Hannes Voigt, Thomas Kissinger, Wolfgang Lehner

SMIX – Self-Managing Indexes for Dynamic Workloads

Erstveröffentlichung in / First published in:

SSDBM '13: Conference on Scientific and Statistical Database Management, Baltimore 29.07. – 31.07.2013. ACM Digital Library, Art. Nr. 24. ISBN 978-1-4503-1921-8

DOI: https://doi.org/10.1145/2484838.2484862

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

SMIX – Self-Managing Indexes for Dynamic Workloads

Hannes Voigt, Thomas Kissinger, Wolfgang Lehner
Database Technology Group
TU Dresden
Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

As databases accumulate growing amounts of data at an increasing rate, *adaptive indexing* becomes more and more important. At the same time, applications and their use get more agile and flexible, resulting in less steady and less predictable workload characteristics. Being inert and coarse-grained, state-of-the-art index tuning techniques become less useful in such environments. Especially the full-column indexing paradigm results in many indexed but never queried records and prohibitively high storage and maintenance costs. In this paper, we present *Self-Managing Indexes*, a novel, adaptive, fine-grained, autonomous indexing infrastructure. In its core, our approach builds on a novel access path that automatically collects useful index information, discards useless index information, and competes with its kind for resources to host its index information. Compared to existing technologies for adaptive indexing, we are able to dynamically grow and shrink our indexes, instead of incrementally enhancing the index granularity.

Categories and Subject Descriptors

H.2.2 [**Physical Design**]: Access methods; H.2.4 [**Systems**]: Relational databases

General Terms

Indexing, Access Paths

1. INTRODUCTION

Indexes are the most fundamental technique to speed up queries in database systems. Since indexes support only fractions of a database's workload while requiring resources to be stored and maintained, indexing poses an optimization problem. With changing data and shifting workloads, the optimum is a moving target. As a secondary data structure, indexes always constitute a trade-off between increased query performance on the one hand and storage resources and maintenance cost on the other hand. Index information that is beneficial today may be unprofitable tomorrow, while another index may have become very useful at the same time. Index optimization is no point-in-time decision, but remains a continuous effort. Self-managing indexing, where index optimization is an integral part of the database system, takes the burden of this effort permanently away from the user.

In relational databases, columns form the primary granularity of indexing; each index covers all values of one or more columns. As databases accumulate growing amounts of data at an increasing rate, the core problems of full column indexing become more evident. If the data in a column doubles over time, a full column index takes nearly twice the time to be created and consumes about twice the storage. At the same time the data of interest is unlikely to double in size as an increasing share of data is primarily kept for reasons of revision, lineage, and versioning.

For instance, consider the order processing workload of the TPC-C benchmark [16]. Typically, an order is only queried during its processing. After the order is processed, the product delivered and the bill paid, the order and its data remains in the system without being individually queried anymore. Another example are wikipedia article revisions. Many articles have hundreds of revisions [20], but only the small fraction of the most recent revisions is queried.

In short, the trend of growing data sizes has two consequences: (1) The traditional full-column index will soon encompass and maintain mainly unused index information. (2) The traditional index tuning based on creating and dropping indexes will soon become prohibitively expensive. Partial indexing breaks the trend by focus indexes on the hot data. An index not crowded with cold and uninteresting data remains lean and efficiently manageable.

Existing self-managed partial indexing approaches rely on data clustering. Either they cluster the data with the index order or into hot and cold data. Both concepts are limited for secondary indexes. With multiple indexes on one table, not every index necessarily has the same clustering. Consider OLAP star queries as an exemplary scenario. On some dimensions star queries select tuples, for instance a certain year, region or product group, which requires an index on the name of other descriptive columns of the dimension table. Other dimensions are later joined against the fact table for a comprehensive report, which requires an index on the primary key, usually a surrogate. Tuples that are hot in

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

the one case may be of no interest in the other; clustering becomes difficult.

In this paper, we propose a novel, adaptable, fine-grained, autonomous indexing infrastructure for secondary indexes in row stores. It is based on the Self-Managing Index (SMIX), a new access path that partially indexes the column it is working on. Like a table scan, a SMIX is available on every column by default. Like an index, it maintains access information in a secondary data structure for faster predicate evaluation. To remain lean, a SMIX constantly adapts the set of indexed tuples to the workload.

The partial index information helps to speed up queries in two ways. *Case 1:* If all tuples that fulfill the queried predicate are indexed, the tuples answering the query can be directly discovered with the index. This is the desired case, because a table scan can be avoided. The more queries fall into this category, the better the partial index is adapted to the workload. *Case 2:* If all tuples in a page are indexed, all tuples matching the queried predicate can be discovered by a table scan. This table scan is able to skip the fully indexed page, if it combines its result set with the scan of the partial index. Hence, skipping a lot of pages during a table scan helps to lower query execution costs in situations where the partial index is not well adapted to the workload and many table scans are necessary to answer queries.

To cover both cases, our SMIX access path completely indexes (1) the most queried values, to answer most of the common queries efficiently, and (2) a proportion of pages, to speed up table scans. This way, a SMIX is able to adapt to a workload while quickly leveraging collected index information.

As the SMIX is a default access path, it automatically collects index information on potentially every column. Two principles keep the SMIX population of a database from exceeding a configurable global resource limit. (1) Every SMIX has an individual resource quota and it is able to displace less queried index entries to lower its resource usage. (2) All SMIXs compete for the globally granted resources, so that invaluable index information automatically drops out of the system.

The SMIX indexing infrastructure comes with only a very few configuration knobs, mainly the amount of resources that can be used for indexing. SMIXs distribute the heavy lifting of index creation over time and focus index creation on the data of interest. Furthermore, the approach does not involve expensive what-if calls to the query optimizer. This way, SMIXs reduce the required user interaction dramatically without sacrificing performance by missing indexing opportunities or imposing to much overhead on the DBS.

The paper is structured as follows. In Section 2, we start with an introductory example to illustrate our approach. We present the approach in detail in the three sections following. Section 3 lays out the general SMIX architecture. Section 4 details the SMIX access path; how it collects, maintains, and displaces index information. Section 5 discusses the global management of all SMIX present in a system. We conducted experiments to evaluate our approach and present the results

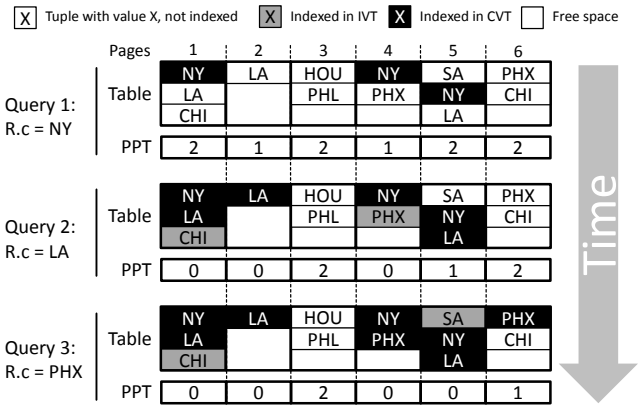


Figure 1: SMIX Indexing Example

in Section 6. Finally, we discuss related work in (Section 7) and conclude the paper (Section 8).

2. INTRODUCTORY EXAMPLE

Before outlining the details of the SMIX approach, we illustrate the general idea of the SMIX access path with an example. A SMIX consist of three data structures: two tree indexes – the *covered values tree (CVT)* and the *intermediate values tree (IVT)* – and a list of counters – the *page population table (PPT)*. We will detail on all three later.

For the example shown in Figure 1, we use a table for smartphone sales of the seven biggest cities in the US. Let this table consist of 13 tuples stored in six pages. At the beginning, the table was not queried before and is now hit by three consecutive queries on the same column for which the optimizer decides to use a SMIX scan. All three queries select a new value, not queried before. Before the first query, the SMIX is uninitialized and the data structures do not exist at all and will be created with the first query. The figure shows the state of the SMIX's data structures after every query.

