Write-Optimized Indexing with Partitioned B-Trees

Christian Riegger
Data Management Lab
Reutlingen University,
Germany
christian.riegger@
reutlingen-university.de

Tobias Vinçon
Data Management Lab
Reutlingen University,
Germany
DXC Technology
Böblingen, Germany
tobias.vincon@dxc.com

Ilia Petrov
Data Management Lab
Reutlingen University,
Germany
ilia.petrov@
reutlingen-university.de

ABSTRACT

Database management systems (DBMS) are critical performance component in large scale applications under modern update-intensive workloads. Additional access paths accelerate look-up performance in DBMS for frequently queried attributes, but the required maintenance slows down update performance. The ubiquitous B$^+$-Tree is a commonly used key-indexed access path that is able to support many required functionalities with logarithmic access time to requested records. Modern processing and storage technologies and their characteristics require reconsideration of matured indexing approaches for today's workloads. Partitioned B-Trees (PBT) leverage characteristics of modern hardware technologies and complex memory hierarchies as well as high update rates and changes in workloads by maintaining partitions within one single B$^+$-Tree. This paper includes an experimental evaluation of PBTs optimized write pattern and performance improvements. With PBT transactional throughput under TPC-C increases 30%; PBT results in beneficial sequential write patterns even in presence of updates and maintenance operations.

CCS Concepts

•Information systems → Data access methods; B-trees;

1. INTRODUCTION

Indexes describe an additional access path to data located in base tables and can speed up look-up performance for specific data maintained in key columns, but they slow down insert and update performance in transactional workloads. The index structure of a B$^+$-Tree [5] became ubiquitous in database management systems [3]. Unlike other index structures, B$^+$-Trees allow to accessing data in sorted order with logarithmic complexity. Point queries and scans are optimally supported as well as uniqueness constraints and multi-column indexing. Records can be created, looked-up, updated and deleted dynamically and transaction safe with relatively high concurrency.

B$^+$-Trees are a dynamic and balanced tree structure. Its logical data model fits well to a physical representation on secondary storage. The size of an index node can correspond to the logical size of a page on persistent storage media. Every index node does have an unique identifier. Internal nodes (including the root) contain sorted separator keys, which direct a search to leaf nodes in a traversal operation. Several data records are sorted stored in leaf nodes. All leaf nodes are in the same level of the B$^+$-Tree and linked with their siblings via page pointers to fully support scans. Typically, a data record consists of indexed columns, a pointer with the record-id of the indexed data record and some header data. The height of a B$^+$-Tree depends on several factors, but with high fan-out is typically low (3 to 6 levels). While insertion and deletion, maintenance operations can occur and nodes may get split or merged and records may get moved. Modification cause an additional maintenance overhead and I/O operations. All operations can take benefit from caching pages in main memory.

Typically, not all required data in a database fits in main memory. Upper levels in a B$^+$-Tree near the root will be cached in most cases. A few nodes in lower levels can be cached due to a high fan-out and limited main memory. If needed, these nodes are fetched from secondary storage, but therefore cached nodes have to be evicted and potential changes have to be written. The result is an unfavorable I/O pattern whereas a sequential one would be much more favorable. Furthermore, the updated data volume is much smaller compared to the size of the written page. Write amplification is the ratio between the logical and the physical write volumes. Workloads with random updates in the index usually result in high write amplification. Moreover, index maintenance operations result in random write I/O.

As we can see, B$^+$-Trees are very efficient for range and point look-ups, but they have shortcomings in index updates with respect to performance, write I/O pattern and write amplification. In Section 1.1 we implement and extend an adaption to B$^+$-Trees, the Partitioned B-Tree (PBT) [1], to handle the above shortcomings. We compare PBT to further approaches in Section 2 and give some implementation details in Section 3. We validated our estimations in a prototype, which is based on Apache Derby, on a TPC-C like benchmark and present our results in Section 4.

