Hannes Voigt, Tobias Jaekel, Thomas Kissinger, Wolfgang Lehner

Adaptive Index Buffer

Erstveröffentlichung in / First published in:

IEEE 28th International Conference on Data Engineering Workshops. Arlington, 01.04-05.04.2012. IEEE, S. 308-314. ISBN 978-1-4673-1640-8

DOI: https://doi.org/10.1109/ICDEW.2012.39

Final edited form was published in "IEEE 28th International Conference on Data Engineering Workshops. Arlington 2012", S.308-314, ISBN 978-1-4673-1640-8
https://doi.org/10.1109/ICDEW.2012.39

Adaptive Index Buffer

Hannes Voigt, Tobias Jaekel, Thomas Kissinger, Wolfgang Lehner

Database Technology Group
Dresden University of Technology
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

***Abstract*— With rapidly increasing datasets and more dynamic workloads, adaptive partial indexing becomes an important way to keep indexing efficiently. During times of changing workloads, the query performance suffers from inefficient tables scans while the index tuning mechanism adapts the partial index. In this paper we present the Adaptive Index Buffer. The Adaptive Index Buffer reduces the cost of table scans by quickly indexing tuples in memory until the partial index has adapted to the workload again. We explain the basic operating mode of an Index Buffer and discuss how it adapts to changing workload situations. Further, we present three experiments that show the Index Buffer at work.**

I. Introduction

With constantly growing datasets partial indexing becomes more and more important. Partial indexing focuses indexes on the data of interest [1], [2]. Data of less interest, e.g., data primarily kept for reasons of revision, linage, and versioning, remain unindexed and do not allocated storage, memory and maintenance resources. The main drawback of partial indexes, of course, is high cost for users that query unindexed data. Having stable workloads, the DBA can reduce this effect by carefully designing and adjusting the definition of the partial indexes. In dynamic workload environments, an online tuning facility continuously adapts the partial indexes.

Partial indexing is especially powerful for secondary indexes. A generally interesting tuple is not necessarily interesting for every secondary index. For instance, cheap products may be often queried by price, whereas well known and specifically advertised products may be often queried by their name. Since secondary indexes are purely redundant data managed for fast retrieval, it is even more important to focus secondary indexes on the tuples that are actually retrieved through that access path.

Index adaptation is not for free. Adding and removing entries from an index involves I/O and memory activities, which add to the total execution cost of the database workload. To gain cost benefits, the achieved reduction of query execution costs must outweigh the adaptation costs. In dynamic environments the future workload is not predictable. Hence, an online tuning facility triggers only adaptation steps that would have provided enough query cost reductions in the recent past to sanctify their adaptation costs. Inevitably, the necessary monitoring time introduces a control loop delay to every online tuning.

Figure 1 illustrates the control loop delay with a simulation of an adaptive partial indexing. Queried is a single column

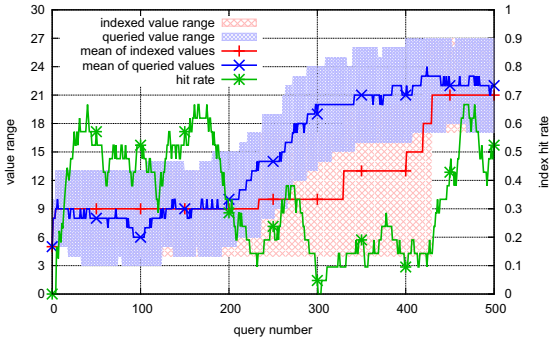

Fig. 1. Control Loop Delay in Adaptive Partial Indexing

of integer values. The simulated tuning mechanism indexes a queried value if it has shown enough potential query cost reduction during the last twenty queries. For simplicity of the simulation, a value is assumed to reach the threshold if it was queried at least six times in the monitoring window. Entries are removed from the index based on a least recently used strategy. The simulation runs for 500 queries. Between query 200 and 300 the focus of the queries shifts from values less 15 to values greater 15 – depicted with the queried value range in the figure. The indexed value range depicts the values covered by the partial index. As can be clearly seen, it takes the tuning mechanism about 200 queries to follow the workload change. During adaptation time the hit rate of the partial index drops significantly. With a low partial index hit rate, the rate of expensive table scans increases. Because of this effect, workload changes burden the database system twice. The required adaptation adds to the total execution costs and additionally the query costs increase because of a suboptimal index configuration.