For the first query on NY, the SMIX scans the complete table. While scanning the table, the SMIX inserts the three qualifying tuples into the CVT. During the first table scan, the SMIX neither skips any pages nor indexes pages into the IVT, instead it initializes all PPT counters with the number of not indexed tuples, remaining in each page.

For the second query on LA, three tuples qualify and are indexed into the CVT. Since none of the pages is fully indexed yet, no page can be skipped during the table scan. However, with the PPT counters now set, the SMIX can decide before the table scan for which pages it wants conduct to complete indexation; let it decide for page 1, 2 and 4. In consequence, the SMIX additionally indexes the two not qualifying tuples in these pages in the IVT. After the second SMIX scan, six tuples are indexed in the CVT, two tuples are indexed in the IVT, and the three pages 1, 2, and 4 are completely indexed. All PPT counter have been update accordingly.

For the third query on PHX, the SMIX again scans the table. This time, the table scan can skip the completely indexed pages 1, 2, and 4. In the remaining pages, the scan finds

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

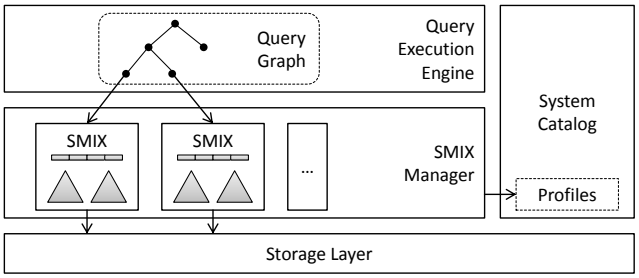


Figure 2: SMIX in Database Architecture

two qualifying tuples and indexes them in the CVT. Additionally, the SMIX decides to complete indexation for page 5, which has only one unindexed tuple left. The SMIX also scans the IVT, to check for qualifying tuples that may be missed during the table scan by skipping pages. Both resulting tuple streams – of the table scan and the IVT scan – are merged together to form the result of the SMIX scan. Since all tuples found by the IVT scan are qualifying tuples for the queried value, the SMIX removes them from the IVT and adds them to the CVT. In the example, this is the case for the second tuple in page 4. After the third SMIX scan, eight tuples are indexed in the CVT, two tuples are indexed in the IVT, and the four pages 1, 2, 4, and 5 are completely indexed.

As it can be seen in the example, skipping of pages does not result in false negatives. Only fully indexed pages are skipped. For fully indexed pages, all tuples are indexed either in the CVT or in the IVT. The CVT is always scanned in the first phase, the IVT is scanned in the second phase. In both cases, the tuples will be discovered.

3. ARCHITECTURE

The core idea of SMIX is a new default access path that adapts itself to the workload. This requires two novel components in the database architecture shown in Figure 2. The first component is the SMIX itself, the new self-managing access path. The second component is the SMIX manager, which supervises the SMIX population in the system.

A SMIX combines the abilities of traditional table scan and index scan in a single access path. Like a traditional table scan, a SMIX acts as a default built-in access path, which is available on every column and does not have to be created explicitly. Like a traditional index, a SMIX incorporates index information, which allows reducing page accesses for queries significantly. Implementation-wise, a SMIX even reuses the logic of these traditional access paths. A SMIX autonomously collects index information based on the tuples that are accessed by the workload. It directly leverages this collected information for the next accesses, even if these accesses relate to other tuples. Additionally, a SMIX not only collects new index information, a SMIX also discards index information that turns out to be less useful. Therefore, a SMIX adapts to the data workload and is also able to control its use of storage and memory resources.

SMIXs co-exist to traditional access paths in the system. The query optimizer still decides which access path to take for a specific query. It applies two general rules for the access path selection: (1) It always chooses a SMIX scan over a table scan, if the optimizer would take an covering index on this column, because a SMIX can quickly adapt to better performance. (2) It always chooses a traditional index scan over the SMIX scan if a single column index is present, because a SMIX rarely exceeds the performance of a traditional index scan and redundant index information should be avoided. If multicolumn indexes are present on the queried column, the optimizer relies on traditional statistics-based decision rules. In order to accomplish that, every SMIX maintains statistics about itself in the system catalog, similar to traditional index and table statistics.

SMIXs are query-driven; they are not created explicitly. A SMIX that was never accessed does not consume any space. Each indexable column has a catalog entry indicating if it has an initialized SMIX present. As the SMIX scan is a default access-path a query can utilize a SMIX scan on a column even if the column's SMIX has not been initialized. The first SMIX scan on a column will initialize the SMIX on that column.

The SMIX manager is the supervisor component for all SMIXs in the system. Since SMIXs are automatically created and allocate new storage and memory resource on their own, they need to be controlled, in order to not exceed the globally available resources. The SMIX manager collects access statistics for every SMIX. Based on these statistics, it defines resource quotas for SMIXs and enforces them.

The globally available resources for indexing are index spaces, where the index information is stored. For SMIXs, we distinguish two types of index spaces: (1) the *establishment space*, (2) the *evolution space*. The establishment space represents disk resources; it offers persistency and supports crash recovery. A SMIX stores its established indexing information here, which has proven value for the workload. In opposition, the evolution space represents faster main memory resources; it is transient and does not support crash recovery. A SMIX stores supporting structures that contain less valuable indexing information in the evolution space. Especially when a SMIX is merely adapted to the workload, it makes heavily use of these supporting structures. Hence, the main-memory-based evolution space allows a faster adaption at lower costs compared to disk. The SMIX manager assigns quotas for establishment space and evolution space to each SMIX, while the absolute size of establishment space and evolution space is configured by the DBA.

In the following Section 4 we dive into detail on the SMIX design, especially which data structures it uses and how it operates. Afterwards, Section 5 gives details on the SMIX manager, how quotas are determined and enforced.

4. SMIX

In this section we describe the functioning of the adaptive indexation logic of a single SMIX, which works on a single column of a relation. At first, we introduce the data structures that are utilized for the indexation process. Followed by the state model, which specifies the different modes of operation a SMIX can be in. Then we show how the data structures and the state model work together. Further, we

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

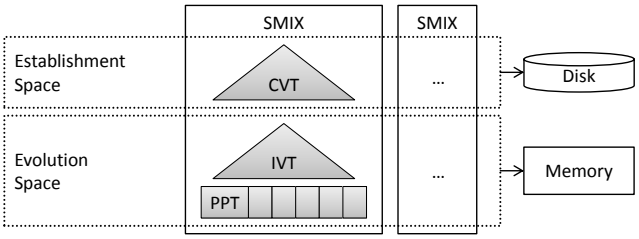


Figure 3: Data Structures of a SMIX

discuss how to reverse the indexation process, displace index information and free resources, which then can be claimed by other SMIXs. At last, we outline how SMIX are maintained during DML operations.

4.1 Data Structures

A SMIX consists of three data structures: the *covered values tree (CVT)*, the *intermediate values tree (IVT)*, and the *page population table (PPT)*. Depending on their usage, each of these structure is either stored inside the establishment space or the evolution space, as illustrated in Figure 3. The following describes the three data structures and their usage in more detail.

Covered Values Tree (CVT): The CVT is a conventional B*-Tree, which resides in the establishment space. The CVT completely indexes the most queried values. For values with qualifying tuples, the CVT holds references on all these qualifying tuples. For values without any qualifying tuples, the CVT holds a single null reference. Either way, the SMIX can serve queries on values present in CVT by a single CVT scan. The SMIX inserts values into the CVT when they are queried. To control its size, the SMIX also removes infrequently queried values from the CVT. By indexing only the most frequently queried values, the CVT reflects the current workload of the database – similar to a cache or a buffer. Thus, once adapted to the workload, the SMIX serves the majority of its queries by a single efficient CVT scan.

