Online Reorganization in Read Optimized MMDBS

Felix Beier
Ilmenau University of Technology
Langewiesener Str. 37
D-98693 Ilmenau, Germany
felix.beier@tu-ilmenau.de

Knut Stolze
IBM Research & Development
Schönaicher Str. 220
D-71032 Böblingen, Germany
stolze@de.ibm.com

Kai-Uwe Sattler
Ilmenau University of Technology
Langewiesener Str. 37
D-98693 Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Query performance is a critical factor in modern business intelligence and data warehouse systems. An increasing number of companies uses detailed analyses for conducting daily business and supporting management decisions. Thus, several techniques have been developed for achieving near realtime response times - techniques which try to alleviate I/O bottlenecks while increasing the throughputs of available processing units, i.e. by keeping relevant data in compressed main-memory data structures and exploiting the read-only characteristics of analytical workloads.

However, update processing and skews in data distribution result in degenerations in these densely packed and highly compressed data structures affecting the memory efficiency and query performance negatively. Reorganization tasks can repair these data structures, but – since these are usually costly operations – require a well-considered decision which of several possible strategies should be processed and when, in order to reduce system downtimes.

In this paper, we address these problems by presenting an approach for online reorganization in main-memory database systems (MMDBS). Based on a discussion of necessary reorganization strategies in IBM Smart Analytics Optimizer, a read optimized parallel MMDBS, we introduce a framework for executing arbitrary reorganization tasks online, i.e. in the background of normal user workloads without disrupting query results or performance.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration

General Terms

Algorithms

Keywords

Online Reorganization, Main Memory Database, MMDBS

SIGMOD'11, June 12–16, 2011, Athens, Greece.

1. INTRODUCTION

Nowadays, more and more companies heavily rely on modern Business Intelligence (BI) analyses for conducting daily business and supporting management decisions. Supported by user friendly BI tools, reports can be easily generated which are used for analyzing market trends, performing customer segmentations, or conducting risk assessments. Therefore, in the recent years, the trend evolved from static and precalculated reports towards ad-hoc and realtime evaluation of reporting queries for an increasing number of users.

In order to meet user expectations regarding realtime query performance for such analyses, the limiting factor of disk I/O performance has to be addressed. For this purpose, several approaches have been developed, e.g. reducing the amount of data to be read from disk by exploiting special data organization strategies like column stores, avoiding disk access by keeping and processing data in main memory, or using parallel processing techniques in order to exploit the computing capacity of a large number of server nodes.

A prominent class of systems exploiting these techniques are read-optimized MMDBS like MonetDB, Oracle TimesTen, SAP NetWeaver BWA, and others. Compared to traditional general purpose database technologies, read-optimized structures require additional effort to handle updates. On the other hand, integrating updated data in queries and reports is a critical issue for currently postulated real-time analytics. However, updates typically lead to a degeneration of the optimized data structures used for compressing and partitioning data. Therefore, these data structures have to be reorganized from time to time to maintain the MMDBS in an operational state and to be able to deliver the optimal performance. The main challenges here are *(i)* to decide *when* it is worth to reorganize (and what), *(ii) how* to reorganize without downtimes. For traditional disk-based systems many reorganization approaches have been developed [18]. But to the best of our knowledge, no detailed research has been conducted on reorganizations in MMDBS and their differences to disk-based systems.

Because downtimes of database systems usually result in high costs for customers, they are unacceptable for maintenance or self tuning tasks. Therefore, we present an incremental online reorganization algorithm that can be used to transform the data stored in a MMDBS without disrupting its 24x7 availability. It works nearly lockless, is adaptable to the system load, and able to cope with a fixed amount of main memory. Furthermore, we analyzed two data maintenance tasks for the IBM Smart Analytics Optimizer (ISAOPT), a new MMDBS developed by IBM [20].

2. RELATED WORK

Reorganization in disk-based systems. For disk based database systems, many research has been conducted to identify reorganization demands and several strategies to perform them. We refer to the definition of reorganization given by Sockut and Goldberg in [18], namely reformatting and restructuring operations, i.e. operations that change logical and physical data structures without modifying the content of the data like update does. In general, reorganizations are executed with the goal of maintaining the DBS in an operational state, repairing data structure degenerations, and improving system performance. Reorganizations may be required to keep the system running, e.g. when system limits like maximum number of files are reached. One example for this are RowIDs in DB2 in versions prior to 9.5 where only 255 rows could be stored in one page. In combination with compression techniques this became a problem because more data could be stored but not be addressed, leading to wasted memory and a lower throughput because memory pages could not be completely filled [7]. Thus, the number of bytes used for RIDs was increased from four to six bytes in following versions [10]. For supporting the legacy format, both RID sizes could be handled by the new DB2 version. To use the new RID size a reorganization had to be triggered explicitly by the DBA for migrating the data.

