all keys in the queries. As such, we exclude these techniques from our evaluation.

3. PDT ALGORITHMS

In this section, we first provide basic PDT algorithms (merge,updates). Subsequently we discuss how PDTs can be a basic building block in transaction processing and provide two pivotal PDT algorithms for this. The Propagate algorithm combines two consecutive PDTs in a single one, and the Serialize algorithm makes two overlapping PDTs consecutive. The latter task may be impossible, precisely under the circumstances where two concurrent transactions are conflicting, hence one should be aborted. Hence, we show the tight interaction, even synergy, of positional difference management and transaction management, both in terms of isolation as well as conflict detection.

Algorithm 1 PDT.FindLeafByRid(rid)
PDT.FindLeftLeafByRid(rid)
PDT.FindLeafByRidSid(rid, sid)
PDT.FindLeafBySid(sid)
Finds the rightmost leaf which updates a given rid. Versions that find the leftmost leaf, or search by sid or (sid, rid) are omitted

```
1: node = this; δ = 0
2: while is_leaf(node) = false do
3:    for i = 0 to node_count(node) do
4:       δ = δ + node.delta[i]
5:       if rid < node.sids[i] + δ then
6:          δ = δ − node.delta[i]
7:          break from inner loop
8:    node = node.child[i]
9: return (node, δ)
```

3.1 Basics

A memory efficient PDT implementation in C is as follows:

```
typedef struct { uint64 16:n, 48:v } upd_t;
#define INS        65535
#define DEL        65534
#define PDT_MAGIC  0xFFFFFFFFFFFF
#define is_leaf(n) ((n).sid[0] != PDT_MAGIC)
#define type(x)    update[x].n /* INS, DEL or ..*/
#define col_no(x)  update[x].n /* column number */
#define value(x)   update[x].v /* valspace offset */
#define F 8

                                typedef struct {
typedef struct {                   uint64 sid[F];
   uint64 sid[F];                  sint64 delta[F];
   upd_t update[F];                void*  child[F];
} PDT_leaf;                     } PDT_intern;
```

This implementation minimizes the leaf memory size, which is important because there will be finite memory for buffering PDT data, thus the PDT memory footprint determines the maximum update throughput that can be sustained in the time window it takes to perform a checkpoint (that allows to free up memory). The leaf of a PDT consists of a SID (large integer), the update type field, and a value reference field. The update type has a distinct value for INS,DEL and for each column in the table (for modifies), hence an ultra-wide 65534 column table fits two bytes. The offset reference stored in value should fit 6 bytes, hence the PDT storage consumption per modification is 16 bytes.

The fan-out F=8 is chosen here such that leaf nodes are 128 bytes wide, hence aligned with (typically) two CPU cache lines. The child pointer in the internal node can either point to another internal node or a leaf, hence its anonymous type. Since internal nodes need only F-1 SIDs, we can fill the first SID value of internal nodes with a special code that distinguishes them from leaves.

The PDT is a B+-tree like tree that holds two non-unique monotonically increasing keys SID and RID.

THEOREM 1. *(SID, RID) is a unique key of a table.*

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 cannot 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.*

Algorithm 2 Merge.next()
The Merge operator has as state the variables $pos, rid, skip$, and DIFF and Scan; resp. a PDT and an input operator. Its next() method returns the next tuple resulting from a merge between Scan and a left-to-right traversal of the leaves of DIFF. pos, rid are initialized to 0, $leaf$ to the leftmost leaf of DIFF and $skip = leaf.sid[0]$ (if DIFF is empty, $skip = \infty$). Note the new rid is attached to each returned tuple.

```
1: newrid = rid
2: rid = rid + 1
3: while skip > 0 or leaf.type[pos] ≡DEL do
4:    tuple =Scan.next()
5:    if skip > 0 then
6:       skip = skip − 1
7:       return (tuple, newrid)
      // delete: do not return the current tuple
8:    (pos, leaf) =DIFF.NextLeafEntry(pos, leaf)
9:    skip = leaf.sid[pos] − tuple[sid]
10: if leaf.type[pos] ≡INS then
11:    tuple = leaf.value[pos]
12:    (pos, leaf) =DIFF.NextLeafEntry(pos, leaf)
13: else
14:    tuple =Scan.next()
15:    while leaf.sid[pos] ≡ tuple.sid do // MODs same tuple
16:       col = leaf.col_no[pos]
17:       tuple[col] = leaf.values[pos][col])
18:       (pos, leaf) =DIFF.NextLeafEntry(pos, leaf)
19: skip = leaf.sid[pos] − tuple[sid]
20: return (tuple, newrid)
```

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.*

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.*

Thus, we can search the PDT for exact match on (SID,RID), and for first-leaf resp. last-leaf on RID and SID alone, using rather standard tree search, listed in Algorithm 1.

MergeScan. We now move our attention to the MergeScan operator which merges a basic Scan on a stable table with the updates in a PDT. Algorithm 2 shows the next() method one would typically implement in a relational query processing engine for such an operator; the task of this method is to return the next tuple. The idea here is that *skip* represents the distance in position until the next update; until which tuples are just passed through. When an update (INS,DEL,modify) is reached, action is undertaken to apply the update in the PDT to the output stream.

The listed algorithm is simplified for a single-level PDT. In case of multiple layers of stacked PDTs, MergeScan consists of a Scan followed by multiple Merge operations. Merge can also be optimized; the version we used in our evaluation was adapted to use block-oriented pipelined processing [18, 3], in which each next() method call returns a block of tuples rather than just one. As the *skip* value is typically large, in many cases this allows to pass through entire blocks of tuples unmodified from the Scan, without copying overhead.

