Experimental Index Evaluation for Partial Indexes in Horizontally Partitioned In-Memory Databases

Marcel Weisgut
Hasso Plattner Institute, University of Potsdam, Germany
marcel.weisgut@student.hpi.de

ABSTRACT

A horizontally partitioned storage layout for column-oriented relational in-memory databases is a popular design choice. One of the reasons for horizontal table partitioning is the increased degree of flexibility in data placement, multi-processing, and physical design decisions. For example, auxiliary data structures such as indexes or filters can be created partition-wise with varying implementations.

However, creating indexes per partition can result in a significant computational overhead for index lookup operations. In this work, we present a partial indexing idea to overcome this overhead. Our proposed approach creates indexes table-wise, but only tuples of frequently accessed partitions are indexed. Our research is still in progress. For the realization of our partial index, the maintenance efficiency of the underlying index structure is particularly relevant. Thus, in this work, we evaluate different index implementations in their lookup speed, maintenance cost, and memory consumption to identify suitable implementations to realize partial indexes. The hash maps *Robin Hood (RH) Flat Map* and *Tessil's (TSL) Sparse Map* achieve overall the best evaluation results. Whereas the former is comparatively faster, the latter has a lower memory footprint.

1. INTRODUCTION

Horizontal partitioning allows to create indexes partition-wise with varying implementations [15, 16]. However, when those indexes are utilized to find qualifying tuples for a given search condition, one lookup operation for each partition's index of the corresponding table has to be performed.

Table 1 shows two exemplary joins that we executed with our research database Hyrise[1]. Tables in Hyrise are horizontally partitioned into partitions of 65 535 tuples. The depicted joins utilize indexes which are created partition-wise. Both joins have the same predicate, the same indexed base table as left input, and a filtered table as the right input. The right, non-indexed table is referred to as *probe table*.

In query 19, the number of tuples in the probe table is 2 771 higher than in query 17. Although this number is small compared to the number of tuples in the indexed table of almost 60 million, it results in a crucial additional effort for the index probing compared to query 17. With the aforementioned partition size, the indexed table consists of 916 partitions. Since indexes are created partition-wise, one equality lookup for each of the 916 created indexes must be executed for each tuple of the probe table to retrieve matching tuples of the indexed table. In total, this results in 1 819 176 equality lookups in query 17 and 4 357 412 in query 19.

[1]Source code at `https://github.com/hyrise/hyrise`

32^{nd} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), September 01-03, 2021, Munich, Germany.

TPC-H Query ID	Join predicate	Number of tuples	
		Indexed table	Probe table
17	l_partkey = p_partkey	∼60 M	1 986
19	l_partkey = p_partkey	∼60 M	4 757

Table 1: Example index (semi) joins generated in Hyrise with the TPC-H queries 17 and 19 using scale factor 10.

Consequently, the index joins' performance depends on the partition count of the indexed data table and scales inversely proportional with it: As the number of partitions increases, so does the number of indexes and the number of lookup operations required.

Our ongoing research focuses on the design and exemplary implementation of a partial indexing strategy to overcome this computational index probing overhead. The contributions of this work are as follows:

- **Partial Indexes.** We introduce an idea of partial indexes to reduce the shown computational index probing overhead. Our partial indexes are to be created table-wise instead of partition-wise, but they store only index entries for frequently accessed partitions.
- **Index Benchmark Framework.** We developed an open-source `C++` benchmark framework, which allows researchers to measure the performance of read and write operations of secondary indexes. The evaluation aspects are the latency of insert, delete, lookup and bulk operations and the index memory consumption.
- **Evaluation of In-Memory Secondary Indexes.** We present an experimental performance evaluation of different index implementations, including trees, radix trees, and hash maps. We evaluated their read and write efficiency and identified index candidates that suite specifically well for our partial index.