In this paper, we address this double burden of workload changes. We present the Adaptive Index Buffer, a technique to speed up table scans under the presence of partial secondary indexes. The technique builds on the index-aware usage of the database buffer. In-memory and without need for recovery, the Index Buffer can build up index information with significantly lower cost than adapting traditional indexes. Consequently, the Adaptive Index Buffer exhibits a notably shorter control loop delay than traditional index adaptation. As a scratch pad index, the Adaptive Index Buffer backs the partial indexes during times of unstable workload. It helps during spikes of deviating

Final edited form was published in "IEEE 28th International Conference on Data Engineering Workshops. Arlington 2012", S.308-314, ISBN 978-1-4673-1640-8
https://doi.org/10.1109/ICDEW.2012.39

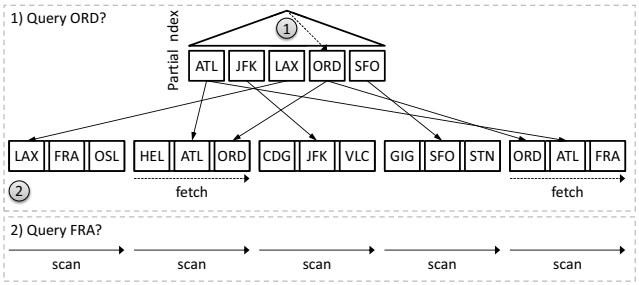

Fig. 2. Partial index

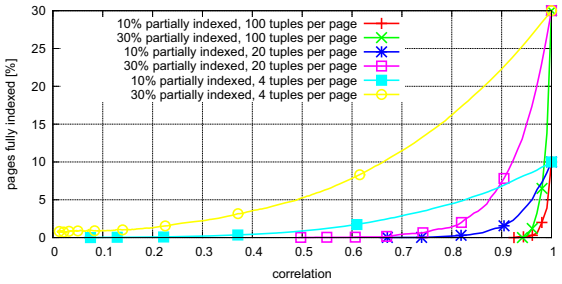

Fig. 3. Share of fully indexed pages with partial indexing

workload, too short to adapt the disk-based partial index, as well as during periods of workload changes.

The remainder of the paper first takes a closer look at partial indexing and its effects on queries not covered by a partial index (Section II). Second, Section III presents the Index Buffer; how it collects index information and speeds up table scans. Third, we discuss the adaption mechanism of the Index Buffer; how it discards collected index information and how it balances between multiple partial indexes (Section IV). Fourth, Section V talks about the results of the evaluation we conducted. Finally, we outline how other approaches related to the concept of the Index Buffer (Section VI) and conclude the paper (Section VII).

II. Partial Indexing

Partial indexes cover only a subset of the values of a column. Consider the example in Figure 2. Assume a database about the on-time performance of international flights. It lists all flights with their departure times, arrival times, and delays. Since the provider mainly sells reports to U.S. airports, it queries the flights most frequently by U.S. airports and uses a partial index on its airport column. As shown in the figure, the index only contains U.S. airports. If the database is queried by the Chicago O'Hare International Aiport (ORD), the database system can use the partial index to answer the query efficiently. It scans the index for ORD and then fetches the qualifying tuple from their respective pages. Because the index is partial, it only requires storage and maintenances resources for U.S. airports.

However, if the provider suddenly creates reports for German airports, the database system cannot use the index. As shown in the figure, a query for Frankfurt Airport (FRA) can only be answered with a full scan of the table. Until the partial index adapts to the new workload situation, the system executes the queries for German reports very inefficiently. In consequence, not only the German reports take longer, but also occupy significantly more resources than their share in the workload would suggest.

Even if a query does not hit the partial index, the index provides useful information. All tuples referenced in the index will not be part of the result set – otherwise the query would have hit the index – and can be safely excluded from the set of candidates. However, the database system can make only use of this knowledge to reduce its I/O if a page solely contains tuples indexed in the partial index. In such a case, the database system can skip the page during the table scan without missing a matching tuple. If only a single tuple in a page is not covered by the partial index, though, the database system has to read the whole page.

The probability of pages fully indexed by a partial index depends on three aspects: (1) the number of tuples per page, (2) the probability of tuples to be in the partial index, and (3) how strong the physical order of the tuples correlates with their logical order regarding the partial index. We have simulated different correlations between logical order and physical order. The simulation started with a logically order set of tuples (correlation equals 1) and gradually swapped randomly picked tuples to decrease the correlation. In each step, we counted the number of fully indexed pages. Figure 3 shows the results for six scenarios. All scenarios are based on 100,000 tuples. For perfectly clustered data (i.e., correlation equals 1), the fraction of fully indexed pages corresponds to the number of tuples covered by the partial index. As can be seen in the figure, the fraction drops quickly with decreasing correlation. For typically page sizes of 10 or more tuples and a correlation of 0.8 or less, less than 5% of the pages remain fully indexed. In general, the physical order and the logical order are barely correlated for multiple secondary indexes. Consequently, pages that are completely covered by a partial index are extremely unlikely and in the outstanding majority of cases the database system has to perform a full table scan.