Schema Evolution. Sometimes it may be necessary to change the logical data structures used in a database. This might be the case when the database model, that represents the real world environment, changes, e.g., when two databases should be merged after a merger of two companies. It may be required to add/delete columns or tables, rename attributes, etc. Such operations are often referred to as *schema evolution* in literature [14, 15] but are out of the focus of this paper.

Degenerations. Due to updates of the stored data, the data structures containing them might degenerate for several reasons, e.g. when tuples are deleted, updated in-place while modifying the record size, or when clustering structures are broken due to moving tuples to overflow pages.

The current state of the art regarding reorganizations in DBS is surveyed in [19]. The authors give a comprehensive overview of different reorganization approaches for e.g. counteracting degenerations with defragmentation, reclustering the data, or restructuring indexes with removing migrated tuples.

Since multiple reorganization strategies exist and executing them is usually an expensive task, cost-benefit analyses have to be performed based on a cost model and statistics about the current state of the system. Yao et al. [24] developed a simple model to determine when a reorg would be beneficial and proposed a dynamic reorg algorithm based on that. The authors assumed dominant costs for searching in database files, linear degeneration of latter, and constant reorganization costs.

Recompression. MMDBS rely on data compression techniques for reducing main memory consumption. However, after several update cycles, the data distribution may change which can deteriorate the compression rate for entropy encoding algorithms like Huffman coding or frequency partitioning. Therefore, reorganization should be performed to recompute the optimal encoding and use it for recompression. This requires to estimate the expected compression rates and data throughputs before and after a recompression, as well as the overhead for the recompression step. An approach to evaluate possible compression schemes and choosing the most beneficial one was presented in [9]. The high dimensional solution space for different column compressions is evaluated in parallel by multiple agents using a swarm algorithm. Each solution is weighted with the expected improvement and compression costs to find the optimal one. Costs and benefits of different compression techniques can also be considered by query optimizers like proposed in [5]. The work described in [11] presents an approach for estimating the size of indexes after compression by applying sampling techniques.

Cluster Reorganization. Since MMDBS usually rely on massive parallel computations on multiple nodes for fast query response times and scaling reasons, additional reorganization tasks are necessary. Because the slowest execution unit in a parallel system bounds the overall performance, it becomes important to manage the workload on single nodes.

[1] proposes two techniques for this while keeping the system online. One-at-a-time movement (OAT), transfers one page after the other from source to destination nodes and blocks user transactions accessing them while the transfer is in progress. The second approach BULK is faster and less disruptive because a fixed buffer of pages is used as transfer unit but requires a lot of memory for keeping copies of affected index structures and the currently reorganized data area.

To achieve a good load balancing among cluster nodes, a dynamic redistribution algorithm was proposed for horizontally partitioned databases in [8]. Data is transferred from overburdened nodes to less utilized neighbors if thresholds for utilization differences are exceeded. For avoiding chain reactions of redistributions in the cluster, a node sequence is precomputed which leads to a good global load distribution.

Another approach for optimizing the data placement in shared-nothing parallel databases was presented in [6]. The intent is to maximize transaction throughputs of the entire system with applying good data declustering strategies. The presented solution applies heuristics to achieve good load balancing while considering caching effects. A reorganization is only executed after a workload depending decision algorithm found candidate relations where expected redistribution benefits outweigh their costs.

3. THE IBM SMART ANALYTICS OPTIMIZER

The platform for our work and experiments is IBM Smart Analytics Optimizer (ISAOPT) [20]. This is a newly developed high performance appliance designed as extension for IBM System z environments. It will be shipped as combination of a computing cluster and a parallel MMDBS. ISAOPT is designed as an accelerator for analytical tasks to improve data warehousing on System z. As shown in fig. 1 the main idea is to offload the computational intensive analytic workload onto the appliance and handle the transactional queries on DB2 for z/OS. This hybrid approach offers high performance for both OLTP and OLAP workloads while giving high QoS guarantees for system availability.