3.2 Update Operations

Adding a new modification or deletion update to a PDT only requires a RID to be unambiguous, as ghost records

Algorithm 3 PDT.AddInsert($sid, rid, tuple$)
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) = this.FindLeafBySidRid(sid, rid)
2: while leaf.sid[pos] < sid or leaf.sid[pos] + δ < rid do
3:    if leaf.type[pos] ≡INS then
4:       δ = δ + 1
5:    else if leaf.type[pos] ≡DEL then
6:       δ = δ − 1
7:    pos = pos + 1
   // Insert update triplet in leaf
8: this.ShiftLeafEntries(leaf, pos, 1)
9: leaf.type[pos] =INS
10: leaf.sid[pos] = rid − δ
11: offset = this.AddToInsertSpace(tuple)
12: leaf.value[pos] = offset
13: this.AddNodeDeltas(leaf, 1)
```

– which may share RIDs with a succeeding tuple – cannot be targets for deletion or modification. We only need to make sure that the new update goes to the end of an update chain that shares the same RID. 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 functions for adding a new modification or deletion update to a PDT are outlined in Algorithms 4 and 5, respectively. B-tree specific details, like splitting full leaves and jumping from one leaf to its successor, are left out for brevity. The omitted function AddNodeDeltas($leaf, val$) adds a (possibly negative) value val to all `delta` fields of the inner nodes on the path from the root to $leaf$.

Algorithm 4 PDT.AddModify(rid, col_no, new_value)
Finds the rightmost leaf containing updates on a given rid, adding a new modification triplet at index pos, or modify in-place.

```
1: (leaf, δ) = this.FindLeafByRid(rid)
2: (pos, δ) = this.SearchLeafForRid(leaf, rid, δ)
3: while leaf.sid[pos] + δ ≡ rid and leaf.type[pos] ≡DEL do
4:    pos = pos + 1; δ = δ − 1
5: if leaf.sid[pos] + δ ≡ rid then // In-place update
6:    if leaf.type[pos] ≡INS then
7:       offset = this.ModifyInsertSpace(pos, col_no, new_value)
8:    else
9:       offset = this.ModifyModifySpace(pos, col_no, new_value)
10:   leaf.value[pos] = new_value
11: else // add new update triplet to leaf
12:   this.ShiftLeafEntries(leaf, pos, 1)
13:   leaf.col_no[pos] =col_no
14:   leaf.sid[pos] = rid − δ
15:   offset = this.AddToModifySpace(pos, col_no, new_value)
16:   leaf.value[pos] = offset
```

One question is how RID and SID values are obtained. Deletion and modification SQL requests identify the to-be-updated tuples using a query. In this query, a MergeScan generates tuples by merging the stable table with the PDT, which results in the RIDs we want.

For inserts, we need (SID,RID) to insert a tuple with SK=`sk` in the right position of the PDT using Algorithm 3. That is, we need to find the tuple before which the insert should take place (if any). It can be found with a query that computes the minimum RID whose tuple has a larger SK:

```
SELECT rid FROM inventory
  WHERE SK > sk ORDER BY rid LIMIT 1
```

This query could profit in the table-scan from the order on SK, exploiting e.g. a sparse index, as mentioned in "Respecting Deletes", and recognising `rid`s are ordered, scan

Algorithm 5 PDT.AddDelete(rid, SK_values)
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) = this.FindLeafByRid(rid)
2: (pos, δ) = this.SearchLeafForRid(leaf, rid, δ)
3: while leaf.sid[pos] + δ ≡ rid and leaf.type[pos] ≡DEL do
4:    pos = pos + 1; δ = δ − 1
5: if leaf.sid[pos] + δ ≡ rid then // In-place update
6:    if leaf.type[pos] ≡INS then // Delete existing insert
7:       this.ShiftLeafEntries(leaf, pos, −1)
8:       this.AddNodeDeltas(leaf, −1)
9:       return
10: else // add new update triplet to leaf
11:   this.ShiftLeafEntries(leaf, pos, 1)
12:   leaf.sid[pos] = rid − δ
13: leaf.type[pos] =DEL
14: offset = this.AddToDeleteSpace(SK_values)
15: leaf.value[pos] = offset
16: this.AddNodeDeltas(leaf, −1)
```

Algorithm 6 PDT.SKRidToSid($tuple[SK], rid$)
This routine takes a partial tuple of sort key attribute values together with the RID of the tuple, and returns the SID within the underlying stable image where the tuple should go. This procedure is needed when we propagate inserts from a higher level PDT to this PDT, as we need to locate the inserts' exact position with respect to deleted stable tuples.

```
1: (leaf, δ) = this.FindLeafByRid(rid)
2: (pos, δ) = this.SearchLeafForRid(leaf, rid, δ)
3: while leaf.sid[pos] + δ ≡ rid and leaf.type[pos] ≡DEL and
   tuple[SK] > this.getDelValue(leaf.value[pos]) do
4:    pos = pos + 1; δ = δ − 1
5: sid = rid − δ
6: return (sid)
```

only until the first qualifying tuple. From the obtained RID, we then need to search the PDT using the SK values to identify the SID, using Algorithm 6. The SK values are needed to un-tie multiple inserts at the same SID.