III. Index Buffer

The Index Buffer leverages the idea of skipping fully indexed pages. As we have shown, pages fully indexed by a partial index are very unlikely. The index buffer complements the partial index by covering the remaining unindexed tuples of a page. This way, more pages become fully indexed and can be skipped safely during a table scan.

Figure 4 illustrates the idea. Here, the Index Buffer indexes the remaining unindexed tuples of Page 2 and Page 5 – HEL and FRA, respectively – so that Page 2 and Page 5 are fully indexed. Now, the table scan, which is necessary to answer a query for Frankfurt Airport, can skip these two pages, which significantly reduces its I/O cost. To find all matching tuples,

Final edited form was published in "IEEE 28th International Conference on Data Engineering Workshops. Arlington 2012", S.308-314, ISBN 978-1-4673-1640-8
https://doi.org/10.1109/ICDEW.2012.39

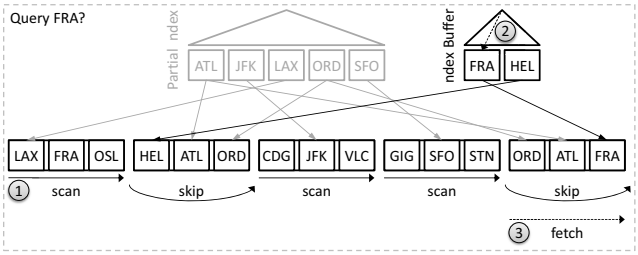

Fig. 4. Partial index with Index Buffer

the database system additionally scans the Index Buffer. In the example, this yields the second FRA tuple from Page 5.

The Index Buffer is an index structure residing within the database buffer. As a helping structure during times of partial index adaptation, the Index Buffer should only consume or occupy resources if it is actually used while it must be quickly available if it is needed. Main memory allows quick adaptation at low cost, which ensures that the Index Buffer involves a significantly smaller Control Loop Delay than the partial index tuning. The Index Buffer builds on a normal B*-Tree [3]. Main memory-optimized index structures such as the CSB+-Tree [4] or a hash table can be used too. Which particular index structure is used is not essential for the general idea of the Index Buffer. Additionally to the B*-Tree, the Index Buffer maintains a counter $C[p]$ for each page p that represents the number of unindexed tuples in the page. The array of all counters is initialized during the creation of the partial index. Every counter is initially set to the number of tuples in the page minus the tuples covered by the partial index.

Queries that do not hit a partial index build up the Index Buffer. While such a query scans the table, the system inserts the so far unindexed tuples of pages chosen beforehand. Algorithm 1 shows the basic procedure. Assume a query with a predicate q, which is not part of the partial index. The algorithm consists of three essential steps. First, the system selects the pages that should be indexed in the Index Buffer (Line 7). Second, the algorithm scans the Index Buffer and adds qualifying tuples to the result (Line 8–10). Third, it scans the table (Line 11–17). Pages with a counter equal zero are skipped, since they are fully indexed. For all other pages, the system scans through all tuples in the page. If a tuple matches the query predicate, it is added to the result set (Line 14). If the page was selected to be indexed and the scanned tuple is not covered by the partial index, the system adds the tuple to the Index Buffer (Line 16). Additionally, it sets the corresponding page counter to zero.

During each table scan, the system indexes a set I of pages in the Index Buffer. This results in a fixed benefit of $|I|$ pages that can be skipped by the next scan. The page counters allow picking the pages most beneficial for indexing. Since the goal is to skip pages during table scans, pages with many already indexed tuples are more valuable for the Index Buffer. The benefit of $|I|$ skippable pages is achieved with less buffer space compared to pages with fewer tuples already indexed in the

Algorithm 1 Indexing Table Scan

```
1:  procedure IndexingScan(R, q, C, B)
2:                                          ▷ q: queried predicate
3:                                          ▷ R: set of pages to scan
4:                                          ▷ C: counters of unindexed tuples
5:                                          ▷ B: Index Buffer for queried column
6:      Q ← ∅                               ▷ initialize result set
7:      I ← SelectPagesForBuffer()
8:      for t ∈ B do                        ▷ Index Buffer scan
9:          if q(t) then                    ▷ tuple matches predicate
10:             Q ← Q ∪ {t}
11:     for p ∈ R with C[p] > 0 do          ▷ table scan
12:         for t ∈ p do
13:             if q(t) then
14:                 Q ← Q ∪ {t}
15:             if p ∈ I ∧ t ∉ IX then
16:                 B.Add(t)
17:         C[p] ← 0
18:     return Q
```

partial index. Note that the system determines I dynamically depending on the current index buffer utilization.