As illustrated in fig. 1, critical parts of the warehouse data or OLAP tasks can replicated to the ISAOPT cluster, i.e., the user can define data marts, consisting of a subset of all tables and columns in the data warehouse which are impor-

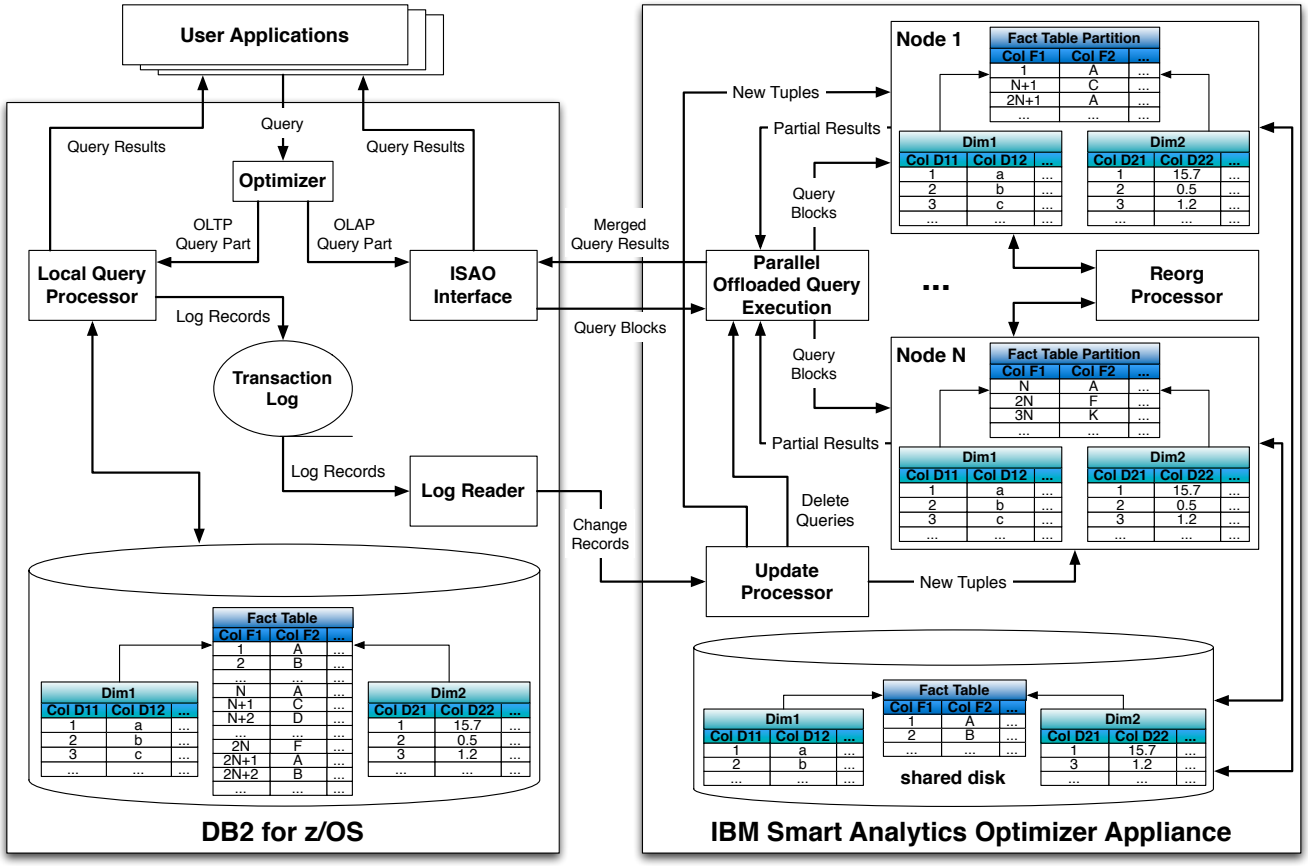

Figure 1: Overview: IBM Smart Analytics Optimizer [20]

tant for the expected workload that shall be offloaded. The warehouse data is stored on disks in DB2 and held completely in main memory of the ISAOPT cluster. A shared disk is used for backup purposes only and not accessed for query processing. In fig. 1, the data mart consists of a large fact table and two dimension tables. Since fact tables usually store the gross of the entire warehouse data, it is partitioned horizontally and distributed among all nodes in the cluster. Dimension tables are usually small and are replicated to each node to avoid distributed join processing.