3.3 Transactions

Stacked PDTs can be used as a building block in transactions as depicted in Figure 14. The goal here is to provide snapshot isolation, where each new query gets a private isolated snapshot of a table by fully copying a PDT. While database systems that offer snapshot isolation as the highest consistency level (e.g. Oracle) are very popular, they still fail to provide full serializability. However, recent research [4] shows that it is possible to guarantee full serializability in a snapshot-based system using extra bookkeeping of read tuples.

To keep the full copying of a PDT efficient, the size of such a PDT must be small, typically smaller than the CPU cache size. This small, top-most, PDT is the only PDT being modified by committing transactions, and we call it the "Write-PDT". Actually, copying is not always required, because if no commits happened between two starting transactions, the new transactions can share the same copy of the Write-PDT with its predecessor. When a transaction commits, it inserts its updates in the original Write PDT (not its private copy!). As concurrent read-queries only read their private copies, they are isolated from these commits.

Propagate. Algorithm 7 lists the Propagate operator that adds to PDT R holding updates from $[t_0, t_1\rangle$, all updates of

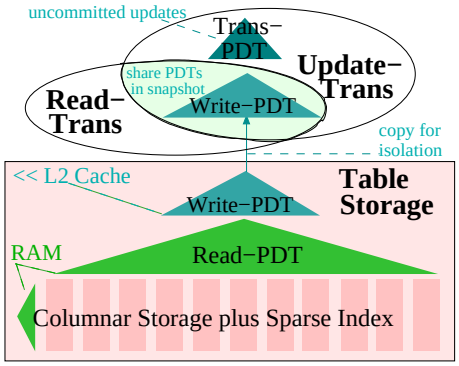

Figure 14: Snapshot Isolation with Layered PDTs

a consecutive PDT W of time range $[t_1, t_2\rangle$:

$$\mathrm{R}^{t_0}_{t_2} \leftarrow \mathrm{R}^{t_0}_{t_1}.\mathrm{Propagate}(\mathrm{W}^{t_1}_{t_2}) \quad (8)$$

with $\mathrm{TABLE}_0.\mathrm{Merge}(\mathrm{R}^{t_0}_{t_2}) = \mathrm{TABLE}_0.\mathrm{Merge}(\mathrm{R}^{t_0}_{t_1}).\mathrm{Merge}(W^{t_1}_{t_2})$.

This operation is used periodically, when the size of the Write-PDT grows too large, to migrate all its contents to a lower-layer "Read-PDT". Architecturally, the idea is that the Write-PDT is CPU cache resident, while the Read-PDT is RAM resident, with the stable table being disk resident. The Propagate algorithm takes all updates in a higher-layer PDT p in left-to-right leaf order and applies them to the current PDT. Observing that the SIDs of one layer are the RIDs of the layer below, Propagate converts RIDs to SIDs while performing its job.

Finally, in order to support multi-query transactions, it should be possible for subsequent queries in the same transaction to see the effects of a previous update in that still uncommitted transaction. This can be supported by adding on top a third layer of PDTs, namely the "Trans-PDT". This Trans-PDT is private to a transaction and initially empty. The effects of any updates in the transaction are added only to the Trans-PDT.[5] In all, the table image a transaction that started at time t sees, is based on the stable table, modified by updates until r in the (typically RAM resident) Read-PDT R, and updates until w in the (typically CPU cache resident) Write-PDT W, with updates the transaction itself carried out stored in the Trans-PDT T:

$$\mathrm{TABLE}_t = \mathrm{TABLE}_0.\mathrm{Merge}(\mathrm{R}^0_r).\mathrm{Merge}(\mathrm{W}^r_w).\mathrm{Merge}(\mathrm{T}^w_t) \quad (9)$$

When the transaction commits, it should propagate the changes from the Trans-PDT to the master Write-PDT. However, we cannot blindly use Algorithm 7, because concurrent transactions may have committed, hence the time ranges represented by the master Write-PDT may *overlap* with the Trans-PDT of the committing transaction.

Overlapping Transactions. At the start of a transaction x at time t_0, a copy of the Write-PDT W^{t_0} is made named Wx^{t_0}, and an empty Trans-PDT Tx^{t_0} is created. Until x commits at t_2, updates are added to the Trans-PDT denoted at commit time $Tx^{t_0}_{t_2}$. If no transaction committed in the meantime, we may just use Algorithm 7 to propagate the updates directly to the master Write-PDT: $W_{t_2} = W_{t_0}.\mathrm{Propagate}(Tx^{t_0}_{t_2})$ to get a database state at time

[5] A fourth level "Query-PDT" can be used in certain queries, to protect them from seeing their own changes. When such a query finishes, its Query-PDT is propagated to its Trans-PDT and removed.

Algorithm 7 PDT.Propagate(W)
Propagates the updates present in argument PDT W to this PDT R. It assumes that W is consecutive to R.

```
1:  leaf = W.FindLeafBySid(0)
2:  pos = δ = 0
3:  while leaf do // Iterate over input updates
4:    rid = leaf.sid[pos] + δ
5:    if leaf.type[pos] ≡INS then // Insert
6:      sid =SKRidToSid(leaf.values[pos][SK], rid)
7:      R.AddInsert(sid, rid, leaf.values[pos])
8:      δ = δ + 1
9:    else if leaf.type[pos] ≡DEL then // Delete
10:     R.AddDelete(rid, leaf.values[pos][SK])
11:     δ = δ − 1
12:   else // Modify
13:     R.AddModify(rid, col_no, leaf.values[pos][col_no])
14:   (pos, leaf) = W.NextLeafEntry(pos, leaf)
```

t_2 that reflects x. However, if a transaction y committed at t_1, where $t_0 < t_1 < t_2$: we need to do two things:
(1) In snapshot isolation, a conflict occurs if the write-set of the transactions overlaps, so we must check for updates in Trans-PDTs Tx and Ty that modify the same tuple.
(2) If no conflicts exist, we need to propagate the changes from Tx into the current master Write-PDT W_{t_1}.