This paper shows how rethinking and adapting established algorithms with few modifications can lead to a desired and favorable behavior and notable performance improvements.

iiWAS '17, December 4–6, 2017, Salzburg, Austria
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-5299-4/17/12...$15.00
DOI: https://doi.org/10.1145/3151759.3151814

1.1 Approach: Partitioned B-Trees

1.1.1 Data Structure Overview

We implement and extend the Partitioned B-Tree (PBT) [1] as an adaption of the traditional B^+-Tree. PBT makes use of most B^+-Tree algorithms with few modifications. The essential difference is the introduction of an artificial leading search key column – the partition number. An index record consists of a partition number, its search key columns and a record-id. Every partition number uniquely identifies a single partition. This idea enables the PBT to maintain partitions within one single tree structure and reuse existing intricate implementation algorithmic optimizations, buffering, etc. Partition numbers are transparent for higher database layers and each PBT maintains partitions independent from other PBTs. Partitions appear and vanish as simple as inserting or deleting records and can be reorganized and optimized on-line in merge steps, depending on the workloads. Partitioning data within a single B^+-Tree enables several possible applications (outlined in [1]). In this paper, we focus on write optimization and a better use of the memory hierarchy in a TPC-C like transactional workload.

Traditional B^+-Trees tend to have the disadvantage of a high write amplification (see Table 2). PBTs write any modification of index records exactly once on eviction of the respective partition, except for later reorganization or garbage collection operations. This is realized by forcing sequential writes of all leaf pages of a whole partition. The operation is illustrated in Figure 1. Leaf nodes of the latest partition are stored in a separate area of the database buffer – the PBT-Buffer. The PBT-Buffer is shared for all PBT indexes in database. Records can be inserted, updated and deleted only in the latest partition. The PBT-Buffer is designed to get active partitions the chance to grow. Once the PBT-Buffer gets full, it selects a victim partition and writes it out sequentially to secondary storage. Once a partition is written out, it is immutable. *First*, once a partition is evicted a new partition is automatically created to host future index entries. *Second*, upon eviction a bloom filter is created covering all index entries within the partition. The bloom filter is written out together with the partition data and remains unchanged as partitions are immutable. The purpose of bloom filters will be explained later in this paper. *Last*, all leaf pages are written sequentially to secondary storage.

The index entries of a PBT-Buffer resident partition are also part of the higher levels of the PBT index. Hence, they are traversed as part of any look-up operation.

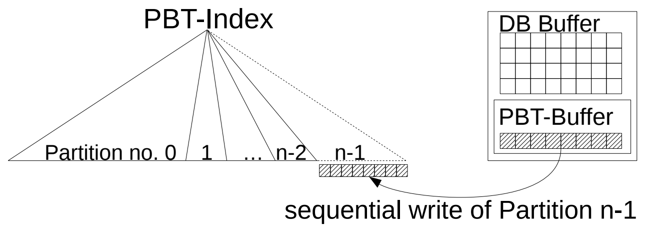


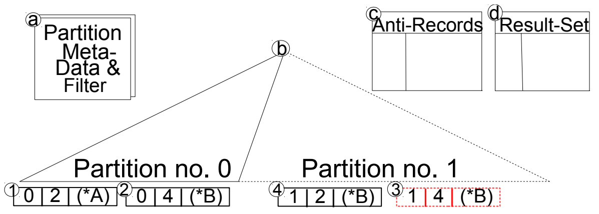
Figure 1: Sequential write of a Partition

It is fair to say that all modifications are cached and leaf pages have a nearly optimal write amplification. Inner nodes in the PBT are passed to the regular cache logic. In fact, that modifications are applied to its latest partition, as well as look-ups will be performed mostly on its latest partition, inner nodes, which direct to leaf nodes in this partition, are cached as well and evictions are very rare for most replacement policies.