Queries or query blocks which can be accelerated are offloaded by the DB2 query optimizer. The offloading decision depends on communication costs, the availability of the relevant data in the accelerator, and expected execution times in DB2. Offloading is completely transparent to user applications and queries will be executed by DB2 if they fail to run on the accelerator. If queries can be offloaded, they are split into parallelizable blocks which are executed on the cluster nodes. Their partial results are merged and transferred back to the user application.

The customer benefits from query execution on ISAOPT are twofold: First, with offloading processing on ISAOPT, the customer achieves a CPU cost reduction on System z. Second, OLAP queries can be accelerated on ISAOPT between one and three orders of magnitude – depending on the type of query – by using modern MMDBS techniques.

3.1 Highly Efficient Data Layout

To store the data in ISAOPT, tables are horizontally partitioned into fixed size blocks that are optimized for vectorized processing and efficient cache utilization like in MonetDB/X100 [4, 25]. A similar layout to the PAX model [3] is used to benefit from columnar processing and co-located storage of each row's attributes. The partitioning is tied to the compression algorithm that has been developed as part of the Blink project [16] and is used by ISAOPT in order to keep the huge amount of data completely in main memory of a computing cluster.

It is a field based dictionary compression algorithm, i.e., it replaces values for single attributes with new, shorter codes and achieves compression rates close to entropy. For this purpose, columns are partitioned based on histograms and a dictionary is constructed. Shorter codes are used in partitions containing frequent values and longer codes for infrequent ones. The cross product of all column partitions in a table are called *cells*.

The compressed data retains some properties which allow an efficient evaluation of several predicate types directly on the compressed attribute values. Column codes within one partition are order preserving. Thus equality and range predicates can be evaluated with fast bitmask operations without decoding each value, saving time for costly dictionary lookups [16]. Furthermore, fixed code lengths are used

which allows an efficient access to single attributes with simple offset computations. The predicate evaluation is parallelized with SIMD vector operations within each thread and by processing several blocks in multiple threads utilizing modern multicore processor architectures [13].

3.2 Snapshot Semantics

ISAOPT was designed for accelerating analytical workloads. Thus, the main focus is on processing read-only queries. Usually, updates play only a secondary role and are performed batch-wise in longer time intervals. But depending on customer requirements the update frequency as well as the update strategy can be configured. Besides the batch wise strategy, ISAOPT plans real-time update support.

In ISAOPT, updates are serialized. Because high query throughput as well as high availability are most important, each task requiring write operations on the data (both updates and reorganizations) has to be performed concurrently to running queries but without affecting query performance or disrupting their results due to temporary inconsistencies.

For this purpose, ISAOPT implements snapshot semantics to hide background modifications without synchronizing write accesses to data structures using locks. With each tuple, validity timestamp codes are associated by introducing two additional auxiliary columns per table, called *epochs*. A *create epoch* identifies the snapshot which was created on insertion of the tuple into the table. The *delete epoch* is used to mark tuples as deleted from the point in time on, that is identified by the snapshot with the respective ID. The maximum possible epoch code is used as delete epoch for tuples which have not been deleted yet. Epochs can be efficiently evaluated with bitmask operations using an additional range predicate that checks if a tuple belongs to a valid snapshot:

Definition 1 (Epoch predicate)

$$createEpoch \leq queryEpoch < deleteEpoch$$

Each data mart has a single epoch value which identifies the currently valid snapshot and is used by new queries processed in the system. Each update creates a snapshot, using a higher epoch value to mark newly inserted and deleted tuples. In-place modifications are split into an insertion and a deletion. After an update completes successfully, it atomically increases the valid table epoch by setting a single value and broadcasting it in the cluster. Subsequent queries will use the new snapshot and reference the updated data set. A rollback of an update is easily possible by simply scanning the data and undoing the updates.

Because no locks are necessary for block accesses, updates affect the query performance only marginally when some system resources like CPU cores and main memory are required to execute updates and are not available for parallel tasks like table scanning.

3.3 Reorganization Tasks

ISAOPT heavily relies on cache efficient, highly compressed data structures and the possibility to operate directly on compressed codes for achieving maximum query performance. But the compression rate and data structure layout is optimal only after the initial load of the warehouse data. Updates that have to be executed to keep the replicated data marts in sync lead to several problems and degenerations that require a reorganization.