To perform (1), the updates in Tx and Ty should be *aligned*, i.e. relative to the *same* database snapshot (matching SID domains). To perform (2), updates in Tx should be *consecutive* with respect to the updates in Ty, i.e. relative to the database snapshot as *produced* by merging updates committed by Ty (SID domain of Tx matches RID domain of Ty). Assuming for now that x and y are the only concurrent transactions, they are guaranteed to have started using equal write-PDT snapshots (i.e. no other transaction can have committed in the meantime). This makes Tx and Ty aligned. We can now check for conflicting updates in Tx and Ty by analyzing the update SIDs in both PDTs (in ordered and synchronized fashion). While we are checking for conflicts, we can also convert the SIDs of the updates in Tx (the committing transaction) so that they become relative to the RID domain as produced by transaction y, thereby serializing x and y. If Ty is a PDT^{t0}_{t1}, serializing means that we are transforming PDT Tx^{t0}_{t2} into a PDT $T'x^{t1}_{t2}$, which allows us to perform (2).

Serialize. Algorithm 8 lists the Serialize() routine that performs this transformation, but also raises an error (returns false) if there are conflicting updates that make the transposition illegal. In case of such a conflict, the committing transaction must be aborted. Note that the conflict checking performed is write-write on the tuple-level and even allows to reconcile modifications of different attributes of the same tuple (by the omitted CheckModConflict() routine).

$$T'x^{t_1}_{t_2} \leftarrow Tx^{t_0}_{t_2}.\mathrm{Serialize}(Ty^{t_0}_{t_1}) \quad (10)$$

Commit. Until now we only considered two concurrent transactions, but Algorithm 9 extends the idea to an arbitrary number of concurrent transactions.

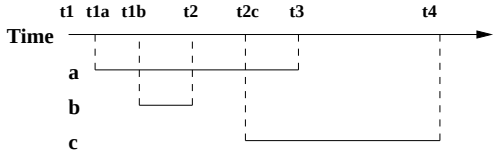

Figure 15: Three Concurrent Transactions

Algorithm 8 PDT.Serialize(Ty)
This method is invoked on PDT Tx with an aligned Ty as input, and checks the updates in Tx (the newer PDT) for conflicts with an earlier committed transaction Ty. FALSE is returned if there was a conflict. Its effect on Tx (referred to as $T'x$) is to become consecutive to Ty, as its SIDs get converted to the RID domain of Ty.

```
1: imax = Tx.count(); jmax = Ty.count()
2: i = j = δ = 0
3: while i < imax do // Iterate over new updates
4:    while j < jmax and Ty[j].sid < Tx[i].sid do
5:       if Ty[j].type ≡ INS then
6:          δ = δ + 1
7:       else if Ty[j].type ≡ DEL then
8:          δ = δ − 1
9:       j = j + 1
10:   if Ty[j].sid ≡ Tx[i].sid then // potential conflict
11:      if Ty[j].type ≡INS and Tx[i].type ≡INS then
12:         if Ty[j].value < Tx[i].value then
13:            δ = δ + 1; j = j + 1
14:         else if Ty[j].value ≡ Tx[i].value then
15:            return false// key conflict
16:         else
17:            Tx[i].sid = Tx[i].sid + δ
18:            i = i + 1
19:      else if Ty[j].type ≡DEL then
20:         if Tx[i].type ≢INS then
21:            return false
22:         else // Never conflict with Insert
23:            Tx[i].sid = Tx[i].sid + δ
24:            δ = δ − 1; i = i + 1
25:      else // Modify in Ty
26:         if Tx[i].type ≡INS then
27:            Tx[i].sid = Tx[i].sid + δ
28:            i = i + 1
29:         else if Tx[i].type ≡DEL then // DEL-MOD conflict
30:            return false
31:         else if CheckModConflict() then // MOD-MOD
32:            return false
33:   else // Current SID in Ty is bigger than in Tx
34:      Tx[i].sid = Tx[i].sid + δ // Only convert SID
35:      i = i + 1
36: return true
```

For each recently committed transaction z_i that overlaps with a still running transaction x we keep their *serialized* Trans-PDTs $T'z_i$ alive in the set TZ. A reference counting mechanism ensures that $T'z_i$-s are removed from TZ as soon as the last overlapping transaction finishes. Basically, all $T'z_i$ are consecutive and hold the changes that the transaction z_i applied to the previous database state. The creation of such a $T'z_i$ is in fact a by-product of the fact that z_i committed. Like described in the two-transaction case, committing entails using the Serialize algorithm to transform the Trans-PDT Tx of the committing transaction possibly multiple times; once for each overlapping transaction $z_i \in TZ$ in commit order. The execution of Serialize both serves the purpose of checking for a write-write conflict (which leads to an abort of x), as well as produces a serialized Trans-PDT $T'x$ that is consecutive to the database state at commit time, hence can be included in TZ and also used to Propagate its updates to the master Write-PDT (i.e. commit).