2. PARTIAL INDEXES

An advantage of partition indexes is the high flexibility: For each partition, it can be decided individually whether an index is to be created. This makes it possible to create indexes only for partitions whose data is accessed frequently.

Such flexibility is not provided with a traditional full table index since it must always be updated when a tuple is inserted into or deleted from the indexed table, or the search-key value of an already indexed tuple is updated. Thus, partition indexes are more efficient in terms of memory consumption as they can selectively index data of a table whereas a full table index contains index entries for all tuples of a table.

Our proposed partial index is a secondary index that breaks the full table index's requirement of indexing a table entirely and can index one or more partitions. Thus, an arbitrary subset of partitions can be indexed. In contrast to tuple-wise indexing of traditional indexes, our partial index's maintenance operations are executed partition-wise. Consequently, index entries are inserted into or deleted from the partial index for the entire data of a given partition. Inplace updates are not considered in this work since our partial index is specifically designed for insert-only in-memory DBMSs that use multiversion concurrency control.

The Need for Maintenance Efficiency. Our partial index only indexes partitions whose data is accessed frequently to reduce the memory footprint. As the workload of a DBMS can change over time [24], so does the frequency with which data is accessed. With a changing set of partitions whose data is accessed frequently, existing partial indexes must continuously be adjusted.

Consequently, the indexing algorithm required for our partial index is adaptive. Depending on the workload, the algorithm must continuously insert index entries partition-wise into or delete them from partial indexes. The index maintenance effort of a partial index depends on factors such as the size of the indexed table, the details of the indexing algorithm (e.g., the access classification of data as frequently or rarely accessed), the executed database workload, and the underlying data structure of the partial index. In this work, we focus on the latter aspect and identify an index implementation suitable for realizing our partial index.

3. INDEX BENCHMARK

We developed an open-source benchmark framework[2] to evaluate single-attribute in-memory secondary indexes in their lookup speed, maintenance and memory performance. It allows to run certain benchmark cases with different index implementations on different datasets. The benchmark cases measure the required execution times of the index operations. In selected benchmark cases, the index's memory consumption is additionally measured. The index implementations included in the framework as well as the framework itself are written in `C++`.

3.1 Index Implementations

Unsync ART. The *Unsync ART* [1, 23] is an implementation of the ART (Adaptive Radix Tree), which is "a fast and space-efficient in-memory indexing structure specifically tuned for modern hardware" [22]. The ART is a specialized radix tree that adaptively and dynamically chooses a compact data structure for each individual internal node, depending on the number of child nodes.

MP Judy. Similar to the ART, the Judy Array is an adaptive radix tree [12]. It is a variant of a 256-way radix tree that is designed to reduce the number of cache misses by using over 20 different compression techniques. We use the Judy Array implementation of M. Pictor in this work [4].

[2]Source code at `https://github.com/mweisgut/IMIB`

TLX B+ Tree. This B+ tree, which is part of the *TLX* collection [14, 11], is an improved version of the *STX B+ Tree* [13]. According to the description of the predecessor, this data structure is an in-memory B+ tree that packs multiple value pairs into each node of the tree and thus reduces heap fragmentation and utilizes cache-line effects [13].

Abseil B-Tree. This B-tree, which is part of Google's open-source collection of `C++` libraries called *Abseil* [17, 18], is a cache-friendly and space-efficient B-tree implementation.

BB-Tree. The *BB-Tree* [2] is an "*almost-balanced* k-ary search tree, where inner nodes recursively split the data space into k partitions according to a delimiter dimension and $k-1$ delimiter values. Data objects are stored in leaf nodes (buckets). When too many data points are inserted (or deleted) and buckets overflow (or underflow), the structure is rebuilt to achieve a balance that is beneficial regarding the depths of leaves." [30].

RH Flat Map. The *Robin Hood (RH) Flat Map* [5] is a fast and memory efficient hash map. For collision resolution, it is based on robin hood hashing. The hash map stores its data in a flat array, which results in very fast access operations.