Recompression and Garbage Collection. As outlined in section 3.1, dictionary codes from frequency partitioning are determined based on column value histograms. Therefore, later inserted values could not be considered for dictionary construction. In order to be able to handle them, new codes have to be assigned which requires a dynamically growing dictionary. But this is not possible for regular partitions since they use fixed length codes, exploit the whole code space to achieve optimal compression, and need to be order preserving for being able to evaluate range predicates without decompression [16, 21]. Thus, to be able to handle new values, ISAOPT introduces two special column partitions: an *extension partition* where a dynamically growing dictionary of fixed preallocated size is used and the *catch-all partition* which stores values unencoded. Applying range predicates directly on encoded values in the extension partition is not possible anymore because they are not necessarily order preserving. But at least, the data can be compressed. The catch-all partition works as last resort for values that cannot be efficiently encoded, or for storing new values after overflow of the extension dictionary. Thus, no compression or runtime benefits are achieved there [21].

Even if a code for an updated value is available, an in-place update may not be possible since the modified column(s) might cause changing the corresponding cell. Because tuples are stored in separate blocks per cell for easy bitmask construction, this would require to re-encode and move the entire tuple to a block for the new cell, leading to a gap in the original block. These gaps disrupt the memory and cache efficiency of the data structures but cannot be avoided due to the implemented snapshot semantics. Furthermore, tuples marked as deleted have to be kept valid as long as a query is referencing them to guarantee correct results. In the worst case, the system may run out of main memory if these gaps are not removed.

Another problem occurs when the data distribution shifts after updates. Since partition codes are assigned depending on the initial value frequency, they can be suboptimal if the underlying histograms change. Values that have previously been infrequent and, therefore, are represented by long codes can become frequent and require more bits for being stored.

Cluster Reorg. In order to achieve a good load balancing in ISAOPT, fact tables are partitioned horizontally among cluster nodes. A round robin strategy is implemented to guarantee that each node handles more or less the same amount of data. Thus, it is very likely that each node receives at least one tuple for each cell. Experiments show that usually only some cells contain most of the tuples while the majority is nearly empty. Since for cache efficiency fixed size blocks are used per cell, a lot of memory is wasted on each node for storing only partially filled blocks. This problem grows linearly with the number of cells and the number of nodes in the cluster and therefore leads to a bad system scalability. Thus, reorganizing the data distribution is necessary here.

4. SNAPSHOT REORGANIZATION

Because system downtimes result in high costs for customers, they are usually not acceptable for maintenance tasks in enterprise-critical database applications. If for example the ISAOPT appliance would not be available for executing the computational intensive analytical workload, the queries have to be processed on System z with slower re-

sponse times and high costs for computations on this platform. Therefore, all maintenance tasks have to be executed with minimal negative impact on the running system. This means, (1) that reorganizations should be performed as background tasks while the normal system workload is processed, (2) with only a minimal performance impact on concurrently running user tasks, (3) that reorganizations must not lead to inconsistent data states visible to running transactions and (4) must be able to cope with limited system resources, i.e. processing power and available memory.

In the following, we present *SnapshotReorg*, a lightweight online reorganization algorithm that can be used for performing arbitrary transformations on data stored in the database system for reorganization purposes, and exploits snapshot semantics to hide modifications from running tasks. It is able to cope with a fixed buffer of main memory and is adaptable to the system load. It is possible to increase the buffer size or the number of available threads executing the task in times of low system utilization for speeding up the reorg process. When these resources are needed for processing the user workload, they can easily be withdrawn from reorg. Furthermore, because the transformation process is interruptible, it can even be aborted and resumed later on without losing much previous work. This property makes the reorg process also easily recoverable in case of errors.

4.1 Reorganization Framework

The reorg process consists of three phases. The **Decision Phase** is triggered after certain events that might entail a reorganization like shown in fig. 2, e.g. a finished update cycle, after a batch of queries has been executed, or the cluster infrastructure changed. Based on statistics, a decision algorithm is executed that determines which reorganization strategy would be beneficial and shall be processed (line 1 in alg. 1). When the reorganization costs do not exceed the expected benefits, the transformation is executed iteratively in parallel by multiple threads. The current load on the system is tracked on each node by a workload optimizer which can increase/reduce the number of threads for *SnapshotReorg*.