The Index Buffer maintains the B*-Tree index and the counters during inserts, updates, deletes and partial index adaptions. Which operation the system has to perform depends (1) if the old tuple t_{old} was in the partial index, (2) if the updated tuple t_{new} will be in the partial index, (3) if the old page p_{old} that contained the tuple is in the Index Buffer, and (4) if the new page p_{new} that will contain the new tuple is in the Index Buffer. Table I lists the different maintenance scenarios with necessary operations.

IV. Index Buffer Management

With various partial indexes in a database, multiple Index Buffers are created over time. All Index Buffers reside in the Index Buffer Space, a share of the database buffer of limited size. The database system controls the size of the Index Buffer Space before it adds new entries with a tables scan. In case the new entries would cause the Index Buffer Space to exceed its configured space bound, the system discards index information from the space to maintain the limit.

The purpose of the Index Buffer is to allow page skipping during a table scan. A single entry in the Index Buffer can reference multiple pages. Further, a single page can be referenced by multiple index entries. Hence, discarding a single entry from the Index Buffer has a double negative effect. First, one or more pages obtain an unindexed tuple and cannot be skipped anymore. Second, all other entries in the Index Buffer referencing these pages occupy memory in the Index Buffer Space without creating any benefit. In consequence, discarding single entries from the Index Buffer contradicts the buffer's purpose.

For the precise and efficient discarding of entries from an Index Buffer, we partition the B*-Tree of an Index Buffer.

Final edited form was published in "IEEE 28th International Conference on Data Engineering Workshops. Arlington 2012", S.308-314, ISBN 978-1-4673-1640-8
https://doi.org/10.1109/ICDEW.2012.39

TABLE I
INDEX BUFFER MAINTENANCE

		$t_{\text{old}} \in IX$		$t_{\text{old}} \notin IX$	
		$t_{\text{new}} \in IX$	$t_{\text{new}} \notin IX$	$t_{\text{new}} \in IX$	$t_{\text{new}} \notin IX$
		IX.Update($t_{\text{old}}, t_{\text{new}}$)	IX.Remove(t_{old})	IX.Add(t_{new})	-
$p_{\text{old}} \in B$	$p_{\text{new}} \in B$	-	B.Add(t_{new})	B.Remove(t_{old})	B.Update($t_{\text{old}}, t_{\text{new}}$)
	$p_{\text{new}} \notin B$	-	Cp_{new}++	B.Remove(t_{old})	B.Remove(t_{old}), Cp_{new}++
$p_{\text{old}} \notin B$	$p_{\text{new}} \in B$	-	B.Add(t_{new})	Cp_{old}−−	B.Add(t_{new}), Cp_{old}−−
	$p_{\text{new}} \notin B$	-	Cp_{new}++	Cp_{old}−−	Cp_{old}−−, Cp_{new}++

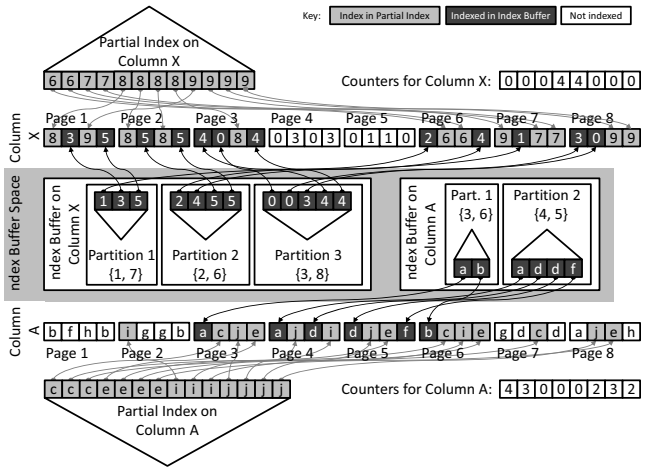

Fig. 5. Partitioned Index Buffers

Each partition covers P pages of the table, so that, the partitions are disjunct in the sets of pages they reference. For instance, if an index entry in Partition 3 references Page 8, all other Index Buffer entries that reference Page 8, as well, are in Partition 3, too. In case the system decides to discard index information, it always drops complete partitions form the Index Buffer Space. This efficiently discards all entries that reference a set of pages.

Figure 5 shows an example with two Index Buffers in the Index Buffer Space. The first Index Buffer complements the partial index on Column X and is partitioned in three partitions. Each partition indexes $P = 2$ pages. For instance, Partition 1 indexes the three tuples in Page 1 and Page 7, which are not covered by the partial index. The second Index Buffer complements the partial index on Column A and is partitioned in two partitions. Note that, similar to normal secondary indexes, it is insignificant for the separation of Index Buffers whether the columns are in the same table or not.