RH Node Map. Similar to the *Robin Hood Flat Map*, the *Robin Hood Node Map* [5] is a fast and memory efficient hash map, based on robin hood hashing. Unlike the *RH Flat Map*, the *RH Node Map* stores its data with node indirection.

PG Skip List. A skip list is a linked list extended by additional pointers to skip nodes when searching for certain stored elements [27]. The skip list included in the benchmark framework is an implementation of P. Goodliffe [7].

TSL Robin Map. Similar to the above listed *RH* Maps, *Tesssil's (TSL) Robin Map* [6] is a fast hash map based on robin hood hashing.

TSL Sparse Map. *Tessil's (TSL) Sparse Map* [9] is a memory efficient hash map that uses sparse quadratic probing. The design goal of this map is to be the most memory efficient possible while keeping reasonable performance.

STD Hash Map. This map is provided by the `C++` standard library as `unordered_map`. It "is an associative container that contains key-value pairs with unique keys" [10].

3.2 Index Operations

Insert (I). The *Insert* operation creates a new key-value entry in the index. Therefore, the key and the value have to be provided as the operation's input parameters. If an entry with the given key exists in the index, the entry is not added. An exception is the *PG Skip List*, which overwrites the existing entry.

Delete (D). The *Delete* operation removes the stored entry that contains a given key from the index.

Equality Lookup (EL). The *Equality Lookup* expects a key as an input parameter and returns the associated value if the index contains a key-value entry with the given key.

Range Lookup (RL). The *Range Lookup* operation expects two keys as input parameters, representing a key range, and returns the values of all the stored entries whose key is within the given key range.

Bulk Insert (BI). Inserting a large number of entries is referred to as bulk loading. Since the *TLX B+ Tree* provides two different functionalities in this regard, we distinguish between a bulk insert and a bulk load. The *Bulk Insert* operation inserts multiple key-value entries into the index without the requirement of the index being empty before.

Bulk Load (BL). Similar to the *Bulk Insert*, the *Bulk Load* inserts multiple key-value entries into the index. Unlike the *Bulk Insert*, the index must be empty before. For example, a bulk load of the *TLX B+ Tree* means that for a given sorted sequence of index entries, the leaves are created first, and the overlying levels of the B+ tree are constructed afterward.

Index Implementation	**I**	**D**	**EL**	**RL**	**BI**	**BL**
Unsync ART	✓	✓	✓	✗	✗	✗
TLX B+ Tree	✓	✓	✓	✓	✓	✓*
Abseil B-Tree	✓	✓	✓	✓	✓	✗
BB-Tree	✓	✓	✓	✓	✗	✓
MP Judy	✓	✓	✓	✗	✗	✗
RH Flat/Node Map	✓	✓	✓	✗	✓	✗
PG Skip List	✓	✓	✓	✓	✓	✗
TSL Robin/Sparse Map	✓	✓	✓	✗	✓	✗
STD Hash Map	✓	✓	✓	✗	✓	✗

Table 2: Index operations supported by the evaluated index implementations. * The entries to be inserted must be sorted.

3.3 Benchmark Cases

The benchmark framework contains the benchmark cases *Insert*, *Delete*, *Equality Lookup*, *Range Lookup*, *Bulk Insert* and *Bulk Load*. These are designed to evaluate the index operations listed in Section 3.2. Due to space limitations, we only present the *Equality Lookup*, *Delete*, and *Bulk Insert* cases. The descriptions of all benchmark cases can be found in the code repository of our benchmark [3].

Equality Lookup. The time required to execute a given sequence of equality lookups is measured. Before executing and measuring the equality lookups, the index is initialized and a given sequence of index entries is inserted using the insert operation.

Delete. The execution time required to delete a given sequence of index entries using the delete operation is measured. Before the delete operations are executed and measured, the index is initialized and the entire sequence of index entries to be deleted is inserted using the insert operation. The order in which the entries are deleted corresponds to the insertion order.