Example. The example in Figure 15 shows concurrent execution times of three transactions: a, b, and c. At start time t_1, we start out with an initial master Write-PDT $W_{t_1} = \emptyset$. At t_{1a}, the first transaction, a, arrives and takes a snapshot

Algorithm 9 Finish(ok, W_{t_n}, Tx^t, TZ)
Commit(w, tx, tz) = Finish(true,w, tx, tz)
Abort(w, tx, tz) = Finish(false,w, tx, tz)
Transaction x that started at t, tries to commit its Trans-PDT Tx^t into master Write-PDT W_{t_n}, taking into account the sequence of relevant previously committed consecutive PDTs $TZ = (T'z1_{t_1}^{t_0}, \cdots, T'zn_{t_n}^{t_{n-1}})$. If there are conflicts between Tx and any $T'zi \in TZ$, the operation fails and x aborts. Otherwise the final $T'x$ is added to TZ and is propagated to W_{t_n}.

```
1: T'x = Tx; i = 0
2: while (i = i + 1) ≤ n do // iterate over all T'i
3:    T = T'zi_{t_i}^{t_i − 1}
4:    if t < t_i then // overlapping transactions
5:       if ok then
6:          ok = T'x.Serialize(T)
7:       T.refcnt = T.refcnt − 1
8:       if T.refcnt ≡ 0 then // x is last overlap with zi
9:          TZ = TZ − T
10: if ok ≡ false then // conflict: x must abort
11:    return false
12: W_{t_{n+1}} = W_{t_n}.Propagate(T'x)
13: T'x.refcnt = ||runningtransactions||
14: if T'x.refcnt > 0 then
15:    TZ = TZ + T'x
16: return true// x can commit
```

of the write-PDT: Wa =Copy(W_{t_1}) = $\emptyset$. Here, we chose notation t_{1a} to suggest that the database state at that time is the same as at t_1. Transaction a starts out with an empty Trans-PDT, $Ta^{t_{1a}} = \emptyset$. At t_{1b}, the second transaction, b, arrives and gets the same snapshot (with empty write-PDT) $Wb = Wa$, since no commits took place in the meantime. At t_2, transaction b commits, thereby propagating its changes from $Tb_{t_2}^{t_{1b}}$ to the current table image. There were no commits during the lifetime of b, so the most up-to-date table image is still represented by Wb, allowing us to commit by propagating Tb directly to the master Write-PDT: $W_{t_2} = W_{t_1}$.Propagate($Tb_{t_2}^{t_{1b}}$). The resulting new master write-PDT reflects the state as seen by incoming transaction c at t_{2c}: $Wc_{t_{2c}}$ =Copy(W_{t_2}). Again we start with an empty $Tc^{t_{2c}} = \emptyset$ for transaction c. The next thing that happens is the commit of transaction a at t_3. We serialize $Ta_{t_3}^{t_{1a}}$ with respect to $Tb_{t_2}^{t_{1b}}$: $T'a_{t_3}^{t_2} = Ta_{t_3}^{t_{1a}}$.Serialize($Tb_{t_2}^{t_{1b}}$), which reports no conflicts. The resulting $T'a_{t_3}^{t_2}$ is consecutive to W_{t_2}, into which we can now safely commit: $W_{t_3} = W_{t_2}$.Propagate($T'a_{t_3}^{t_2}$). Finally, when transaction c commits, at t_4, we still have $T'a_{t_3}^{t_2}$ around, which is aligned with $Tc^{t_{2c}}$, so we can do $T'c_{t_4}^{t_3} = Tc_{t_4}^{t_{2c}}$.Serialize($T'a_{t_3}^{t_2}$), which can then be propagated to the master Write-PDT: $W_{t_4} = W_{t_3}$.Propagate($T'c_{t_4}^{t_3}$).

In summary, at transaction commit, we check for conflicts against all transactions that committed during the lifetime of the transaction. Each such overlapped commit is characterized by its serialized Trans-PDT; we cache these for those recent transactions that still overlap with a running transaction. By serializing the committing Trans-PDT with these cached PDTs one-by-one, we finally obtain a Trans-PDT that is consecutive to the current database state, and can be Propagate-ed to it (and added to the cache itself). These Serialize operations will also detect any conflicts, in case of which the transaction gets aborted instead. Together, this amounts to optimistic concurrency control.

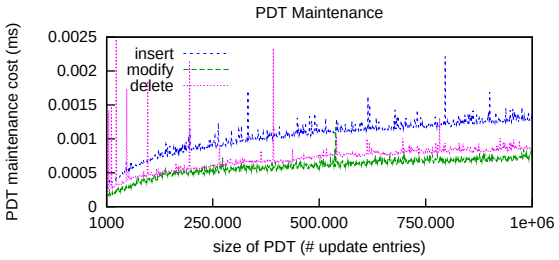

Figure 16: PDT Update Performance Over Time

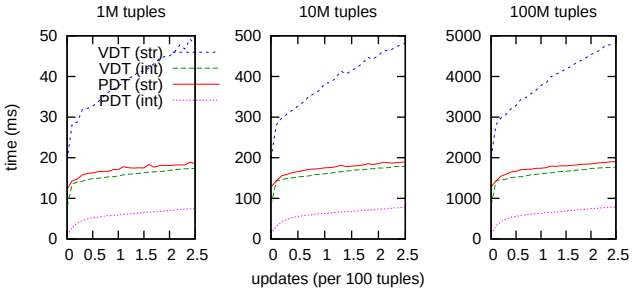

Figure 17: MergeScan: Scaling and Key Type

4. EXPERIMENTAL EVALUATION

To evaluate the benefits of the proposed techniques, we run two sets of benchmarks. First, with micro-benchmarks we demonstrate the performance of inserting, deleting and modifying data of varying size using PDTs. We also measure the performance of MergeScan, using different sort key data types and update rates. Then, we investigate if PDTs succeed in providing uncompromising read performance in large-scale analytical query scenarios, using the queries from the TPC-H benchmark.