Intermediate Values Tree (IVT): The IVT is also a conventional B*-Tree and is stored inside the transient evolution space. The IVT completely indexes the remaining values of a page that are not already indexed by the CVT. This way, the IVT complements the CVT; all tuples of pages referenced in the IVT are indexed either in the CVT or in the IVT. (To avoid redundant indexing information, the tuple reference set of the CVT and the IVT are disjoint.) With help of the IVT, the SMIX can increase the number of completely indexed pages significantly. The SMIX can safely skip these completely indexed pages during a table scan without risking false negatives. All potential result tuples missed in the table scan can be discovered by a CVT or an IVT scan. While still adapting to the workload, the SMIX uses the IVT to speed up the table scans still required for a large share of queries. Thus, the IVT serves as a temporary supporting structure in times of workload changes. The IVT can be seen also as a specialized buffer pool, which buffers data in a processing-oriented form.

Page Population Table (PPT): The PPT is a list of counters located inside the evolution space. The list contains a counter for each page of the table the SMIX serves. Each PPT counter indicates how many tuples in its page are neither indexed by the CVT nor by the IVT. The PPT serves two proposes: (1) It helps to quickly identify pages that are most worthwhile to be indexed in the IVT (pages with a low counter greater than zero). (2) It allows to easily identify pages that can be skipped by a table scan (pages with a counter equal to zero). The SMIX initializes the PPT with the first table scan it has to perform. Subsequently, the SMIX maintains the PPT incrementally. The memory the PPT consumes depends on the accuracy a page is monitored and the number of pages.

Please note that conventional B*-Trees are not a necessity for the SMIX concept. A CVT or an IVT built as a hash index, a spatial index, the use of cache optimized index such as the CSB+-Trees [12] fits in equally well.

Table 1: State Characteristics of a SMIX

	Unstable	Stable
CVT hit rate	low	high
% index scans	low	high
% table scans	high	low
Need to adapt	high	low
IVT present	yes	no
PPT present	yes	no
CVT build up	yes	yes
Automatic displacement on CVT	no	yes

4.2 State Model

The operational mode of a SMIX depends on its need to adapt itself to the workload. A SMIX is well adapted to the workload if it can serve the majority with an efficient CVT scan, i.e., if its behavior resembles a standard index scan. Accordingly, we define the percentage of the recent queries that could be answered with a CVT scan as the CVT hit rate. If the CVT hit rate of a SMIX is low, the SMIX will operate in *unstable* mode and try to adapt to the workload. If the CVT hit rate is above a given threshold Th, the SMIX operates in *stable* mode. Depending on its CVT hit rate, a SMIX switches between both states. A SMIX measures its CVT hit rate h using a ring bitmap B with a configurable time frame t so that $h = |\{x|x \in B \wedge x = 1\}| \cdot t^{-1}$. Table 1 subsumes the most important characteristics of the two operational states.

Unstable State: In the unstable state, the SMIX has a low CVT hit rate, which results directly from the low number of SMIX accesses that are processed by a CVT scan. This means that the index needs to adapt to the current workload. The SMIX builds up and maintains the CVT and the additional IVT to temporarily speed-up the high number of table scans. In effect, the SMIX occupies resources in the establishment space as well as in the evolution space. In order to force the SMIX in a conceivable amount of time into the stable state, the SMIX desists displacing entries from the CVT automatically. (Nevertheless, the SMIX displaces CVT entries if it is forced to by the SMIX manager.)

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

Algorithm 1 SMIX Scan

```
1: procedure SmixScan(v)                          ▷ v: queried value
2:     Q ← CVT.Scan(v)                                  ▷ CVT scan
3:     if Q = ∅ then
4:         S ← SelectPagesForIVT(PPT)
5:         for t ∈ IVT do                               ▷ IVT scan
6:             if t.c = v then
7:                 Q ← Q ∪ {t}
8:                 IVT.Remove(t)
9:                 CVT.Add(t)
10:        for p ∈ R with PPT[p] > 0 do                 ▷ Table scan
11:            for t ∈ p do
12:                if t.c = v then
13:                    Q ← Q ∪ {t}
14:                    CVT.Add(t)
15:                    PPT[p] ← PPT[p] − 1
16:                else if p ∈ S ∧ t ∉ CVT then
17:                    IVT.Add(t)
18:                    PPT[p] ← PPT[p] − 1
19:    return Q
```

Stable State: The stable state is characterized by a high CVT hit rate, which means that the SMIX serves most queries with an efficient CVT scan. This implies a low probability of the need for expensive table scans. Therefore, the SMIX discards the IVT and the PPT and solely relies on the CVT. In effect, the SMIX occupies resources in the establishment space only. The SMIX further builds up the CVT, in case of unindexed values are queried. Additionally, it automatically displaces the most infrequent accessed CVT entries. Since these entries obviously do not fit the current workload anymore, they are not worth keeping. The goal of a SMIX is to get into the stable state, in order to serve the current workload most efficiently.

4.3 SMIX Scan

Traditionally, the database system accesses the requested data either via a full table scan or by scanning a fully covering index. Our approach combines both access paths in a single SMIX scan. During this SMIX scan, the CVT and the IVT gather indexing information, which is used to speed up subsequent SMIX scans.

A SMIX scan consists of two phases. In the first phase, the SMIX scans the CVT for the queried value. If the value is found in the CVT, the result of the CVT scan answers the query. In the second phase, the SMIX scans the IVT and the table for the queried value in case the CVT scan was negative. The table scan, though, performs three additional actions besides searching for tuples with the queried value. (1) The table scan skips all pages with a PPT counter equal to zero. (2) The table scan indexes all unindexed tuples of pages with the lowest PPT counter in the IVT. (3) The table scan indexes all qualifying tuples in the CVT. Note, the introductory example in Section 2 shows the second phase of a SMIX scan.

Algorithm 1 shows the SMIX scan in detail. Given a query $R.c = v$, the SMIX on column c of table R is scanned for value v. As the first step, the SMIX scans its CVT (line 2). If the CVT contains v, the SMIX scan is done and returns the result of the CVT scan. If the CVT result is empty instead, the SMIX performs a second step. Preparative, the SMIX determines the pages that should be fully indexed during this scan (line 4). Thereafter, the SMIX scans its IVT (line 5) and the table (line 10). IVT and table can be scanned in parallel. However, this may cause touching tuples twice, because the table scan adds tuples to the IVT. The IVT scan adds all qualifying tuples to the result and moves them to the CVT. The table scan also adds all qualifying tuples to the result and to the CVT. Additionally, if the scanned page was selected to be fully indexed, the table scan adds all not qualifying tuples that are not already indexed in the CVT to the IVT. For all tuples added to either of the indexes, the table scan decrements the PPT counter of the corresponding pages. Finally, the SMIX returns the result.

A SMIX scan operates in the unstable state as described. In the stable state, as detailed in Section 4.2, the SMIX performs only a table scan in case the CVT scan was negative nor does it scan or maintain an IVT. For range predicates the SMIX scan always has to perform a table scan, because the CVT may not cover the complete range. Hence, range predicates do not count as a CVT hit even if the CVT scan is positive.

The number of pages the SMIX indexes in the IVT in each table scan derives from the CVT hit rate. The lower the CVT hit rate is, the more table scans require speed up, the more pages we want to be completely indexed after the next table scan. Accordingly, the SMIX determines the number of pages as $\frac{h}{\theta}b$, where h is the SMIX's current CVT hit rate, θ is the configured threshold for the stable state and b is the total number of pages in the table. Note that this approach quickly increases the number of pages that can be skipped in the table scan; it has no influence on the improvement of the CVT hit rate nor does it facilitate reaching the stable state.