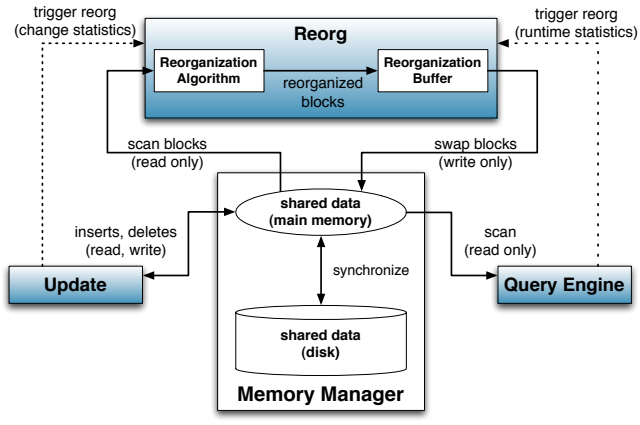

Figure 2: Reorganization Algorithm Overview

In the **Transformation Phase**, tuple data is transformed block/page wise into a shadow copy which remains hidden from queries and updates as long as this phase is running. On completion, the reorganized data is atomically exchanged with the old one during the **Swap Phase**. For hiding new shadow copy blocks, the snapshot semantics is used which will be explained later. Because updates might be running asynchronously to the reorganization process, each reorg has to be registered for a mart before the actual data transformation starts (line 4 in alg. 1). This will cause the *mart* object to track changes made by updates during the *transformation phase* which will be applied later in the *swap* for synchronizing the shadow copy.

The old blocks to be reorganized (depending on the reorganization task) are managed in a source buffer (lines 5, 6 in alg. 1). They could originate from arbitrary sources, e.g. are already stored in the reorganizing node's memory, are streamed from disk or transferred via network from other nodes. Thus the data exchange between nodes is processed similarly to the strategy used in BULK (Sect. 2), but neither leads to inconsistencies nor blocks query execution since snapshot semantics (Sect. 3.2) is exploited. Queries do not need to be aborted or restarted because new blocks remain hidden due to the epoch predicate.

Because it is infeasible to hold a shadow copy of the entire warehouse data in memory, a reorg buffer of fixed size is used (line 7 in alg. 1). This target buffer size can be configured automatically by the workload optimizer. Reorganized tuples are inserted in the buffer until it is filled or the source buffer exhausted (lines 10-13 in alg. 1). On buffer overflow, a special handling is required for blocks that could not be reorganized completely (line 15 in alg. 1). Like illustrated in fig. 3, such a block still contains tuples that must remain valid and, therefore, cannot be overwritten. But it must be prevented that already reorganized tuples are valid "twice", i.e., stored in an old and a new block. Incompletely processed blocks are avoided by either discarding all reorganization steps applied to them, or by introducing a buffer overflow area and process them always completely. The first

Algorithm 1 Reorg Main Algorithm

Require: reorg trigger event *event*, data mart *mart*, fixed buffer size *bufSize*.
Ensure: reorganizes *mart* if beneficial.

```
1:  DR := decisionAlgorithm(event, mart)
2:  if DR.isReorgBeneficial = true then
3:    func := DR.transformationFunction
4:    mart.setReorgActive(true, func)
5:    snapshot := mart.createReorgSnapshot(DR.task)
6:    sBuf := createSourceBuffer(snapshot.getBlocks())
7:    tBuf := createTargetBuffer(bufSize)
8:    while sBuf.tupleCount > 0 do
9:      bufferOverFlow := false
10:     while sBuf.tupleCount > 0 AND
        bufferOverFlow = false do
11:       t := sBuf.readNextTuple()
12:       bufferOverFlow := tBuf.insert(func(t))
13:     end while
14:     if bufferOverFlow = true then
15:       handleBufferOverflow(sBuf, tBuf, func)
16:     end if
17:     mart.swap(sBuf.getBlocks(), tBuf.getBlocks())
18:   end while
19:   mart.setReorgActive(false)
20: end if
```