Bulk Insert. The execution time required to insert a given sequence of index entries using the bulk insert operation is measured. Before the execution and measurement, the index is initialized. Besides, the index's memory footprint is measured. Therefore, the allocated memory is measured before the index's initialization and after inserting the index entries. The difference of the allocated memory values is considered the index's memory consumption.

3.4 Datasets

To execute the different benchmark cases for the index implementations, different data is needed. For all the benchmark cases, a sequence of keys is required as input for the benchmark. This input represents a sequence of index entries, where a single key represents the search-key value and the position of the key, starting at one, represents the tuple identifier (TID). We refer to this input data as *entry keys.*

We use dense and sparse unsigned integer values as the entry keys to evaluate the index implementations. Additionally, we consider both ascending sorted and unsorted keys. The sorted dense dataset contains sequentially increasing, consecutive integer values, starting at one. For the unsorted dense dataset, the sorted dense keys are randomly shuffled. The unsorted sparse data sequence contains randomly picked, unique unsigned integer values. This sequence is sorted in ascending order for the sorted sparse data.

Additionally, search-key values and ranges of search-key values are required as input data for the *Equality Lookup* and the *Range Lookup* experiments, respectively. From a sequence of entry keys, a number of keys is randomly selected as equality lookups data sequence. For the range lookup key ranges, a number of key ranges is selected from a given sequence of entry keys. For each of the selected ranges, the number of qualifying keys is the same.

For our evaluation, we use five different sizes per combination of dense/sparse and ascending/random entry keys. The datasets have the sizes of two, four, six, eight, and ten million entry keys. Furthermore, we use one million lookup operations in the respective benchmark cases. For the range lookups, we used a selectivity of 0.01%.

4. EVALUATION

Since all of the index implementations listed in Section 3.1 support unique keys but not non-unique keys, we only use datasets with unique entry keys in our evaluation. For a given dataset, we ran the benchmark cases using 64-bit unsigned integers as search-keys and TIDs. The measurements reported are the median of 12 runs.

Join operations between foreign and primary keys are an exemplary database use case in which an index for a column with unique unsigned integer values can be utilized. Primary keys are often IDs, which are often represented as unsigned integer values. This applies, for example, when primary keys are surrogate keys.

The index benchmark cases were executed on a machine with four Intel(R) Xeon(R) CPU E7-4880 v2 CPUs on which each socket serves as a non-uniform memory access (NUMA) node. Each NUMA node has 512 GB of DDR3-1600 memory. Benchmarks were bound to a single NUMA node using `numactl -N 2 -m 2`. The index benchmark was compiled with `clang 9.0.1-12` and the options `-O3 -march=native`. We used the `C++` standard library version `libstdc++.so.6.0.28`.

4.1 Results

Due to space limitations, we only show the benchmark results for the *Equality Lookup*, *Delete*, and *Bulk Insert* benchmark cases. The remaining benchmark results can be found in our code repository [3].

Equality Lookups. Figure 1 shows the cumulative execution time of one million equality lookups performed with various entry key distributions, entry key orderings, and numbers of entries stored in the index. In each quadrant, the measurements of one data distribution and ordering combination is shown for different numbers of stored index entries.

As a reference, we included the lookup duration of the *Sorted Vector*, which is a dynamic array (`std::vector`) that stores its values in sorted order. To keep the dynamic array sorted all the time, new values are inserted into the proper position. To find the position of certain values during a lookup operation, a binary search is performed.

Delete. Figure 2 shows the cumulative execution time of deleting various numbers of index entries using the delete operation. The data characteristics of the entry keys vary in the four different quadrants.

Bulk Insert. Figure 3 shows the execution time required to insert various numbers of index entries using the bulk insert operation. The data characteristics of the entry keys vary in the four different quadrants.