4.4 Displacement

While SMIXs grow incrementally, SMIXs are also able to shrink. Shrinking is crucial for the constant adaption to a shifting workload. By displacing index entries that are not worth keeping for the current workload, the SMIX frees storage and maintenance resources, which can be spent on index entries that are more valuable for the current workload. Displacement is triggered in two different ways: *forced displacement* and *automatic displacement*. The SMIX manager orders forced displacement to keep a SMIX within its resource quotas. Every time a SMIX index new data and requires more space, the SMIX manager checks the quotas and triggers a forced displacement is required. The SMIX itself performs automatic displacement to remove merely used index information. With every query a SMIX process in the stable state, the SMIX checks if some index information can be displaced. A SMIX implements different displacement strategies for its CVT and its IVT. Both strategies are detailed in the following.

CVT Displacement: The CVT contains the index entries most valuable to the current workload. It reflects the current workload and serves the majority of queries, once the SMIX is in the stable state. However, when the workload shifts, CVT entries created in an early workload episode

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

Table 2: SMIX Maintenance

		$t_{\text{old}} \in$ CVT		$t_{\text{old}} \notin$ CVT	
		$t_{\text{new}} \in CVT$	$t_{\text{new}} \notin$ CVT	$t_{\text{new}} \in CVT$	$t_{\text{new}} \notin$ CVT
p_{old}	p_{new}	CVT.Update(t_{old}, t_{new})	CVT.Remove(t_{old})	CVT.Add(t_{new})	-
$\in$ IVT	$\in$ IVT	-	IVT.Add(t_{new})	IVT.Remove(t_{old})	IVT.Update(t_{old}, t_{new})
	$\notin$ IVT	-	PPT[p_{new}]++	IVT.Remove(t_{old})	IVT.Remove(t_{old}), PPT[p_{new}]++
$\notin$ IVT	$\in$ IVT	-	IVT.Add(t_{new})	PPT[p_{old}]−−	IVT.Add(t_{new}), PPT[p_{old}]−−
	$\notin$ IVT	-	PPT[p_{new}]++	PPT[p_{old}]−−	PPT[p_{old}]−−, PPT[p_{new}]++

may not be valuable to the current workload episode. To displace these entries from the CVT in efficient way, the SMIX can remove the least frequently accessed leaf nodes from the CVT. Displacing a leaf node removes the range of values from the CVT, which have their entries in that node. This may include also valuable entries, but the overhead of tracking the value of single entries would be prohibitively high and valuable entries will return soon.

More specifically, the CVT displaces a leaf node in five steps. (1) It selects the leaf node that should be displaced. (2) It removes the referencing entry from leaf nodes parent node. (3) It frees the page the leaf node was stored in. (4) It removes overlapping entries from the neighboring leaf nodes. (5) It increments PPT counters accordingly for every page that is referenced in the leaf node and the overlapping entries.

The crucial step is the selection of the leaf node to displace. To displace the leaf node that is least worth to the current workload, the SMIX selects the node least recently used by a query. For that propose the SMIX maintains the *historic mean access interval* (Δ) and the *current mean access interval* ($\hat{\Delta}$) for all leaf nodes of its CVT. The higher both measures, the less a node was recently used. Both measures are explained in detail in Section 5.1. For forced displacement, the SMIX removes the leaf nodes with the highest $\hat{\Delta}$ values. For automatic displacement, the SMIX removes leaf nodes whose $\hat{\Delta}$ exceed its Δ by a factor D: $\hat{\Delta}/\Delta > D$. The factor D controls the aggressiveness of automatic displacement on CVTs; the lower D, the more aggressively the SMIX displaces leaf nodes.

IVT Displacement: The IVT is a supporting structure held in memory, which helps a SMIX during time of adaption. More specifically, its index entries are gap fillers to complement the entries of the CVT so that the pages are fully indexed and can be skipped in a table scan. Displacing a whole node of IVT entries would cause many pages not to be completely indexed anymore, which would thwart the purpose of the IVT without freeing many resources. Thus a reasonable fine grained displacement is not possible for the IVT. In consequence, the SMIX simply discards the whole IVT. Displacing the IVT does not hurt the stability of a SMIX, it merely slows the SMIX on the next table scans, because it can skip less pages. A SMIX displaces its IVT if it is ordered to do so by the SMIX manager (forced displacement) or if it enters the stable state (automatic displacement), since a SMIX does not maintain an IVT in the stable state (see Section 4.2 for details).

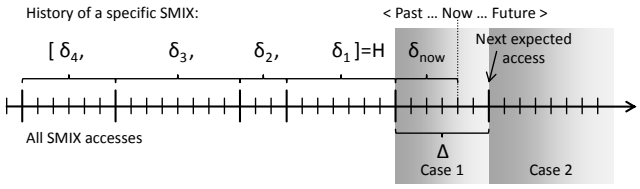


Figure 4: Access Interval Measures

4.5 Maintenance

The SMIX maintains the CVT, IVT, and PPT during inserts, updates, and deletes. Which operation the SMIX has to perform depends (1) if the old tuple t_{old} was in the CVT, (2) if the updated tuple t_{new} will be in the CVT, (3) if the old page p_{old} that contained the tuple is covered by the IVT, and (4) if the new page p_{new} that will contain the new tuple is covered by the IVT. Table 2 lists the different maintenance scenarios with necessary operations.

5. SMIX MANAGER

With every individual SMIX, the SMIX manager takes the task of supervising the whole population of SMIXs present in a system. The main goal of the SMIX manager is to ensure that the globally granted resources for SMIXs are not exceeded. The SMIX manager determines resource quotas for every SMIX and enforces these quotas.

5.1 Resource Quotas

Resource quotas define how much storage and memory every SMIX can occupy in the establishment space and the evolution space, respectively. Depending on its state, a SMIX gets a share of both, storage and memory (unstable) or only for storage (stable). For each SMIX, the SMIX manager continuously calculates the shares. The shares of a SMIX reflect (1) how often the SMIX is used and (2) the size of the column the SMIX indexes.

We determine the usage of a SMIX with the same measures used for the displacement in a CVT: the *historic mean access interval* Δ and the *current mean access interval* $\hat{\Delta}$. For both, consider Figure 4. Looking at a history of all accesses to all SMIXs in a system, we can determine how frequently a particular SMIX is used by averaging the length of the intervals between accesses to this SMIX. For that, the SMIX manager maintains a history $H = [\delta_1, \ldots, \delta_n]$ of the recent access interval lengths δ_i for each SMIX, so that δ_1 presents the most recent access interval and n is the maximal history length maintained. Additionally, every SMIX has a counter δ_{now}, which the SMIX manager increases with every access

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

to any of the other SMIXs. If a SMIX is accessed, its δ_{now} becomes the new first entry of the history, so that $H = [\delta_{now}, \delta_1, \ldots, \delta_{n-1}]$. The historic mean access interval Δ of a SMIX is the mean of its history, $\Delta = (\delta_1 + \ldots + \delta_n) \cdot n^{-1}$. The smaller Δ, the more frequent a SMIX is accessed. In the example shown in the figure, the considered SMIX has a history of $[7, 3, 8, 6]$, $\delta_{now} = 4$ and $\Delta = 6$.