The Index Buffer Space is managed based on the benefit and the size of index information. New index information can displace old index information of same or larger size if it provides more benefit than the old index information. The benefit of index information in the Index Buffer depends on two factors: (1) The number of pages covered by the index information and (2) how frequent the index information is used. The higher the factors are the more pages the workload can skip. The size of the index information is the memory it requires to be stored. Generally, the number of entries in an index causes the size of the index.

TABLE II
LRU-K OPERATIONS

	Index Buffer B of queried column	Index Buffer B' of other columns
partial index hit	$H_B[0]$++	$H_{B'}[0]$++
no partial index hit	shift(H_B, +1); $H_B[0] = 0$	$H_{B'}[0]$++

How frequent an Index Buffer is used, is determined based on its access history. Analogously to the LRU-K algorithm [5], the system maintains a history H_B of the K last access intervals for each Index Buffer B. The average of the history of Index Buffer B gives the mean access interval $\mathcal{T}_B$ so that $\mathcal{T}_B = K^{-1} \cdot \sum_{H_B} H_B[i]$. Queries that the system can answer with the partial index do not utilize the Index Buffer. Consequently, the Index Buffer Management shifts the history to the next interval only if a query actually uses an Index Buffer. Table II summarizes the operations on the LRU-K history of Index Buffers in the cases the partial index of the queried column is hit or is not hit.

Further, let the number of pages covered a partition p be $\mathcal{X}_p$. Then, the benefit of partition p results from $b_p = \mathcal{X}_p \cdot \mathcal{T}_B^{-1}$ where B is the Index Buffer the partition belongs to. Accordingly, the benefit of an Index Buffer B is the sum of the benefits of its partitions:

$$b_B = \sum_{p \in B} b_p \ .$$

Similar to partitions, the benefit of new index information results from $b_I = |I| \cdot \mathcal{T}_I^{-1}$, where I is the set of pages to index and $\mathcal{T}_I$ is the mean access interval of the Index Buffer that will accommodate the new index information.

For the size of index information, we denote the number of entries in a partition p as n_p. Analogously, the size of new index information results from the number of entries to add, denoted as n_I. Based on the counters for unindexed tuples in pages, the system can easily determine n_I as

$$\sum_{s \in I} C[s] \ .$$

Further, let n_F be the free space left in the Index Buffer Space.

Final edited form was published in "IEEE 28th International Conference on Data Engineering Workshops. Arlington 2012", S.308-314, ISBN 978-1-4673-1640-8
https://doi.org/10.1109/ICDEW.2012.39

Algorithm 2 Select Pages for Indexation

```
1:  procedure SELECTPAGESFORBUFFER(R, C, S)
2:                                   ▷ R: set of pages to scan
3:                                   ▷ C: counters of unindexed tuples
4:                                   ▷ S: set of all partitions in Index Buffer Space
5:      D' ← ∅                       ▷ D': set to collect candidate partitions
6:      I' ← ∅                       ▷ I'': set to collect candidate pages
7:      repeat
8:          n_A ← n_F + Σ_{p∈D'} n_p
9:          I ← I'
10:         I'' ← I'
11:         repeat
12:             I' ← I''
13:             I'' ← I'' ∪ {SelectNextPage(C, R)}
14:         until n_{I''} > n_A ∨ |I''| > I^MAX
15:         b_{I'} ← |I'| · T_{I'}^{-1}
16:         D ← D'
17:         D' ← D' ∪ {SelectNextPartition(S)}
18:     until b_{I'} ≤ Σ_{p∈D'} b_p ∨ I' = I
19:     DropPartitions(D)
20:     return I
```

Before the system adds new index information to the Index Buffer, it has to select the pages it wants to index. At that point the system also checks the space bound of the Index Buffer Space and triggers displacement of old index information if required. The page selection routine ensures that there is enough Index Buffer Space available to index the pages it returns.

The management strategy of the Index Buffer wants to achieve two contrasting goals. On the one hand the Index Buffer should index as much pages as possible to be quickly of help. On the other hand existing index information should stay as long as possible in the Index Buffer to be present if needed. To balance between both goals, the management strategy indexes precisely so many pages that the resulting new index information is more benefical than the old index information that the system must discard to clear the space required for the new index information. Additionally, there is configurable upper bound for new index information per table scan.