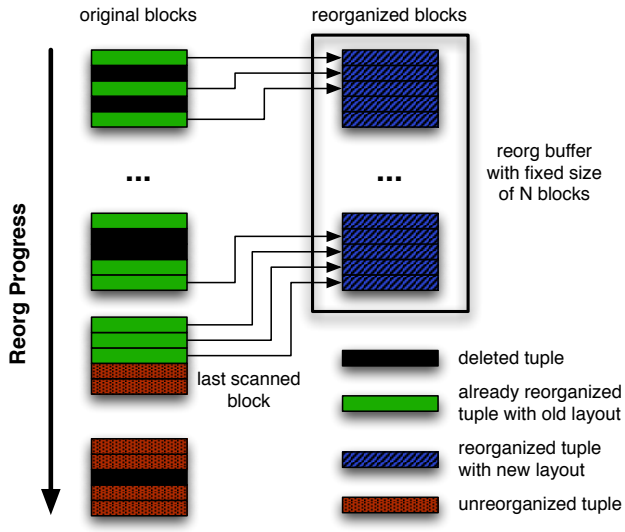

Figure 3: Reorg Buffer Overflow

solution meets the memory constraints but loses reorganization work. The second solution does not waste any work but requires additional memory. An alternative would be to mark already reorganized tuples as deleted with the reorganization epoch. But for this solution additional recovery information has to be stored for being able to restore the old blocks in case the swap fails after this point in time.

Because both, reorganizations and updates are performing write operations which could lead to temporary inconsistencies, a special synchronization of writes is required. A writing task with a larger epoch has to wait until all changes made by tasks with lower epochs have been committed or rolled back before committing its own changes and increasing the active epoch. Otherwise, the incomplete modifications using the lower epoch would be implicitly regarded as committed by the epoch predicate.

Depending on expected runtimes of updates and reorganization iterations, the used epoch values can be chosen that faster tasks are only minimally blocked. Therefore, the slower task preallocates an epoch with a higher value so that the expected number of cycles of the faster task can be completed before commit. If for example realtime updates are performed, an update iteration will be most likely faster than a reorganization step. For large batch-wise updates and small reorganization buffers, it may be faster to finish several reorganization iterations before a single update commits. Because *SnapshotReorg* is a background task, an update with a higher epoch value must be able to discard a concurrent reorganization step or enforce a premature swap before the reorganization buffer is filled completely. This is important for cases where only some resources are assigned to *SnapshotReorg* and therefore it may take a long time until a swap would be processed.

Each step of filling the buffer with transformed blocks in the *transformation phase* and exchanging them with old ones creates a new snapshot during the **Swap Phase**, using a new epoch value. The mart object's *swap* algorithm is described in listing 2.

Algorithm 2 Mart::Swap Algorithm

Require: list with old blocks L_{old}, list with new blocks L_{new}.

Ensure: atomically replaces the blocks in L_{old} with those in L_{new}.

```
 1: lockMartForNewUpdates()
 2: runningUpdates := getCurrentUpdateList()
 3: if nextUpdateEpoch < reorgEpoch then
 4:    nextUpdateEpoch := reorgEpoch+1
 5: end if
 6: for all b_old ∈ L_old do
 7:    b_old.deleteEpoch := reorgEpoch
 8: end for
 9: for all b_new ∈ L_new do
10:    b_new.createEpoch := reorgEpoch
11:    b_new.deleteEpoch := MAX
12: end for
13: unlockMartForNewUpdates()
14: for all u ∈ runningUpdates do
15:    u.suspend()
16:    u.refreshSnapshot(L_new)
17:    u.resume
18: end for
19: applyTrackedDeletes(L_new)
20: waitForUpdatesWithLowerEpoch(reorgEpoch)
21: queryEpoch := reorgEpoch
22: reorgEpoch := getNextReorgEpoch()
```

To be able to commit the reorg changes, the swap procedure guarantees in lines 3-4 that following updates use a higher epoch ID than the current reorg step (if it has not been preallocated yet in case of slow and long running updates). After this, the create and delete epochs for both new shadow copy blocks and old source blocks are modified in lines 6-12 that they will be included in snapshots created after swap finishes. Because multiple values need to be modified, this leads to temporary inconsistent snapshots until all blocks have been processed. During this critical section, starting new updates must be prevented which is achieved with locks in lines 1 and 13. But compared to the *transformation phase* this is a lightweight operation and new updates are blocked for a short time only.

While blocks are transformed in the background to query and update processing, tuples might be inserted or marked as deleted in the meantime. As long as *SnapshotReorg* is filling the buffer, new blocks remain hidden from updates (cf. view 1) in fig. 4). Thus, no update is blocked which would be the case if write access is synchronized with locks. This is important because the amount of available resources for reorganizations depends on the system load and therefore, it might take a longer time until the transformation phase finishes and writes to the new blocks can be performed.