Updates of already written data in previous partitions require an out-of-place invalidation. This can be done by inserting a Replacement-Record in the new partition. Deleting a record in an old partition requires the insertion of an Anti-Record in the new partition [1]. Therefore, Replacement-Records can be viewed as a combination of an Anti-Record and a new inserted Index-Record. In the search algorithm, Anti-Records, matching to the search criteria, can be collected and invalidated Index-Records can be sorted out from the result set, because the PBT starts probing the latest partition and ends up in the oldest partition, except if the search algorithm can break up earlier (a detailed example is provided later in this paper).

Implications of Look-Up Overheads. Probing every partition leads to a higher read amplification by a factor of the partitions count in comparison to a traditional B^+-Tree. There is a certain impact on performance due to following facts. *First*, modern storage media allow fast read operations. Asymmetric storage media, like SSDs, can perform several requests in parallel with low latencies. Write operations are much slower. *Second*, newly inserted data is well cached in the PBT-Buffer and common database buffer. The latest partition can be traversed and looked-up without any I/O operation on secondary storage media. A look-up can break up earlier, if a query requests a matching record without any sort order. *Third*, bloom filters for every closed partition and partition key ranges can avoid traversing partitions without relevant records. This behavior saves read I/O operations. Partition key ranges are useful for any kind of query. Partitions can be skipped, if the requested record key is not in the partition key range. If the requested record is in the partition key range, the bloom filter of a partition has to be checked. Bloom filters are created, after the partition was closed for modifications. So, a well sized simple bit vector guarantees a false positive rate of less than 1%. Bloom filters are useful for uniqueness constraint checks and point queries with a full search key. For range queries and partial key searches further filter methods have to be considered. *Last*, the number of partitions can be reduced. It is possible to merge partitions in a merge sort, which would be processing and memory intensive, but mechanics can be implemented to pass low impact on performance of current workload. The result of a merge sort is a new partition, that can be written sequentially. In the merged partition, Anti-Records may neutralize Index-Records, both can be garbage collected. Afterwards, two or more merged partitions can be removed from the whole tree structure in a single step and space on secondary storage can be reclaimed. A further approach is Adaptive Merging [4], whereby often queried records can be cached in a new partition and older partitions has not to be queried for this records. Both features are not considered in current implementation.

Example of a PBT. In Figure 2, important data structures as well as the result for different look-up transactions in a list are depicted for a specific example. The important data structures are *(a) Partition Meta-Data*, which is maintained for every partition in a PBT, *(b) the PBT index structure*, which was build on column **x** of a base table, *(c) a list of Anti-Records*, which is maintained for every scan operation and *(d) the Result Set* of a scan. Two partitions exist in the example. *Partition 0* is filled with two Index-Records – *1* has a key of **2** and references to tuple **A** in base table and *2* has a key of **4** at eviction time of this partition and references to tuple **B**. In *Partition 1*, an update was performed on tuple **B** – an *Anti-Record (3)*, which invalidates *Index-Record (2)* in *Partition 0*, and a new *Index-Record (4)* with the new key value **2** were inserted – modifications were performed write optimized out-of-place.



Transaction	traverse Partition 1 in (b)	traverse Partition 0 in (b)	Result
T1: COUNT(x=2)	add Index-Record (**4**→**d**)	add Index-Record (**1**→**d**)	**2**
T2: COUNT(x=4)	remember Anti-Record (**3**→**c**)	skip Index-Record (**2**) by (**c**)	**0**
T3: COUNT(x<5)	add Index-Record (**4**→**d**), remember Anti-Record (**3**→**c**)	add Index-Record (**1**→**d**), skip Index-Record (**2**) by (**c**)	**2**
T4: COUNT(x=5)	skip due to partition key range located in (**a**)	skip due to partition key range located in (**a**)	**0**
T5: GET1(x=2)	add Index-Record (**4**→**d**)	not processed	***B**
T6: GET1(x=3)	no Index-Record found in (**b**)	skip due to bloom filter in (**a**)	**-**