Algorithm 2 shows the page selection routine. It returns the set I of pages for indexation (Line 20) and discard a set D of partitions to ensure that enough space is available (Line 19). To determine I and D, the routine gradually adds partitions to D (Line 17). In each iteration the algorithm preforms three steps. First, the algorithm determines the total size of the partitions in D (Line 8). Second, it selects the set of pages I so that the new index information will fit in the available space (Line 9–14). Specifically, the algorithm adds pages in ascending order of there counter C as long as

$$n_I \leq n_F + \sum_{p \in D} n_p \ .$$

At most the system indexes I^{MAX} pages during one table scan. Third, the algorithm determines the benefit of the new index information b_I resulting from an indexation of the pages in I (Line 15). The algorithm repeats the three steps as long as the benefit of indexing the pages in I is higher than the benefit of the partitions in D

$$b_I > \sum_{p \in D} b_p$$

or I does not change anymore (Line 18).

The system selects each partition in D following a two-staged selection algorithm. In the first stage, the algorithm randomly selects an Index Buffer B with the probability

$$\frac{b_B^{-1}}{\sum_{B' \in S \setminus B_N} b_{B'}^{-1}} \ ,$$

where S is the set of all Index Buffers in the Index Buffer Space and B_N is the Index Buffer the new entries should be added to. Index Buffers with a low benefit are more likely to be picked. In the second stage, the algorithm selects a partition from that Index Buffer. A possible incomplete partition ($\mathcal{X}_p < P$) has the lowest benefit within an Index Buffer and will be picked first. Afterwards, complete partitions are picked in descending order of their size (n_p), beause they have the same benefit.

V. Evaluation

To evaluate our Index Buffer approach, we conducted a series of experiments. We implemented the Index Buffer concept prototypical in the H2 Database Engine 1.3 [6]. We ran all experiments on an Intel Core 2 Duo U9600 processor at 1.6 GHz with 4 GB of DDR3 main memory and a 128 GB Samsung SSD. We used Microsoft Windows 7 64bit edition as the operating system and Java SE 6 as runtime environment.

For all experiments, we used a common data setup, which consists of a single table with three INTEGER columns (A,B,C) for indexing and one VARCHAR(512) column as payload. The integer columns are populated with random values uniformly distributed from 1 to 50,000. The size of the payload values is also uniformly distributed, but ranges from 1 to 512. We filled the table with 500,000 tuples, resulting in an effective table size of 220 MB on disk. In each column the top 10% of the value range are indexed in a partial index, i.e., values from 1 to 5,000. In experiment 1, 2 and 3 we queried the unindexed values randomly. The experiment 4 shows the case queries address also value covered by the partial index. In each experiment the workload consists of 200 queries.

The first experiment illustrates the basic behavior of a single Index Buffer. Accordingly, we queried only Column A. The Index Buffer Space was set to unlimited size, $I^{\text{MAX}} = 5{,}000$ pages were index at most during a table scan, and each Index Buffer partition indexed a maximum of $P = 10{,}000$ pages. For each query, we measured the runtime of individual queries, the total number of entries in the Index Buffer, and the number of pages that were skipped. Figure 6 shows the results. For comparison, the figure also shows the runtimes of the same queries without tables scan. As can be seen, the first couple

Final edited form was published in "IEEE 28th International Conference on Data Engineering Workshops. Arlington 2012", S.308-314, ISBN 978-1-4673-1640-8
https://doi.org/10.1109/ICDEW.2012.39

of queries exhibited a slightly longer runtime, but quickly the execution time dropped below the level of the table scan. The obvious reason is the Index Buffer, which indexed an increasing number of tuples. Quickly, the tables scan was able to skip a large number of pages. Since the Index Buffer Space was of unlimited size in this experiment, all pages were complete indexed after 20 queries. At that point, the query execution times had the level of an index scan.

The second experiment shows the influence of the maximum number of pages indexed per table scan I^{MAX} and the space bound of the Index Buffer Space L. We run this experiments with the same setting as the first experiment, expect that we varied I^{MAX} and L. The effects of I^{MAX} and L are independent from each other and can be seen in a single experiment. As Figure 7 shows, I^{MAX} determines how aggressive the Index Buffer indexes new pages. The higher I^{MAX}, the more pages were indexed during one scan. In consequence, the query execution times dropped more quickly within the first 15 queries for higher I^{MAX}. Of course, the Index Buffer also occupied Index Buffer Space more quickly for higher I^{MAX} (not shown in the figure). The size of the Index Buffer Space limited the maximum number of entries and with it the number of pages that could be skipped. As Figure 7 also shows, the smaller the Index Buffer Space, the less the Index Buffer could speed up table scans.