The historic mean access interval Δ already gives a good measure how frequently a SMIX is used. However, Δ only reflects the past, i.e., accesses that happened. If a SMIX is not accessed anymore because of a workload change, it will keep a small Δ. To consider also a SMIX's current interval between the last access and the next access to come, we distinguish two cases: Either the expected point for the next access (1) is still ahead ($\delta_{now} \leq \Delta$) or (2) has already past ($\delta_{now} > \Delta$). In the first case, we stick with the expected interval and assume the current interval to equal Δ. In the second case, we know the expectation is wrong and the current interval is at least as long as δ_{now}. The current mean access interval $\hat{\Delta}$ of a SMIX is the mean of its history including the current interval:

$$\hat{\Delta} = \begin{cases} \Delta & \delta_{now} \leq \Delta \\ \frac{\delta_{now} + \delta_1 + \ldots + \delta_n}{n+1} & \delta_{now} > \Delta \end{cases}$$

The size s of the SMIX indexed column calculates from the number of tuples k in the corresponding table and the column width l: $s = kl$. A SMIX is worth higher shares than other SMIXs if it is accessed more frequently and if it has to cover a larger amount of data compared to other SMIXs. Hence, in a SMIX population P, the share of a SMIX is

$$\omega = \frac{\hat{\Delta}^{-1} s}{\sum_P \hat{\Delta}^{-1} s} \ .$$

The population P encompasses all SMIXs for the establishment space and all unstable SMIXs for the evolution space, resulting in two shares ω_{est} and ω_{evo}, respectively. Based on these two shares, the SMIX manager grants each SMIX a number of pages in the establishment space and in the evolution space as resource quotas.

5.2 Quota Enforcement

With the help of the quotas, the SMIX manager determines which SMIX can grow and which SMIX has to shrink. Because of the different displacement characteristics, the enforcement of the individual quotas for establishment space and evolution space differs, too. In the following, we describe the enforcement for both spaces in detail.

Quota Enforcement in Establishment Space: The establishment space hosts CVTs only. CVTs grow and shrink moderately, which allows an optimistic enforcement of the quotas. Requests for new CVT pages are always granted to a SMIX regardless of its quota. If the available establishment space is exhausted, the SMIX manager will order forced single-page displacements until the required number of pages is free. For every displacement, the SMIX manager selects a SMIX by two rules: (1) If SMIXs exceed their quota, the SMIX manager will pick the SMIX with the largest relative excess. (2) If multiple SMIX have same relative excess, the SMIX manager will pick the least recently used SMIX among them (highest $\hat{\Delta}$). The selected SMIX decides which specific page it will displace (see Section 4.4).

Quota Enforcement in Evolution Space: The evolution space hosts unstable SMIXs, specifically PPTs and IVTs. Both account for the quota of a SMIX. PPTs change their size only in case of inserts to the table and are only displaced, if the SMIX reaches the stable state. In case the quota of a SMIX is too low to fit its PPT, the SMIX manager removes the IVT and PPT completely. However, the manager keeps determining the SMIX's quota, so that the SMIX may be allowed to initialize again later. In contrast to PPTs, IVTs change their size rapidly. On the one hand, an IVT grows rapidly, if the SMIX is below its quota in the evolution space and completes indexing for many pages in each table scan. On the other hand, an IVT shrinks suddenly to zero size, if the SMIX manager forces its displacement. In consequence, the SMIX manager enforces the evolution space quotas pessimistically to avoid excessive displacement. Requests for new IVT pages are granted to a SMIX as long as the SMIX does not exceed its quota. This strategy prevents a SMIX from actively exceeding its quota. However, a SMIX can passively exceed its quota in case the quota changes. After each recalculation of the evolution space quotas, the SMIX manager determines which SMIXs exceed their quota and orders a forced displacement on one of them. To select the SMIX that will have to displace its IVT, the SMIX manager applies the same two rules as in the establishment space (largest relative excess and least recently used).

6. EVALUATION

To evaluate our SMIXs approach, we conducted a series of experiments. We start giving an overview of our prototype and describe the setup that we use for our experiments. Afterwards, we present a basic experiment to illustrate the behavior and the inner mechanics of a SMIX, especially its ability to adapt to a changing workload. Based on that experiment, we investigate the influence of the different SMIX parameters. We continue by evaluating a set of common workload patterns for a single SMIX. Finally, we extend this to a more complex scenario, which involves multiple SMIXs.

6.1 Experimental Setup

We implemented our prototype of SMIX in PostgreSQL 9.0.2. In the prototype, SMIXs are the new default access path available by default on every column. Implementation-wise, our prototype reuses the existing heap scan code and B+-tree code of PostgreSQL as much as possible. We ran all experiments on an Intel Core i7-2600 processor at 3.4 GHz with 8 GB of DDR3 main memory and a 1TB Samsung hard drive at 7200 rpm. We used Microsoft Windows 7 64bit edition as the operating system.

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. Think of the table as a dimension table in an OLAP scenario as described in the introduction. The integer columns are the column queried, either to slice the dimension or to join it with the already sliced fact table. The payload data represent descriptive properties of the dimension that are not part of selection predicates. All three integer columns are populated with random values uniformly

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

Table 3: SMIX Base Configuration

Parameter	Value
Size of establishment space	64MB
Size of evolution space	128MB
Threshold θ	95%
Time frame t	200
Displacement factor D	∞
History length n	3

distributed from 1 to 50 000. The size of the payload values is also uniformly distributed from 1 to 512. We filled the table with 5 000 000 tuples, resulting in an effective table size of 1.5 GB on disk. In the base configuration, the database system is configured to use 256 MB of shared buffers. Furthermore, the base configuration encompass an initial setting for all the parameters listed in Table 3. The base configuration applies for all experiments, unless stated otherwise.

6.2 General Performance

Our initial experiment illustrates the general behavior and performance of a single SMIX and how it adapts to a changing workload. In the experiment we run a workload consisting of two subsequent episodes. Each workload episode is a set of 5 000 queries in the form of **SELECT COUNT(*) FROM R WHERE c=**x. For every query, we pick x randomly from a equally distributed continuous range of the domain of column **c**. In the first episode $x \in [1\,000, 2\,000]$ and in the second episode $x \in [25\,000, 26\,000]$. We used the base configuration, except the size of the evolution space was reduced to 64 MB. For all 10 000 queries, we measured the execution time, the CVT hit rate, the CVT size, and the IVT size. We compare our measurements with two baselines: the traditional table scan and the traditional full-column index.

Figure 5(a) shows the execution time and the CVT hit rate over the course of the workload. The traditional table scan and the traditional full-column index exhibit a constant execution time over the complete workload of about 1 200 ms and 1 ms, respectively. The execution time remains constant, since the two traditional access paths treat every equally and do not adapt to the workload. In contrast, the execution time the SMIX varies over the course of the workload. The figure shows the exact execution time of each query (thin blue line without markers) and the sliding average of 100 queries (green line with circle markers). With the first query, the SMIX initializes its PPT and indexes the first value in the CVT. With the second query, the SMIX indexes about half of the table into the main memory-based IVT because it has a low CVT rate and plenty of evolution space available in the beginning. In consequence, the SMIX execution times of the first two queries (1 250 ms and 4 880 ms, respectively) exceed the traditional table scan. From the third query on, the execution time of the SMIX is significantly lower than the table scan. When the SMIX has to perform a table scan, it takes about 650 ms to answer the query because the SMIX, still in its unstable state, can leverage the IVT and skip pages during the table scan. Otherwise, if the query hits the CVT, the execution time is below 1 ms and therefore comparable to an traditional index. Over the course of the workload, the SMIX collects an increasing share of the queried values in the CVT. Consequently, the CVT hit rate increases and the average execution time drops quickly below 100 ms.

With query 3 196, the CVT hit rate exceeds the threshold θ and the SMIX changes into the stable state. Within the process, the SMIX discards its IVT and PPT. Without its two helping structures now, the SMIX cannot skip pages to speed up the table scan. Consequently, the SMIX execution time in case of table scan jumps to 1 200 ms. Already well adapted to the current workload, though, the SMIX can answer the majority of queries with a CVT scan. The average execution time stays at the low level and drops even further below 50 ms as the SMIX keeps indexing new values into the CVT. As well visible in the figure, the number of spikes indicating that the SMIX has to perform a table scan decrease further towards query 5 000.

With query 5 000, the workload switches to the next episode. None of the values indexed in the SMIX's CVT is queried anymore. The CVT hit rate drops instantly below the threshold and the SMIX changes into the unstable state again. Within a few queries, the SMIX rebuilds its IVT. The adaption process repeats.