Benchmark setup. For our experiments we used two hardware platforms. The workstation system is a 2.40GHz Intel Core2 Quad CPU Q6600 machine with 8GB of RAM and 2 hard disks providing a read speed of 150MB/s. The server system is a 2.8GHz Intel Xeon X5560 machine with 48GB of RAM and 16 SSD drives providing 3GB/s I/O performance. The micro-benchmarks were performed on workstation, were memory-resident and the results are averaged over 10 consecutive runs. The standard deviation over these runs is very small and thus not reported. The TPC-H benchmarks were performed on both workstation and server.

Update Microbenchmarks. The first set of experiments demonstrate the logarithmic behavior of PDTs when they grow due to execution of ever more updates. Figure 16 depicts the time needed to perform inserts, deletes and modifies respectively, to a constantly growing PDT (up to 1 million operations). Clearly, inserts are more expensive than modifies and deletes since the keys must be compared to compute insert SIDs.

MergeScan Microbenchmarks. Figure 17 presents the results of scanning a table of 4 columns and 1 key column (integer or string) with updates managed by PDTs and VDTs. The query used is a simple projection of all 4 columns after a varying number of updates have been applied. In all cases PDT outperforms VDT by at least a factor 3. Furthermore, this experiment demonstrates linear scaling of query times with growing data size for both PDT and VDT.

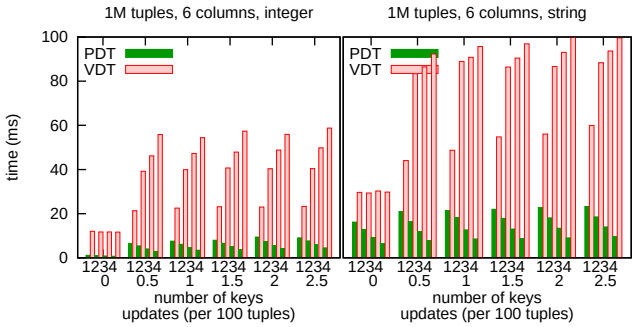

Figure 18: MergeScan: Single- vs Multi-Column

The benefits of PDT are especially visible when the key column contains strings. In that case, as the percentage of updates is increased, value-based merging in VDTs becomes significantly slower due to expensive string comparisons. On the other hand, PDTs do not need to perform value comparisons, thus their cost is lower and does not increase significantly with update ratio.

The next set of experiments investigate the impact of increasing the number of key columns in a table of 6 columns. Here we expect VDTs to suffer when instead of a single-column sort key we have multiple sort columns, as the value-based merge-union and -diff logic becomes significantly more complex with multi-column keys. As in PDTs MergeScans do not need to look at the sort key columns, they are not influenced by this at all. Figure 18 depicts the results for both integer and string type of keys. The x-axis is divided into 2 dimensions: for each update percentage we conduct the experiment with a varying number of keys, from 1 to 4 columns. The query projects the remaining non-key columns. For VDTs, the query time increases significantly when the number of keys to be scanned and compared is increased. For PDTs, query time decreases because fewer columns have to be projected when the number of keys increase, while merge cost stays constant.

Overall, presented micro-benchmarks demonstrate that PDTs are significantly more efficient than VDTs, especially with complex (string, multi-attribute) keys.

TPC-H Benchmarks. Whereas in the micro-benchmarks we only focused on the PDT operations under controlled circumstances, we now shift our attention to overall performance of analytical queries using the full TPC-H query set (22 queries). The focus in these experiments is establishing whether PDTs indeed succeed in allowing column-stores to be updated, without compromising their good read-only query performance. We compare clean queries (no-updates), that is, queries to a clean database that only has been bulk-loaded, to queries to a database that has been updated by the official 2 TPC-H update streams which update (insert and delete) roughly 0.1% of two main tables: lineitem and orders. We test both PDTs and VDTs implemented inside the VectorWise system on both workstation and server systems described above.

The experiments were conducted for scientific evaluation only, and do not conform with the rules imposed on an official TPC-H benchmark implementation. Therefore, we omit any overall score, and explicitly note that these individual query results should *not* be used to derive TPC-H performance metrics.