Simple Vector and *Sorted Vector* are included as reference. The bulk insert of the *Simple Vector* consecutively inserts the sequence of given entries to a dynamic array (`std::vector`). The *Sorted Vector* additionally sorts the dynamic array after finishing the insertion.

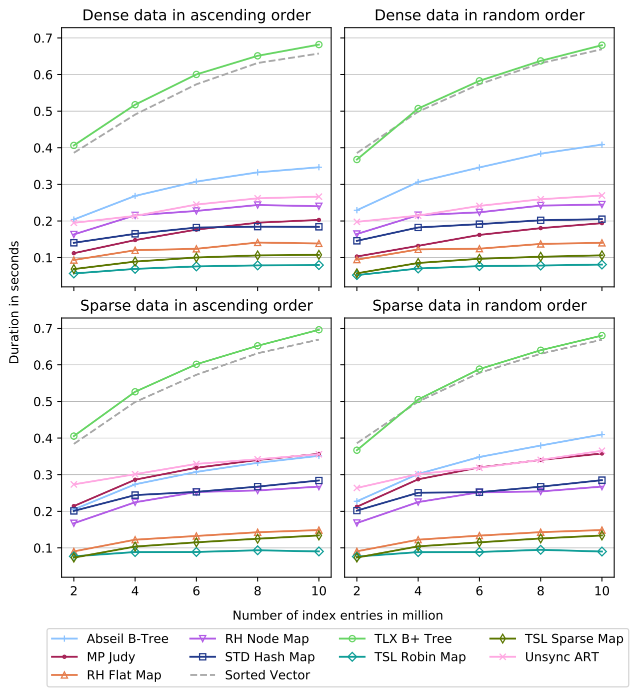

Figure 1: Cumulative duration of one million equality lookups for different numbers of stored index entries.

Based on the evaluation measurements, we subjectively categorized these measurements into the categories *low (L)*, *moderate (M)* and *high (H)* as shown in Table 3. Since the measurements are execution times and memory consumptions, respectively, lower categories are better. We did not categorize the execution times of range lookups and bulk loads since the former is supported only by the B-tree implementations and the latter only by the *TLX B+ Tree*.

Non-Competitive Implementations. Executions of different benchmark cases showed that the index operations could not be executed in a reasonable time for the *BB-Tree* and the *PG Skip List*. Therefore, we excluded these implementations from further evaluation experiments.

Key Findings. As can be seen in Table 3, the *RH Flat Map*, *TSL Sparse Map*, and *Unsync ART* are the only implementations that achieved good or moderate performance on all datasets in the evaluated aspects. Whereas the *RH Flat Map* shows overall lower operation execution times compared to the *TSL Sparse Map*, the latter has a lower memory footprint. The *Unsync ART* achieves an overall moderate performance in the evaluated aspects. Unfortunately, it does not support bulk inserts and provides a faulty delete operation. Apart from the high memory consumption of the *TSL Robin Map* on all datasets, this hash map achieves overall low operation execution times. In fact, it mostly requires lower equality lookup, delete and bulk insert execution times compared to the *RH Flat Map* and *TSL Sparse Map*.