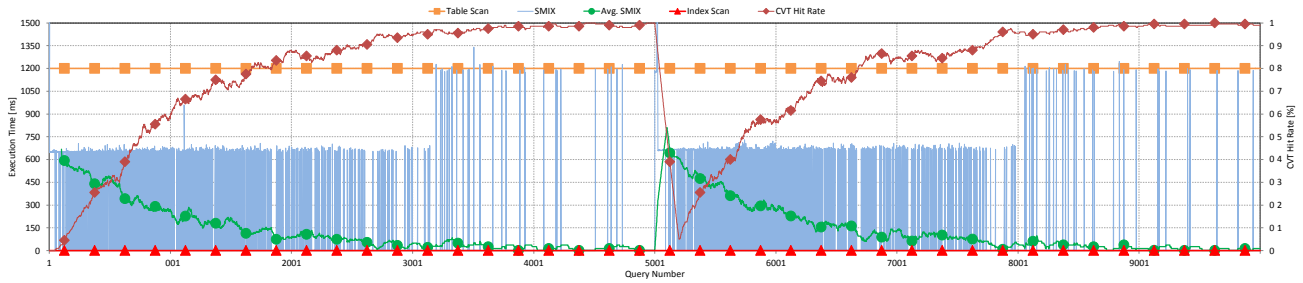
Figure 5(b) shows the cumulative execution time for the SMIX and the two traditional access paths in comparison. As can be seen, the SMIX quickly becomes profitable. With the 10th query, its cumulative execution time falls below the table scan. The bump at query 5 000 reflects the costs of re-adaption. As expected, the traditional full-column index requires less execution time – its creation not included. In our experimental setting a full-column index on column **a** requires approximately 107 MB disk space. In comparison, the CVT consumes about 3 MB after the first workload episode and 6 MB after the second episode, as illustrated in Figure 5(c). In total, the CVT consumes about 10% of the configured 64 MB available establishment space. Figure 5(c) shows the amount of memory consumed by the IVT over the course of the workload. Following its nature as an intermediate supporting structure, the IVT consumes all the available evolution space of 64 MB while the SMIX in the unstable state. In this experiment, the SMIX shows a good adaption behavior and proves its ability quickly detect workload changes.

6.3 Parameter Impact

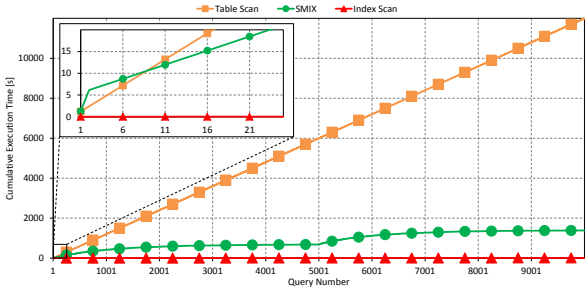
In the next step we want to investigate the impact of three important SMIX parameters: the *size of the evolution space*, the *stability threshold* θ and the *CVT displacement aggressiveness factor* D All three parameters have influence on the overall adaption performance. We used the base SMIX configuration (Table 3) and the same workload episodes as in the previous experiment. In the following we describe the effects of different settings for each single parameter.

Size of Evolution Space: Figure 6(a) shows the execution times (smoothed over a period of 100 queries) for three different sizes of the evolution space. We configured the evolution space with 16 MB, 64 MB, and 128 MB. Consider that the IVT would take up to 140 MB, if it would index all remaining tuples of the relation. Because the evolution space is only utilized if a SMIX is in an unstable state, we see big

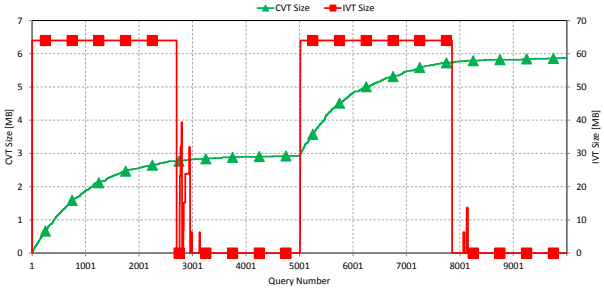
Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862



(a) Execution Time and CVT Hit Rate



(b) Cumulative Execution Time



(c) Size of CVT and IVT

Figure 5: General Performance

differences in the early adaption stage. The 128 MB configuration shows the best performance for the first queries of an episode because the IVT is able to index many of the remaining tuple quickly, which allows all subsequent table scans to skip almost every page. At this point, the SMIX operates close to the speed of an index scan. The opposite is visible at the 16 MB setting. Here, the IVT is able to index merely 1% of the remaining tuples. This results in a much longer execution time in the early adaption stage. The often necessary table scans are not able to skip a large part of the pages. The larger the evolution space, the better the adaption behavior. Nevertheless, memory is a costly ressource and the evolution space size should be set carefully.

Stability Threshold θ: In Figure 6(b) we visualized the smoothed execution times for various settings of the stability threshold θ. We ran the experiments for a threshold of 50%, 75% and 95%. For this experiment, we see the differences as soon as the SMIX enters the stable state. The main observation is that a low threshold of 50% leads to higher execution times in the early stable stage. This poor performance happens because only 50% of the queried values hit the CVT. Thus, the SMIX needs to invoke a table scan for the other 50% of the queries, which is not able to skip any pages since the IVT supporting structure was dropped, in the moment, the SMIX entered the stable state. In our experiments, a threshold value of 95% turned out to be the best solution.

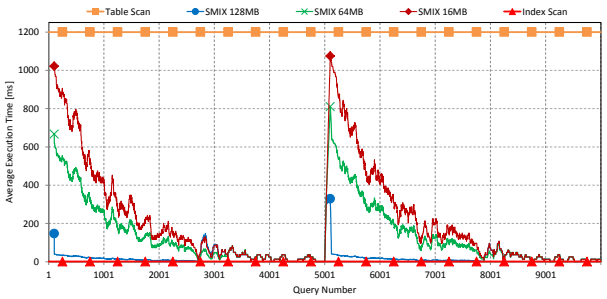
CVT Displacement Aggressiveness D: In this experiment series we investigate the influence of the automatic displacement, which was disabled in all previous experiments for reasons of simplicity. Because automatic displacement is only allowed in the stable state, we are going to see the differences only in the stable passages of the experiments. Figure 6(c) shows the CVT sizes of the SMIX for the settings of 5, 10, and 15. A low D means a high aggressiveness. With D set to 15, we observe a slow displacement of unused CVT entries at the end of the second episode. For settings of 10 and 5, the experiment shows a much faster displacement in the second episode. However, the more aggressive displacement also leads to displacements in the first episode, where it effects CVT entries that are used by the workload. Thus, as a rule of thumb we recommend a D of 15 because it is not critical to perform an automatic displacement of stale index information as soon as possible.

6.4 Workload Patterns

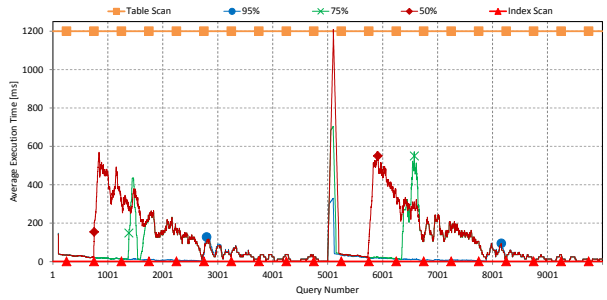
In this subsection, we analyze the adaption process for a set of different common workload types. We start with a workload that extends the range of queried values at a single blow. The next workload moves its value range slowly to another position. And finally we investigate a workload that queries a set of scattered values.