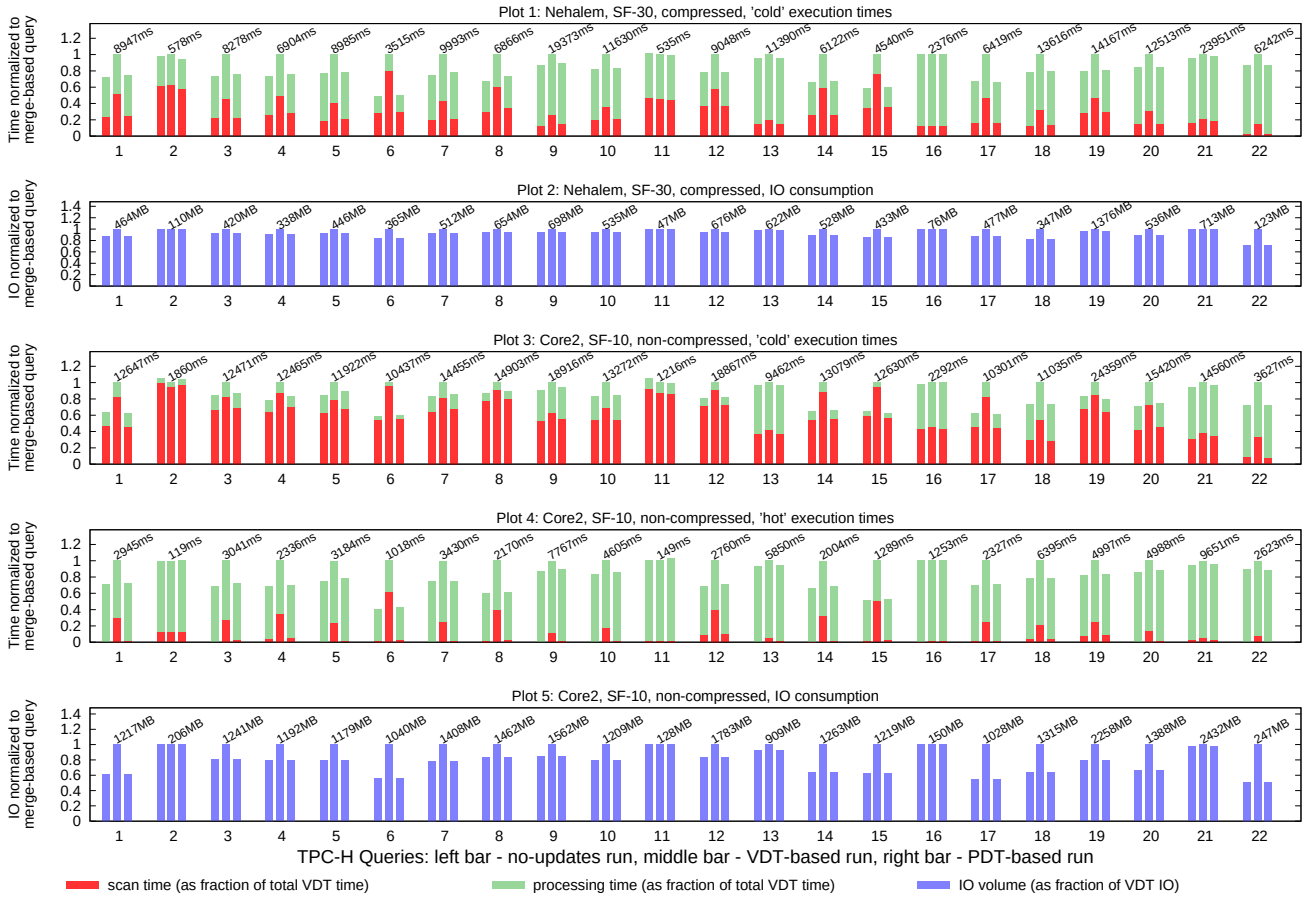

Figure 19: TPC-H: server (compressed SF-30) vs workstation (non-compressed SF-10)

The lineitem table in our setting is ordered on a {l_orderkey, l_linenumber} key, while the orders table is ordered on a {o_orderdate, o_orderkey} combination. Due to this ordered columnar table storage, which is very much like row-wise "index-organized" table storage (a clustered index), certain queries can exploit range predicates; however the update task is non-trivial, as the inserts touch locations scattered throughout the tables.

On the server, we measured the performance using a compressed SF-30 (30GB) dataset, while on workstation we used an uncompressed SF-10 (10GB) dataset. The results are presented in Figure 19 For each of the 3 scenarios a separate bar is plotted for every TPC-H query.[6]

We provide two types of results: *(i)* I/O volume consumed and *(ii)* query performance, separated into data-scanning (including reading from disk, decompression, and applying updates, as applicable), and query processing time. All results are normalized against the VDT runs, with the absolute numbers provided for those.

The top two segments of Figure 19 present the results on the server, using a compressed, disk-resident ("cold") SF-30 dataset. Plot 2 shows that the I/O volume for VDT runs is consistently higher (or equal) than in no-updates and PDT runs, due to mandatory scanning of sort key columns in VDT merging. However, on this platform, the I/O difference is relatively small, due to good compression ratios for the (sorted) key columns. Still, the overhead of value-based merging, visible in Plot 1, can make the "scan" part often significantly higher for the VDT scenario. PDT runs demonstrate a very small "scan" overhead over no-updates, resulting in a negligible impact on the total query time.

The bottom 3 segments of Figure 19 present results on the workstation, using a non-compressed SF-10 dataset, both memory- ("hot") and disk-resident ("cold"). Plot 5 shows that with non-compressed keys, the I/O volume in the VDT case is significantly higher, up to a factor 2. This demonstrates the benefit of PDTs on tables with keys that have large physical volume (strings; hard to compress; multi-column). This increase in I/O is directly reflected in the performance of the "cold" runs (Plot 3). Plot 4 demonstrates a scenario where lack of compression (which constitutes a significant part of the "scan" cost in Plot 1), combined with memory-resident data, makes accessing data a "zero-cost" operation for the no-updates runs. With the data-access cost eliminated, Plot 4 demonstrates the absolute CPU cost of applying the updates to the data stream. Here, the VDT approach can have a very significant overhead, consuming up to half of the total processing time (e.g. in query 6). The CPU cost for PDT merging is significantly lower, and negligible in most cases.