Figure 2: Look-up in a PBT-Index

Transaction 1 counts all records with a value of **2**. This key is in partition key ranges in *(a)* of *Partition 1*, so this partition has to be traversed in-memory. *Index-Record (4)* is found in leaf nodes in PBT-Buffer and will be added to *(d) Result Set.* No further record will be found in *Partition 1*. *Partition 0* will be traversed next due to positive criteria in partition key ranges and bloom filter in *(a)*. *Index-Record (1)* will be added to *(d) Result Set* and the scan ends. The result of the transaction is **2**. *Transaction 2* counts all records with a value of **4**. The search algorithm will perform similar to *Transaction 1*, except it will find *Anti-Record (3)* in *Partition 1* and add it to *(c) Anti-Records* list. When traversing and look-up *Index-Record (2)* in *Partition 0*, the search algorithm will recognize the *Anti-Record (3)* in *(c)* and ignores the record. The result is **0** records are matching to the search criteria. *Transaction 3* is a range scan, that combines the behavior of *Transaction 1* and *2*, except that bloom filters cannot be used. *Transaction 4* does not traverse any partition, because its search criteria **x=5** is not in the partition key ranges, which are located in *(a) Partition Meta-Data*, and are checked first. The index can return a result of **0** without any read I/O on secondary storage. *Transaction 5* looks-up one tuple with a key of **2**. After checking *(a) Partition Meta-Data*, *Partition 1* will be traversed in-memory. *Index-Record (4)* will be found in a leaf node in PBT-Buffer, without performing any read I/O on secondary storage. This record can be returned and the tuple **B** can be fetched from base table. *Transaction 6* requests one tuple with a key of **3**. This key will be in partition key ranges in *(a) Partition Meta-Data* of both partitions. *Partition 1* has to be traversed in-memory, whereby no record will be found. *Partition 0* has not to be traversed, because its bloom filter will return a negative result, and read I/O on secondary storage can be saved.

2. RELATED WORK

LSM- [6] and bLSM-Trees [7] are optimized for high update rates as well as bulk loads and achieve sequential write patterns to secondary storage media by presorting inserted and updated data in the *C0* component, which is located in main memory, and merge operations between all components. **PBT** achieve a similar write behavior on secondary storage media by sequential writes on eviction of partitions in the PBT-Buffer. Insert operations can be performed in the latest partition without any I/O operation on secondary storage at any point in time. No B^+-Tree-like update inplace operation is performed. If the latest partition gets immutable, a new one was created before, which can handle ongoing modifications. Inner nodes in **PBT**, which direct to leaf pages of its latest partition, are cached in the common database buffer mostly, for which reason no considerable I/O operations occur while index traversal. The latest partition in **PBT** is comparable to *C0* component in LSM trees. Furthermore, it is possible for **PBT**s to create a separate partition to handle bulk loads. This is beneficial in several ways. Data in further partitions is not affected by a bulk load. There is low influence on performance of other transactions. Furthermore, if the bulk load transaction fails, the whole partition can be easily removed from a **PBT**. The partition can be written sequentially after commit of the bulk load transaction.

Write Optimized B-Trees [2] optimally support append-only sequential writes, without losing the ability to update data in-place. For disks, this is beneficial for writes, but reads become more randomly and pre-fetching for scans gets expensive. **PBT** looks-up partition after partition. Pages in a partition are allocated nearby. Pre-fetching can be performed without arm-movement. In case of SSDs, a further problem occur. Pages are not protected of further modification and may become invalidated. As a result, pages in blocks become invalid and to clean-up space, SSDs have to copy valid pages in a new block to delete invalidated pages finally. This behavior increases the write amplification of a SSD. If pages in a **PBT** get evicted, they cannot become invalid, except for merges, reorganization and garbage collection. This operations affect all pages in a partition and blocks on SSD can be erased without copying valid pages.

3. IMPLEMENTATION DETAILS

We implemented our prototype in Apache Derby release 10.10. Derby is an open source relational database implementation, based on Java, JDBC and SQL standards. It uses B-Tree Secondary Index (B2I) implementation based on Lehman and Yao's B^+-Tree [5] index structure. The replacement policy of the cache manager is a clock algorithm.