Widening Workload: In the first episode, this workload queries a continuous range of $x \in [1\,000, 2\,000]$. The following episode doubles this range to $x \in [1\,000, 3\,000]$ and still includes the range of the first episode. Both episodes execute 5 000 queries. Figure 7(a) shows the measured execution times and the corresponding CVT hit rate. After the first episode, the SMIX is well adapted to the workload. As soon as the workload extension happens, the CVT hit rate drops below the threshold and ends near 50% because the CVT already indexed half of the value range during the first episode. This falling CVT hit rate puts the SMIX into the unstable state, where it uses an IVT to boost the necessary table scans. After query number 9 282, the CVT hit rate passed the Threshold and the SMIX reenters the stable state and the adaption to the workload extension finished.

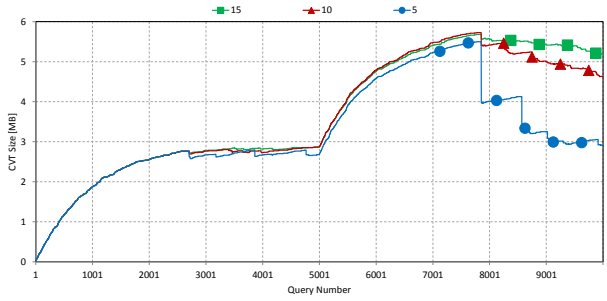
Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862



(a) Execution Times for different Evolution Space Sizes

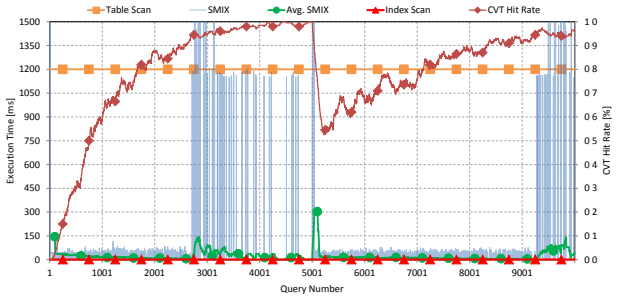


(b) Execution Times for different CVT Hit Rates

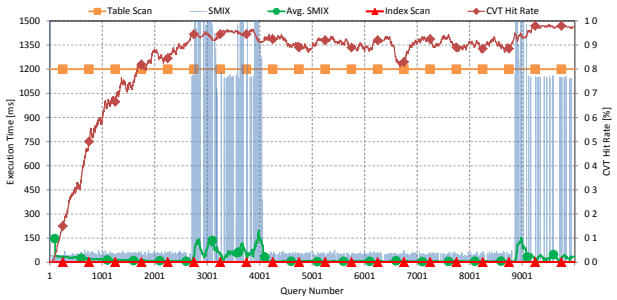


(c) CVT Sizes for different Settings of D

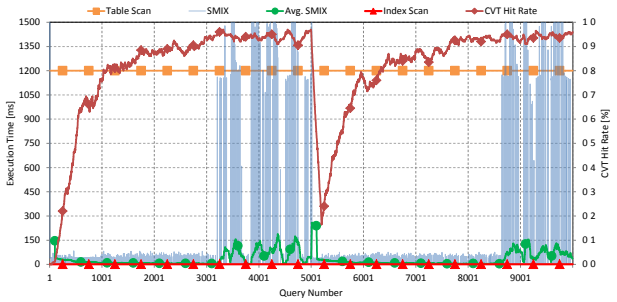
Figure 6: Parameter Impact



(a) Widening Workload



(b) Shifting Workload



(c) Scattered Workload

Figure 7: Execution Time and CVT Hit Rate on Different Workload Patterns

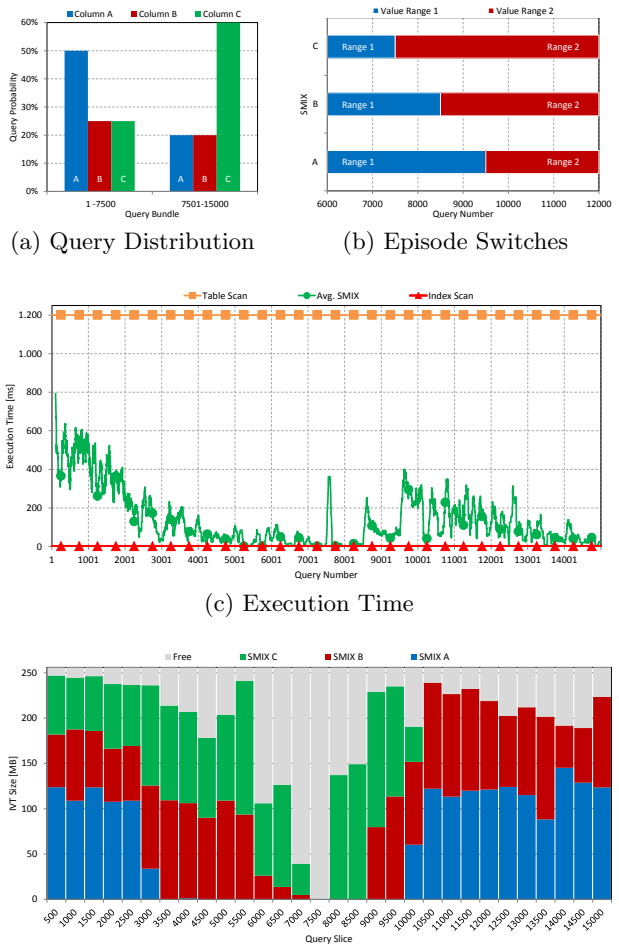
Shifting Workload: This workload consists of three episodes. (1) 3 000 queries on a continuous range of $x \in [1\,000, 2\,000]$. (2) 6 000 queries on a continuous range that slides linearly from $x \in [1\,000, 2\,000]$ to $x \in [1\,500, 2\,500]$. (3) 1 000 queries on the final range of the previous episode. Figure 7(b) visualizes execution times and CVT hit rates for this workload. At the end of the first episode, the SMIX is well adapted. With the beginning of the next episode, the queried range starts to move slowly. It takes about 1 000 queries for the SMIX to detect this slow workload change. After this detection period, the SMIX rebuilds an IVT and actively supports the adaption process. Once the workload shifting is done at the end of the second episode, the SMIX is back in the stable state.

Scattered Workload: In a lot of cases, queries are executed on scattered values rather than continuous ranges. Again, the workload consists of two episodes. Each episode consists of 5 000 queries on randomly preselected values. Figure 7(c) shows the experimental results. Compared to previous experiments on continuous ranges, we observe a faster rising CVT hit rate, but no other visible difference to a workload on a continuous value range.

6.5 Complex Scenario

Finally, we conclude our evaluation by using a more complex workload that involves SMIXs on the columns a, b, and c of the table R. The workload executes a total of 15 000 queries and each query has a given probability to hit one of the three columns that changes after 7 500 queries. These probabilities are shown in Figure 8(a). Furthermore, a query on a specific column addresses a value of a uniformly distributed range of 500 continuous values. We use two disjunct value ranges, where each SMIX starts with the first value range and after a specific number of queries, the SMIX switches over to the second value range. This mapping is visualized in Figure 8(b). E.g., query number 8 000 has a probability of 60% to hit $SMIX_C$ and when that occurs, x is a random value in the first value range. For the experiment, we changed the size of the evolution space to 256 MB. The execution times (smoothed over a period of 100 queries) are visible in Figure 8(c) and Figure 8(d) shows the corresponding sizes of the IVT for each SMIX. At the beginning, all SMIXs start in the

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862



(a) Query Distribution

(b) Episode Switches

(c) Execution Time

(d) Evolution Space Occupancy

Figure 8: Complex Scenario

unstable state. For that reason, every SMIX builds up an IVT to speed up table scans. Because of the limited evolution space, all three SMIXs have to compete for the available memory. Therefore, the SMIX manager assigns an evolution space share to each SMIX. This share mainly depends on the access frequency of a SMIX. Consequently, $SMIX_A$ receives twice as much evolution space than $SMIX_B$ and $SMIX_C$. After query number 2 725 $SMIX_A$ is mostly in a stable state and misses its need for an IVT and is not involved in the evolution space distribution anymore. Thus, $SMIX_B$ and $SMIX_C$ are able to grow and speed up their table scans until all SMIXs are finally stable after query number 6 899. With query 7 501, the column access distribution changes and the value range of column c changes. Therefore, this SMIX builds up its IVT again, but has not to compete with other SMIXs since all of them are currently in a stable state. 1 000 queries later, $SMIX_B$ also changes its value range and starts the competition with $SMIX_C$. And finally, $SMIX_A$ joins this competition too, but, because $SMIX_C$ started earlier and was queried more often, it already entered the stable state. Thus, only $SMIX_A$ and $SMIX_B$ are left to require evolution space until they also reach the stable state.