In all, we can see that value-based merging can be significantly slower (>20%) than positional merging as introduced by PDTs, and PDTs consistently achieve query times very close to querying a clean database.

[6] Queries 2, 11 and 16 do not touch updated tables, hence the results for them do not differ between runs.

5. RELATED WORK

Differential techniques [20] historically lost out to update-in-place strategies as the method of choice to implement database updates handling. Gray pointed out ("UPDATE IN PLACE: A poison apple?") that differential techniques, which do not overwrite data, are more fail-safe, but recognized that magnetic disk technology had seduced system builders into update-in-place by allowing fast partial file writes [12]. Update-in-place, also implies random disk writes; where hardware improvement has lagged compared to sequential performance. While our paper does not touch the subject of using differential techniques for row-based systems, it recognizes differential techniques as the most salient way to go for column-stores, where update-in-place is additionally hindered by having to perform I/O for each column, aggravated by compressed and replicated storage strategies.

The idea to use differential techniques in column-stores is not new (the idea to do so positionally, is) and was suggested early on [8]. The Fractured Mirrors approach [19] that combined columnar with row storage, also argued for it. Both papers did not investigate this in detail. The open-source columnar MonetDB system uses a differential update mechanism, outlined in [2]. C-Store proposed to handle updates in a separate write-store (WS), with queries being run against an immutable read-store (RS), and changes merged in from the WS on-the-fly. We see our PDT proposal as particularly suited for columnar stores, because our positional approach allows queries that do not use all key columns, to avoid reading them – a crucial advantage in a column-store, and is also computationally faster.

As for previously proposed differential data structures, the Log-Structured Merge-Tree (LSMT) [16] consists of multiple levels of trees, and reduces index maintenance costs in insert/delete-heavy query workloads. Similarly, [13] proposes multi-level indexing. The goal of improving insertions using not stacked, but *partitioned* B-trees was explored in [9]. A possible reason why multi-tree systems so far have not been very popular, is that lookup requires separate I/Os for each tree. The assumption in our PDT proposal, and similarly in the value-based "VDT" (our terminology) approaches used e.g. in MonetDB, is that only the lowest layer data structure (columnar storage) is disk-resident and requires I/O. The availability of 64-bits systems with large RAM thus plays in its advantage.

Finally, we have shown how PDTs can be used as an alternative way to provide ACID transactions; an area of database functionality where ARIES [15] is currently the most prominent approach. There are commonalities between ARIES and our proposal, like the use of a WAL and checkpointing, but the approaches differ. ARIES uses immediate updates, creating dirty pages immediately at commit, rather than buffering differences. Instead of tuple-based locking and serializability used in ARIES-based systems, column-stores tend to opt for snapshot isolation, typically with optimistic concurrency control. PDTs fit this approach, offering lock-free query evaluation, using three layers of PDTs (Trans,Write,Read). While snapshot isolation allows anomalies [1], user acceptance for it is high, and recently it was shown that snapshot-based systems can provide full serializability [4] with only a limited amount of bookkeeping. In all, we think that given hardware trends, and the specific needs of column-stores, PDT-based transaction management is a new and attractive alternative.

6. CONCLUSIONS AND FUTURE WORK

We have introduced the Positional Delta Tree (PDT), a new differential structure that keeps track of updates to columnar, compressed (read-optimized) data storage. Rather than organizing the modifications based on a table key (i.e., value-based), the PDT keeps differences organized by *position*. The PDT requires reading less columns from disk than previous value-based differential methods and is also computationally more efficient. We have described algorithms for ACID transaction management using three PDT layers (Read,Write,Trans); allowing efficient lock-free query execution. PDTs can thus be seen as a new and attractive approach to transaction management in column stores, that does not compromise its high analytical read performance.

Future work is underway on keeping join indices up-to-date with PDTs, and using PDTs in information retrieval systems. Other topics include keeping PDTs in Flash memory as well as using PDTs for row-stores.

7. REFERENCES

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD*, 1995.

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

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

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

[5] G. Canahuate, M. Gibas, and H. Ferhatosmanoglu. Update conscious bitmap indices. In *Proc. SSDBM*, 2007.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proc. SIGMOD*, 2001.

[7] X. Chen, P. O'Neil, and E. O'Neil. Adjoined dimension column clustering to improve data warehouse query performance. In *Proc. ICDE*, 2008.

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

[9] G. Graefe. Sorting and indexing with partitioned b-trees. In *Proc. CIDR*, 2003.

[10] G. Graefe. Efficient Columnar Storage in B-trees. *SIGMOD Record*, 36(1), 2007.

[11] G. Graefe. Fast loads and fast queries. In *DaWaK*, 2009.

[12] J. Gray. The transaction concept: Virtues and limitations. In *Proc. VLDB*, pages 144–154, 1981.

[13] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proc. VLDB*, 1997.

[14] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proc. VLDB*, 1998.

[15] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *TODS*, 17(1), 1992.

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

[17] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras. Multi-dimensional clustering: a new data layout scheme in db2. In *Proc. SIGMOD*, 2003.

[18] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proc. ICDE*, 2001.

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

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

[21] D. Slezak and V. Eastwood. Data warehouse technology by InfoBright. In *Proc. SIGMOD*, 2009.

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

[23] S. Tatham. Counted B-trees. In *www.chiark.greenend.org.uk/sgtatham/algorithms/cbtree.html*, 2001.