A prototypical implementation of Partitioned B-Trees affects the modules *B2I* and *CacheManager* mainly. Further database modules and functionalities are not affected and there is no need for adaption. In this section, we outline the changes to the standard algorithm.

3.1 Partition Management

Before we start demonstrating the applied changes to index operations, we need to understand partition management and selection. Every single B$^+$-Tree has typical meta data, additional information and one or more Partitions. Partitions are stored in a linked list in main memory. A Partition object consists of the partition number, an indicator, if it is mutable, timestamps for creation and closing time, minimum and maximum record stored in partition (partition key ranges) and a count of records. A Partition maybe contains a Bloom Filter, which is located in main memory, but can be allocated on any storage media in memory hierarchy, e.g. on fast non-volatile memories.

PBT loops over partitions in reverse order in several look-up operations, until the algorithm break. Remember, maybe not every partition contains a requested key. To filter out unnecessary partitions, the search key ranges (key_{start} and key_{stop}) of the look-up is required to determine the next relevant partition. For purpose of using bloom filters, it has to be determined, if an exact full key is requested. Bloom filters cannot answer partial keys or key ranges. In both cases, a partition key range check can be performed, to filter out an unnecessary partition.

3.2 Index Operation Algorithms

PBT Insert Algorithm. Inserting a new record in a PBT will be performed in the newest partition, located in PBT-Buffer in main memory. First the index has to be opened for update. Then the insert operation can be performed. An index record consists of a key, which is an array of columns, and a row location pointer. In a PBT, index updates will only be done in the newest active partition. But for uniqueness constraints, older partitions have to be checked, too. If there is no uniqueness constraint, the check of older partitions can be skipped. The key is not in every partition available. If the new key is not within the range of the partition or the key is not in the bloom filter, which will be created while the partition was closed and evicted to secondary storage, there must be done no look-up for this partition. The bloom filter for the newest partition has not been created yet, but the look-up will be performed in-memory. The first column of the key will be transformed to a partitioned key. It is a sequence of bytes, the first two bytes consist of the partition number value and all other bytes consist of the original key. After all, the index-record will be added to the B$^+$-Tree in a leaf page on its regular position inside the partition. If the page is full, on which the index-record should be inserted, a page split has to be performed first. The index-record type of a regular insert operation is an Index-Record.

PBT Look-Up Algorithm. Look-Up operations in Apache Derby consist of two algorithms that return one record per invocation – *next* and *fetch*. First the PBT has to be opened for read access. The Scan has to be initialized, whereby a scan partition is selected and the scan position is set to one slot before the first matching key. *Next* will be invoked, which set the position to the first matching slot on a page. There is a fast exit implemented, if a point query is performed and the key was invalidated in a newer partition. After the position was set, it is necessary to determine, if the key at the slot is less the upper key range of the scan – then the next partition has to be checked – and if the record was marked as deleted (Ghost-Record in regular B2I implementation). If the record is valid, the record type (Index-Record or Anti-Record) has to be determined. Anti-Records will be added to a list of Anti-Records, located in the current scan partition ($anti_{part}$) and the next slot can be checked. If the record type is an Index-Record and the record is not located in the Scan-Anti-Record-List $anti_{scan}$, the position is set to the next valid record and can be fetched. *Next* will be invoked again, if a further record is required. If there is no further matching key in a partition, the next partition is selected, whereby the $anti_{part}$ list will be appended to $anti_{scan}$ and the search key range will be updated. This algorithm can be invoked, until all partitions are processed and no further key can be found. When fetching a record and returning it to further database layers, the partition number is made transparent.

In a partition, the sort order is ascending or descending. For fetches, that affect more than one partition, the sort order maybe is not as expected from regular B$^+$-Trees. In this case the predicate "ORDER BY *indexed column* ASC/DESC" has to be used in the query. So the query planner is able to use the procedure, that merges records in the desired sort order. Additional optimizations can be performed on aggregates.