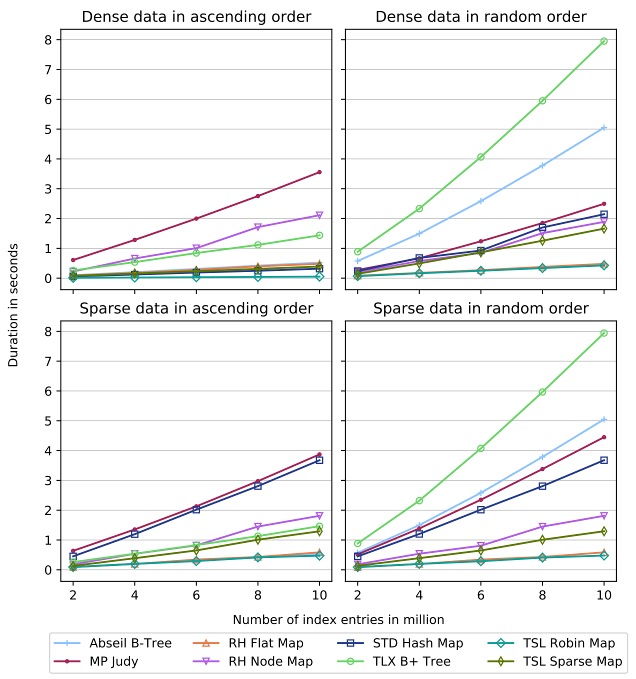

Figure 2: Cumulative duration of the *Delete* operation for different sized sets of index entries.

Aspect	Data		Index implementation								
	Distribution	Ordering	Abseil B-Tree	MP Judy	RH Flat Map	RH Node Map	STD Hash Map	TLX B+ Tree	TSL Robin Map	TSL Sparse Map	Unsync ART
Equality Lookup duration	dense	ascending	H	M	L	M	M	H	L	L	M
	dense	random	H	M	L	M	M	H	L	L	M
	sparse	ascending	M	M	L	M	M	H	L	L	M
	sparse	random	M	M	L	M	M	H	L	L	M
Insert duration	dense	ascending	L	M	M	H	L	M	M	L	M
	dense	random	H	M	L	M	M	H	M	M	L
	sparse	ascending	L	L	M	M	H	M	M	M	M
	sparse	random	H	M	L	M	H	H	M	M	M
Delete duration	dense	ascending	L	H	L	M	L	M	L	L	-
	dense	random	H	M	L	M	M	H	L	M	-
	sparse	ascending	L	H	L	M	H	M	L	M	-
	sparse	random	H	H	L	M	H	H	L	M	-
Bulk Insert duration	dense	ascending	L	-	M	H	L	M	L	L	-
	dense	random	H	-	L	M	L	H	L	M	-
	sparse	ascending	L	-	M	H	H	M	M	M	-
	sparse	random	H	-	L	M	M	H	L	L	-
Memory foot-print	dense	ascending	L	L	M	M	M	M	H	L	L
	dense	random	L	L	M	M	M	M	H	L	L
	sparse	ascending	L	M	M	M	M	M	H	L	M
	sparse	random	L	M	M	M	M	M	H	L	M

Table 3: Categorization of the measured values into *low (L)*, *moderate (M)* and *high (H)* (lower is better).

Furthermore, the B-trees, which are the only implementations supporting range queries, generally perform better on sorted data as shown in Table 3. The *Abseil B-Tree* performs generally better than the *TLX B+ Tree*. However, even on sorted data, both implementations achieved the highest equality lookup execution time.

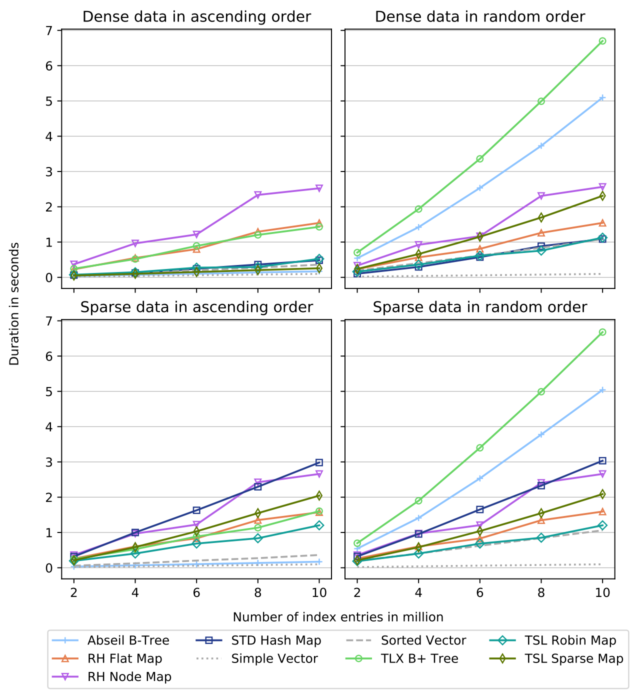

Figure 3: Duration of the *Bulk Insert* operation for different sized sets of index entries.