The third experiment shows the Index Buffer Management. We ran the workload of 200 queries on all three columns. Half of the queries selected tuples on Column A, one third on Column B, and one sixth on Column C. After 100 queries, we switched the query mix to: One sixth on Column A, one third on Column B, and one half on Column C. The Index Buffer Space was limited to 800,000 entries, at most $I^{\text{MAX}} = 5{,}000$ pages were index during each table scan, and each Index Buffer partition indexed a maximum of $P = 10{,}000$ pages. Figure 8 shows the number of entries in each of the three Index Buffers. In the first workload period the Index Buffer on Column A occupied more than half of the Index Buffer Space. The Index Buffer on Column B occupid most of the remaining Index Buffer Space, while the Index Buffer on Column C only sporadically accumulated entries. After the switch of the query mix, the situation quickly turned around. In the second workload period, the Index Buffer on Column C rapidly grew to roughly 55% of the Index Buffer Space and the Index Buffer on Column A practically shrinked to zero.

The fourth experiment considers the Index Buffer Management under the influence of varying partial index hit rates. The general setting is similar to the third experiment except that the queries on Column A also query values covered in the partial index on that column. To show the influence of the partial index hits on the allocation of Index Buffer Space, we switched the definition of the partial index after 100 queries. Queries on Column A among first 100 queries hit the partial index with a probability of 80%. During the following 100 queries the partial index hit rate for Column A queries is only 20%. The query distribution is fixed during the complete workload. Specifically, half of the queries are against

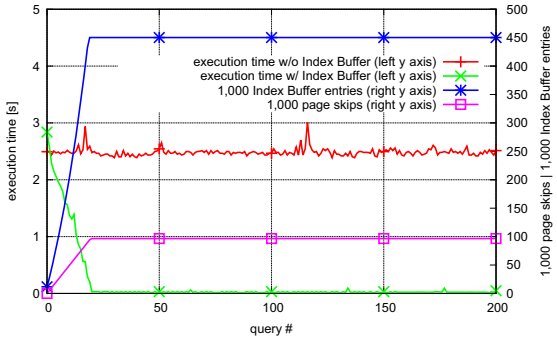

Fig. 6. Single Index Buffer with $I^{\text{MAX}} = 5{,}000$

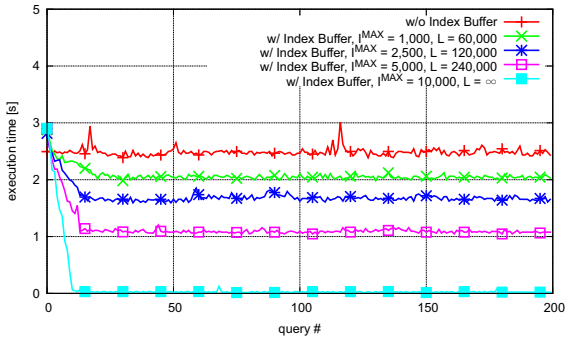

Fig. 7. Single Index Buffer with varying Index Buffer Space size and I^{MAX}

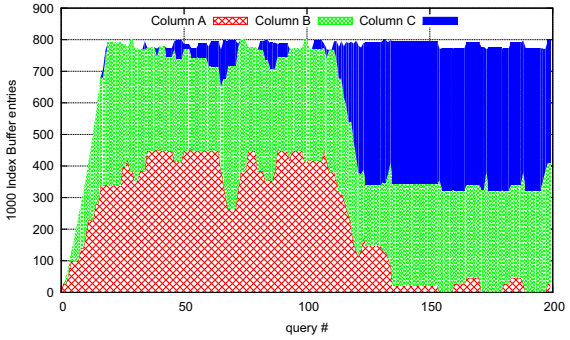

Fig. 8. Three Index Buffers with limited space

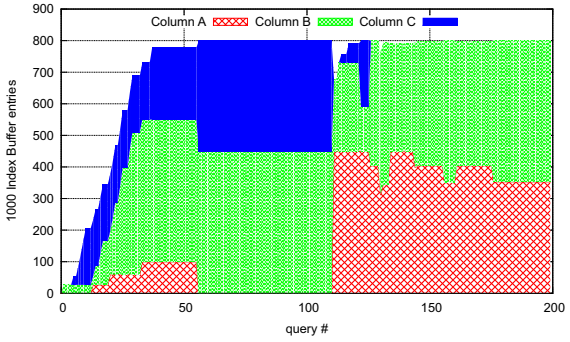

Fig. 9. Three Index Buffers with limited space and hits on the partial index of Column A

Final edited form was published in "IEEE 28th International Conference on Data Engineering Workshops. Arlington 2012", S.308-314, ISBN 978-1-4673-1640-8
https://doi.org/10.1109/ICDEW.2012.39

Column A, one third against Column B, and one sixth against Column C. The Index Buffer Space settings were the same as in experiment three. The maximum number of pages to index in one table scan was set to $I^{\text{MAX}} = 10{,}000$ and the maximum number of pages an Index Buffer partition can index was set to $P = 10{,}000$. Figure 9 shows the three Index Buffers and their entries in the Index Buffer Space. After the Index Buffer Space was filled the Index Buffers competed for space. For the first period the partial index of Column A was hit frequently and the Index Buffer Manager decided to give less space to this Index Buffer. Thus, Column B and Column C gain more Index Buffer Space though they are queried less often. This state changed after 100 queries and the Index Buffer of Column A was used more often than in the first period. As can be seen the Index Buffer of Column A got more Index Buffer Space and grew quickly after the partial index hit rate changed, whereas the Index Buffers of Column B and Column C shrink.