PBT Delete Algorithm. In general, deletions are not performed physically in Derby, but logically. Therefore a deletion marker is set at a matching Index-Record and the record becomes a Ghost-Record. This behavior reduces index maintenance operations. Slots of Ghost-Records can be recycled, by reusing the slot for new insertions, or can be garbage collected later. A PBT can make use of this behavior, in case of deleting an Index-Record that is located in newest partition. A look-up operation is performed and the scan position is set by algorithm *next*. Then the status flag can be set by the *delete* algorithm. If the Index-Record is not located in newest partition, an Anti-Record has to be inserted. Therefore, a new Index-Record is inserted in the newest partition and its status flag is set to *invalidate*, whereby it becomes an Anti-Record.

PBT Update Algorithm. Updates in Apache Derby are a combination of the *delete* and *insert* algorithm. For PBT the algorithms *next*, *delete* and *insert* are performed. Modifications will be done only in newest partition.

3.3 Evict Partition

Eviction from PBT-Buffer of a partition asynchronously occurs in a server transaction. This happens, if the reserved space for leaf pages in the PBT-Buffer is reached. Which partition will be written to disk depends on the cache logic. Frequently updated partitions should grow, but an eviction of a partition should also clean-up as much space as possible in the cache. Currently, the partition with most used buffers will be closed. A new partition is created. The older one becomes immutable. Any record is read from the pages in the PBT-Buffer and added to a newly created bloom filter. A read lock is required, but that does not affect index updates, because changes will be done in the new partition. After all, the pages are written to secondary storage in a sequential manner. An enhancement could be to merge leaf pages. In fact of, the partition won't be changed at any time, it is not necessary to leave space on any leaf page. For a leaf page fill ratio of 50% to 75%, the pages which have to be written can be also reduced up to the half. Freed pages can be returned to free pages of the new partition.

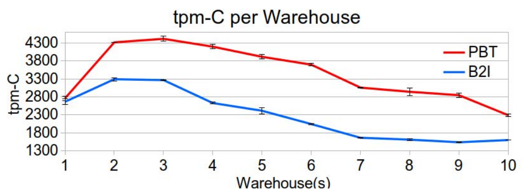


Figure 3: tpmc per Warehouse PBT vs. B2I

WH	PBT AVG	PBT SDEV	B2I AVG	B2I SDEV	Perf. impr.
1	2760	± 2.2%	2663	± 3.1%	3.6%
2	4320	± 0.1%	**3292**	± 1.3%	31.2%
3	**4420**	± 1.6%	3269	± 0.3%	35.2%
4	4200	± 1.5%	2628	± 0.9%	59.8%
5	3917	± 1.6%	2412	± 3.5%	62.4%
6	3700	± 1.0%	2041	± 1.0%	81.2%
7	3059	± 0.5%	1659	± 0.2%	84.4%
8	2942	± 3.4%	1607	± 1.2%	83.0%
9	2847	± 2.0%	1534	± 1.0%	85.7%
10	2287	± 1.9%	1596	± 0.1%	43.3%

Performance Improvement at maximum occupancy factor: 30.5%

Table 1: TPC-C Performance Evaluation

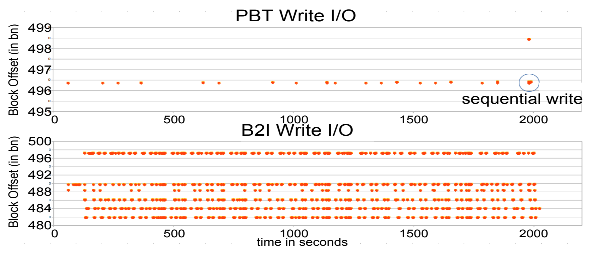


Figure 4: Write I/O Pattern – PBT vs. B2I

Warehouses	1	2	3	4	5	6
PBT						
Total Written GB	1.16	1.87	1.87	1.85	1.80	1.63
Disk Write I/O *1k	19.25	30.9	33.4	33.9	32.7	29.1
File Write I/O *1k	328.7	534.3	533.1	532.0	514.3	466.7
AVG I/O Size	15.75	15.86	14.66	14.30	14.41	14.68
B2I						
Total Written GB	1.49	2.87	3.27	2.81	2.91	2.81
Disk Write I/O *1k	58.1	250.4	356.0	341.2	321.2	352.7
File Write I/O *1k	435.9	994.9	1227.9	1093.8	1131.1	1104.4
AVG I/O Size	6.70	3.00	2.41	2.31	2.38	2.09