5. RELATED WORK

Partial Indexes. Stonebraker presented the first concept of partial indexes and how they can be used in a DBMS [31]. According to his specification, a single partial index is created for a subset of a table's data depending on a given qualification condition. Index entries are only created for those tuples that satisfy the given condition.

Seshadri and Swami extended Stonebraker's idea and proposed a generalized partial indexing concept, which includes six different strategies that can be used to create partial indexes [29]. These strategies use various statistical information such as the distribution of column accesses by queries, query predicate values, and data values to decide which indexes to be created.

Database Cracking [19] is an adaptive indexing technique that partially indexes columns. Using this technique, indexes are created "adaptively and incrementally as a side-product of query processing" [28]. The core cracking algorithm creates a copy of a column – a so-called *cracker column* – and incrementally sorts it during the execution of range queries [19]. The incremental sorting is realized by splitting the cracker column into multiple partitions – so-called *column slices* – with certain value ranges. During the execution of a range query, the cracker column or already existing column slices, respectively, are split based on the range predicate's boundaries. The partially sorted cracker column is utilized by database operations to speed up the execution time.

Olma et al. recently proposed an online adaptive partitioning and indexing algorithm for in situ query processing engines [26], which they integrated into their query processing system *Slalom*. Their approach reduces data access costs by partitioning a raw dataset into partitions and applying indexing strategies for each partition individually. Based on continuously collected statistics, their algorithm performs the partitioning and indexing in a gradual manner as a side effect of query processing. Thus, a priori workload knowledge is not required. Based on access frequencies, the algorithm decides individually for each partition whether and which in-memory partition index is to be created. Thus, instead of indexing the entire dataset, indexes exist only for subset of frequently accessed partitions.

Olma et al.'s algorithm shares similarities with our partial index approach: Both approaches refine the set of existing indexes automatically and dynamically based on data access frequencies. In addition, both approaches choose the set of data to be indexed on partition granularity. However, while their algorithm creates indexes partition-wise, our approach creates indexes table-wise across selected partitions. Additionally, whereas Olma et al.'s work refers to in situ query processing systems, our approach is designed for in-memory DBMS. Thus, their indexes are created for partitions of raw data files whereas our indexes are to be created for table data stored in main memory.

Index Evaluations and Benchmarks. Xie et al. conducted an experimental performance evaluation of five modern in-memory database indexes [32]. Their aspects of investigation were the throughput, latency, scalability, memory consumption and cache miss rate. They executed equality lookups, inserts and updates with varying experiment configurations in different dimensions. These dimensions are the size of the dataset, the number of execution threads, the ratios of equality lookup, insert, and update operations, and the skewness in the workload and dataset.

Alvarez and his fellow researchers presented an experimental performance evaluation between the *ART* and the *Judy Array* and different hash table variants [12]. Their evaluation aspects are the insertion and equality lookup throughput and the memory footprint.

Kipf, Marcus and their co-authors developed an open-source index benchmark framework called *Search On Sorted Data Benchmark (SOSD)* [8] that allows researchers to compare in-memory search algorithms, including (learned) indexes, on equality lookup performance over sorted data [20]. According to the SOSD authors, their framework contains diverse synthetic and real-world datasets, optimized baseline implementations, and the "first performant and publicly available implementation of the [learned] Recursive Model Index (RMI)" [20], which is proposed by Kraska et al. [21]. Marcus, Kipf et al. extended their initial work about the SOSD in a follow-up work [25]. In this work, they presented their framework in more detail and conducted a performance analysis of three recent learned index structures.