7. RELATED WORK

Nowadays commercial database management systems offer index tuning tools, [1, 22, 4], which recommend an index configuration for a given workload and a storage bound the configuration has to fit into. However, all these state-of-the-art tools consider the database workload as static and predictable. Other approaches extend the idea of the index tuning tool to dynamic workloads. Here, the tool analyses the workload as a series of events over time and recommends a series of index configurations [2, 19]. Since all index tuning tools work offline, they do not add any extra load to the processing of the regular workload. Adversely, the user has to be able to predict the regular workload. If the workload changes unpredictable, the user has to notice this change and rerun the tool. With frequent shifts in the database workload, this get very inconvenient. Hence, research concentrated on autonomous index tuning in the recent past. A couple of solutions have been proposed [3, 14, 13]. All of them stick with the core concepts of the index tuning tools: full-column indexing and what-if evaluation. Consequently, they suffer from the same two drawbacks: (1) The index tuning remains very coarse-grained and the resulting indexes are likely to include a lot of data that is not of interest. (2) The index tuning requires expensive creation and dropping of complete indexes. Although SMIXs are also an autonomous index tuning approach, they are fundamentally different. SMIXs inherently index only data of interest and build on less expensive incremental index creation and adaption.

Other approaches aim in the same direction. Partial indexing [17, 15, 21] breaks with the paradigm of full-column indexing, too. In essence, the idea is to partition tables into interesting tuples and uninteresting tuples and index only the partition of interesting tuples. The idea is appealing, since it avoids effectively the unnecessary indexing of data in a very simple way. However, creating such a partitioning is not necessarily possible for two reasons: (1) A tuple may not be equally interesting for each index that indexes the tuple. For instance, cheap products may be often queried by price, whereas specifically advertised products may be often queried by their name. (2) The interestingness of a tuple regarding a specific index may not be a function of the attribute that the index is defined on. In the same example, the name of a product does not define the interest in the product, it is the advertisement. SMIXs do not exhibit these problems because they do not rely on partitioning. The interestingness of tuples can be randomly distributed over the value range of a column. Furthermore, every column of a table can show a different distribution of interestingness. In any case, SMIXs perform equally well.

Approaches based on incremental partitioning of unsorted data [10, 8, 7] (also known as database cracking), incremental merging of pre-sorted data chunks [6, 5] and combinations of both [9] specifically remedy the creation costs of indexes. Both concepts piggyback on queries to create index information on the tuples that are requested; this lowers the creation costs and distributes the effort over time. Although approaches are appealing, they do not represent solutions to the index optimization problem. Without any dropping of index information, every incremental index creation converges to a regular full index so that also unin-

Final edited form was published in "SSDBM '13: Conference on Scientific and Statistical Database Management. Baltimore 2013", Art. Nr. 24, ISBN 978-1-4503-1921-8
https://doi.org/10.1145/2484838.2484862

teresting tuples will be indexed at some point. Our SMIX approach is more comprehensive. SMIXs do not only involve incremental collection of index information, but also incremental displacement. Instead of converging to full indexation, SMIXs converge to the workload.

Complementary to this paper, we already presented a SMIX demo in [11] and a more general version of the IVT concept in [18].

8. CONCLUSION

In this paper, we presented Self-Managing Indexes, a novel, adaptable, fine-grained, autonomous indexing infrastructure. It is built on a novel default access path, called SMIX, which combines the traditional table scan and with index structures. A SMIX exhibits two key features: First, it automatically indexes the most queried tuples completely and additionally completes indexing of pages during periods of workload adaption to lower the cost of necessary table scans. Second, it is able to discard entries again if they become less useful. With these two features, a SMIX can adapt to the database workload and control its resource usage at the same time. All SMIXs within a SMIX population compete for resources; the most frequently used SMIX gets the most resources. The SMIX manager component supervises this competition by continuously determining resource quotas for every SMIX depending on its usage. In a series of experiments, we evaluated how SMIXs operate and perform. SMIXs showed significantly better performance than traditional scans. In periods of constant workload, SMIXs reach the same performance plateau as traditional indexes, while requiring less storage resource if the workload is focused. During periods of shifting workload, SMIXs consume additional memory resources to boost query performance.

We strongly believe the SMIX approach points into a new direction of how we can build the autonomous indexing infrastructures of the future, to lower the total cost of indexing as workloads become more dynamic and the amount of data to manage increases rapidly. In our ongoing work on the topic, we will investigate how we can lower the granularity of collecting and discarding index information even more. Additionally, we plan to leverage the optimizer cost model to improve the SMIX competition and add support for range queries and joins. We want to improve the support for range predicate, so that a SMIX can avoid a table scan in case the CVT hits the complete range. Finally, we will further investigate in which way our approach interacts with other approaches of autonomous indexing approach and how ideas can be combined to make autonomous indexing reality.

9. ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 “Highly Adaptive Energy-Efficient Computing”.

10. REFERENCES

[1] 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.

[2] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic Physical Database Tuning: Workload as a Sequence. In *SIGMOD'06*, 2006.

[3] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE'07*, 2007.

[4] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL Tuning in Oracle 10g. In *VLDB'04*, 2004.

[5] G. Graefe and H. A. Kuno. Adaptive indexing for relational keys. In *ICDEW'10*, 2010.

[6] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT'10*, volume 426, 2010.

[7] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *The Proceedings of the VLDB Endowment*, 5(6), 2012.

[8] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR'07*, 2007.

[9] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *The Proceedings of the VLDB Endowment*, 4(9), 2011.

[10] M. L. Kersten and S. Manegold. Cracking the Database Store. In *CIDR'05*, 2005.

[11] T. Kissinger, H. Voigt, and W. Lehner. SMIX Live – A Self-Managing Index Infrastructure for Dynamic Workloads. In *ICDE'12*, 2012.

[12] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *SIGMOD'00*, 2000.

[13] K.-U. Sattler, M. Luehring, K. Schmidt, and E. Schallehn. Autonomous Management of Soft Indexes. In *SMDB'07*, 2007.

[14] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-Line Index Selection for Shifting Workloads. In *SMDB'07*, 2007.

[15] P. Seshadri and A. N. Swami. Generalized Partial Indexes. In *ICDE'95*, 1995.

[16] K. Shanley. TPC Releases New Benchmark: TPC-C. *SIGMETRICS Performance Evaluation Review*, 20(2), 1992.

[17] M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4), 1989.

[18] H. Voigt, T. Jäkel, T. Kissinger, and W. Lehner. Adaptive Index Buffer. In *SMDB'12*, 2012.

[19] H. Voigt, W. Lehner, and K. Salem. Constrained Dynamic Physical Database Design. In *SMDB'08*, 2008.

[20] Wikipedia. Wikipedia:Pruning article revisions, July 2012. `http://en.wikipedia.org/wiki/Wikipedia:Pruning_article_revisions`.

[21] E. Wu and S. Madden. Partitioning Techniques for Fine-grained Indexing. In *ICDE'11*, 2011.

[22] 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.