Table 2: TPC-C Write I/O Statistics (all indexes)

4. PERFORMANCE EVALUATION

We evaluated the performance of PBT in a TPC-C like benchmark (Java TPC-C) with few modifications due to limitations of Apache Derby 10.10. A connection pool was implemented with up to 20 JDBC connections and a special connection for concurrent delivery transactions was added. Warehouses are the scalability factor of this workload. The database size as well as the concurrent transactions increase with every warehouse.

The experimental set-up with standard TPC-C schema and workload was configured with following parameters: (i) Warehouse Count: 1-10, (ii) Threads: 10 per warehouse, (iii) Connection Pool: up to 20 + 1, (iv) Ramp-Up Time: 30 minutes, (v) Duration: 1 hour. The server configuration is as follows: (i) Operating System: Windows 7 Professional (64 Bit), (ii) CPU: Intel Core i5 4670 Dual Core (3.40 GHz), (iii) RAM: 4GB, (iv) HDD: Seagate ST500DM002. Apache Derby was configured with following parameters: (i) Pagecachesize: 400 MB, (ii) DeadlockTimeout: 10 seconds, (iii) Lock-WaitTimeout: 30 seconds, (iv) CheckpointIntevall: at 2 GB Log Size, (v) LogswitchIntervall: at 500 MB Log Size.

As depicted in Figure 3 and Table 1, PBTs have a 30% better performance at maximum occupancy rate at 3 warehouses in comparison to B2I at 2 warehouses. Furthermore, PBTs have a relative constant performance between 2 to 5 warehouses in relation to B2I, where the performance decrease in a high factor. This results indicate that PBTs have a better scalability and elasticity. Better performance can be explained by well cached new Index-Records and a sequential write I/O pattern.

Evidences for write optimization are depicted in Figure 4 (for a specific update intensive index) and Table 2 (includes all indexes in TPC-C schema and was accumulated over the whole test duration). PBTs write only file extends and some modified inner nodes to secondary storage until time 2000sec, where a sequential write of a partition eviction happens (see Figure 4). Write I/O pattern of B2I is random over all index nodes. The average write I/O size of PBT is constant between 14 and 16 sequentially written pages, B2I has a relatively good rate at 1 warehouse, because evictions are rare for this database- / buffer cache-size ratio – checkpoints result in most write I/O. For bigger sizes, the average sequentially written pages are less than 3. Total written data is for B2I much higher than for PBT, in contempt of less performance, what is an indicator for a high write amplification in B2I.

5. SUMMARY AND CONCLUSION

Directing data modifications in an index to main memory, e.g. in a partition in PBT-Buffer, enables PBT to perform near optimal write I/O patterns for update intensive transactional workloads, like TPC-C. The sequential write I/O pattern and a low write amplification are beneficial for performance and durability of secondary storage media technologies. Recently inserted Index-Records are well cached in main memory and enable noteworthy look-up performance for this workload.

6. ACKNOWLEDGMENTS

The authors wish to thank Goetz Graefe for the support and the valuable comments. This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

7. REFERENCES

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

[2] G. Graefe. Write-optimized b-trees. pages 672–683. VLDB Endowment, 2004.

[3] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[4] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *In ICDE, Workshops*, 2010.

[5] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *TODS*, 6(4), 1981.

[6] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[7] R. Sears and R. Ramakrishnan. blsm: A general purpose log structured merge tree. In *In Proc. SIGMOD '12*.