6. CONCLUSION AND FUTURE WORK

In this work, we presented the idea of a partial index that is created table-wise but only indexes data of frequently accessed partitions. Using our index benchmark framework, we evaluated the lookup speed, maintenance cost and memory consumption of various index implementations to identify a suitable implementation for partial indexes.

Since our partial indexes are designed to execute maintenance operations on a partition level, the efficiency of bulk operations is particularly relevant. Regarding the equality lookup, delete and bulk insert performance, the *TSL Robin Map* achieved the best overall results.

As a next step, we plan to implement the proposed partial index in our research database Hyrise so that we can perform end-to-end DBMS performance evaluations afterward.

7. ACKNOWLEDGMENTS

I sincerely thank my supervisors Jan Kossmann and Markus Dreseler, who supported me intensively and always gave me constructive impulses and detailed, critical feedback. Furthermore, I would like to thank the anonymous reviewers for their valuable feedback.

8. REFERENCES

[1] ARTSynchronized. `https://github.com/flode/ARTSynchronized`, Accessed: 2021-03-01.

[2] BB-Tree. `https://github.com/flippingbits/bb-tree`, Accessed: 2021-03-01.

[3] Index Benchmark Evaluation Results. `https://github.com/mweisgut/IMIB/tree/gvdb21/evaluation_results`, Accessed: 2021-03-02.

[4] Judy Template. `https://github.com/mpictor/judy-template`, Accessed: 2021-03-01.

[5] Robin Hood Map. `https://github.com/martinus/robin-hood-hashing`, Accessed: 2021-03-01.

[6] Robin Map. `https://github.com/Tessil/robin-map`, Accessed: 2021-03-01.

[7] Skip List. `https://github.com/petegoodliffe/skip_list`, Accessed: 2021-03-01.

[8] SOSD. `https://github.com/learnedsystems/SOSD`, Accessed: 2021-03-01.

[9] Sparse Map. `https://github.com/Tessil/sparse-mapp`, Accessed: 2021-03-01.

[10] std::unordered_map. `https://en.cppreference.com/w/cpp/container/unordered_map`, Accessed: 2021-03-01.

[11] TLX. `https://github.com/tlx/tlx`, Accessed: 2021-03-01.

[12] V. Alvarez, S. Richter, X. Chen, and J. Dittrich. A comparison of adaptive radix trees and hash tables. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1227–1238, 2015.

[13] T. Bingmann. STX B+ Tree C++ Template Classes, 2013. `https://panthema.net/2007/stx-btree/`, Accessed: 2021-03-01.

[14] T. Bingmann. TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers, 2018. `https://panthema.net/tlx`, Accessed 2021-03-01.

[15] C. Chasseur and J. M. Patel. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 6(13):1474–1485, 2013.

[16] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 313–324, 2019.

[17] Google Inc. Abseil. `https://abseil.io`, Accessed: 2021-03-01.

[18] Google Inc. Abseil - C++ Common Libraries. `https://github.com/abseil/abseil-cpp`, Accessed: 2021-03-01.

[19] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, 2007.

[20] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[21] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 489–504, 2018.

[22] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 38–49, 2013.

[23] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of practical synchronization. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, 2016.

[24] L. Ma, D. V. Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2018.

[25] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking Learned Indexes. *PVLDB*, 2021.

[26] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal*, 29(1):569–591, 2020.

[27] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, 33(6):668–676, 1990.

[28] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 7(2):97–108, 2013.

[29] P. Seshadri and A. N. Swami. Generalized partial indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 420–427, 1995.

[30] S. Sprenger, P. Schäfer, and U. Leser. Bb-tree: A practical and efficient main-memory index structure for multidimensional workloads. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 169–180, 2019.

[31] M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[32] Z. Xie, Q. Cai, G. Chen, R. Mao, and M. Zhang. A Comprehensive Performance Evaluation of Modern In-Memory Indices. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2018.