VI. Related Work

Substantial research has been done in the field of index tuning. First research in that area dates back to the late 1970s. Nowadays commercial database management systems offer index tuning tools, [7], [8], [9], which recommend an index configuration for a given workload and a storage bound the configuration has to fit into. In their core, these tools base on a *what-if* interface to the optimizer [10], to evaluate possible index configurations. What-if calls are expensive since they involve a complete logical query processing.

However, all these state-of-the-art tools consider the database workload as static and predictable. If the workload changes unpredictably, the user has to notice this change and rerun the tool. Since this is prohibitively labor-intensive with dynamic workloads, research concentrated on autonomous index tuning in the recent past. A couple of solutions have been proposed [11], [12], [13]. All of them stick with the expensive core concepts of the index tuning tools: what-if evaluation and creation and dropping of complete indexes. Consequently, they suffer from a large control loop delay.

Partial Indexes are a way to reduce the control loop delay. Smaller and more focused indexes are less expensive to create, to maintain, and to drop. Partial Indexing is a well understood concept [1], [2] and is available in major database systems, e.g., SQL Server [14]. A system called Shinobi builds a fine-grained online tuning approach on the idea of partial indexing [15]. However, Shinobi realizes the partial index by partitioning a table into interesting tuples and uninteresting tuples and indexing the partition of interesting tuples completely. In case an index is not hit, Shinobi can easily skip all indexed pages by just scanning the unindexed partition. The downside is that all indexes of the table index the same set of tuples. Although Shinobi is an appealing new approach to online index tuning, it utilizes the power of partial index only in very limited way. The Index Buffer allows page skipping without limiting the power of partial indexing.

VII. Conclusion

We presented the Adaptive Index Buffer. An Index Buffer is a scratch pad index. It helps to lower the query cost on partially indexed columns during times of workload changes, while the partial index is not adapted yet to the new workload. During table scans of queries that cannot use the partial index, the Index Buffer completes the indexing of pages so that these pages can be skipped in subsequent tables scans. Memory-based and without expenses for crash recovery, the Index Buffer can build up index information very quickly. To not exhaust memory resources, the Index Buffer management enforces an upper limit of total Index Buffer entries. The management is based on how often and how many page skips index information allows. Additionally, we also presented three experiments which show the behavior and effect of the Index Buffer. We are convinced that the Index Buffer is a useful puzzle piece to bring self-tuned adaptive partial indexing to life.

References

[1] M. Stonebraker, "The case for partial indexes," *SIGMOD Record*, vol. 18, no. 4, 1989.
[2] P. Seshadri and A. N. Swami, "Generalized partial indexes," in *ICDE'95*, 1995.
[3] D. Comer, "The ubiquitous b-tree," *ACM Computing Surveys*, vol. 11, no. 2, 1979.
[4] J. Rao and K. A. Ross, "Making b$^+$-trees cache conscious in main memory," in *SIGMOD'00*, 2000.
[5] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *SIGMOD'93*, 1993.
[6] T. Mueller, "H2 database," http://www.h2database.com.
[7] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala, "Database Tuning Advisor for Microsoft SQL Server 2005." in *VLDB'04*, 2004.
[8] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 Design Advisor: Integrated automatic physical database design." in *VLDB'04*, 2004.
[9] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin, "Automatic SQL tuning in Oracle 10g," in *VLDB'04*, 2004.
[10] S. Chaudhuri and V. R. Narasayya, "Autoadmin 'what-if' index analysis utility," in *SIGMOD'98*, 1998.
[11] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning." in *ICDE'07*, 2007.
[12] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "On-line index selection for shifting workloads," in *SMDB'07*, 2007.
[13] K.-U. Sattler, M. Luehring, K. Schmidt, and E. Schallehn, "Autonomous management of soft indexes," in *SMDB'07*, 2007.
[14] S. Acharya, P. Carlin, C. A. Galindo-Legaria, K. Kozielczyk, P. Terlecki, and P. Zabback, "Relational support for flexible schema scenarios," *The Proceedings of the VLDB Endowment*, vol. 1, no. 2, 2008.
[15] E. Wu and S. Madden, "Partitioning techniques for fine-grained indexing," in *ICDE'11*, 2011.