New tuples can be written into separate blocks using the new reorganization scheme (which has been registered on reorg start) to avoid that they have to be processed by reorg again. These blocks can be merged after each update iteration. But marking tuples as deleted possibly leads to conflicts between the new and the old version which have to be solved. First, *SnapshotReorg* might read uncommitted deletes (right blocks in reorg buffer in fig. 4). Because each update that is active while a reorganization is running might

be rolled back, they must not be physically removed. They are regarded as valid, transformed, and stored in the buffer. Only older tuples can be skipped. Second, *SnapshotReorg* might have created a reorganized shadow copy of a tuple that is marked as deleted afterwards (blocks in reorg buffer on the left hand side in fig. 4).

As stated above when discussing alg. 1, delete (and rollback) operations are tracked by the mart as soon as the reorg process registers. They are applied to the shadow copy blocks in line 19 before the swap finally finishes. Uncommitted deletes are rolled back if the corresponding update has been rolled back and missing deletes are applied for each committed update. For preventing that the list with tracked operations increases due to concurrent updates, these updates are modified slightly in lines 14-18. They are suspended for a short time and their snapshot views are updated (cf. view 2) in fig. 4). Further deletes are processed twice – in the old as well as the reorganized blocks. Thus new deletes do not need to be processed by reorg and the reorg step can easily be canceled with discarding the new blocks without losing any deletes. But other well known strategies like optimistic locking might be possible to solve the update conflicts.

After the swap finished successfully, the epoch value used for new query snapshots is increased in line 21, guaranteeing a consistent view on the data for queries that are started after this point in time (cf. view 3) in fig. 4). Finally, the epoch to be used for the next reorg iteration is determined in line 22 like previously discussed. The old reorganized blocks marked as deleted remain valid as long as another task is referencing them. As soon as they are not used anymore, they are physically deleted and overwritten by the next reorganized blocks.

4.2 Epoch Overflow Handling

Using snapshot semantics requires a strictly ordered sequence of epoch values for correctly evaluating valid tuples with the epoch predicate. Thus, using a fixed code length for epoch columns will sooner or later result in an epoch overflow. This has to be handled accordingly to guarantee correct query results. For this reason, either old epoch codes may be reused if they are not needed anymore, or code lengths have to be increased. Neither of both approaches defines an optimal solution.

The first one is implemented with wrapping epochs in C-Store [22] and impacts query execution times because they have to evaluate additional predicates for disjoint epoch sequences before and after an overflow. If no overflow would occur, only the simple range predicate is sufficient.

The second approach requires a complete reencoding of all tuples. Especially in read optimized data structures, increasing code lengths is usually difficult. Besides, this only defers the overflow problem. Of course, the number of epoch resizes grows only logarithmically with respect to the maximum epoch numbers which are encodable. But if it can be avoided, this solution would be preferable.

For circumventing both problems, we developed a new approach which closely corresponds to the idea of cut-off points introduced in TSQL2 [12]. A cut-off point can be defined for a table and denotes the point in time when deleted tuples will become invalid and are not considered for historical queries anymore. Reorg therefore can safely remove the affected tuples without impacts on running queries.

In our environment we assume that snapshots are only used internally and historical queries for temporal analyses are only executed using the time dimension. Only the most recent epoch is used for processing new queries. Therefore, we can define a cut-off point for the newest snapshot using an older epoch ID in the ordered sequence than the currently active one and additionally is not referenced by a running query anymore. All epochs preceding this ID can be reused for new snapshots.

To avoid additional predicates, epochs are reset while the system is running. The reset is executed in place without disrupting query results. For being able to do this, an additional epoch column pair is introduced for each tuple. Only one of them is active for epoch predicate evaluation by queries which is determined on query start when its snapshot view is created. But both are adjusted incrementally after modifications by table updates. The number of running queries that are referencing either the first or the second snapshots is tracked with reference counters.

After the initial load of the table data, all epochs are initialized to the minimum value and queries use the first pair for evaluating the epoch predicate. A sequence of update operations or reorganization steps increasing epoch values can be performed increasing the first epochs. As soon as the maximum possible epoch value for the first table epoch is reached, an overflow handling is triggered. The update causing the overflow uses the the second table epoch. When this update commits, new queries start creating their snapshots only using the second epochs. Following updates are regularly executed, increasing the second epoch, until the next overflow occurs.

Till the first overflow, all queries are only referencing the first epochs. After switching to the second ones, all tuples must be scanned to reset the first - now inactive - epoch column pair to the minimum values. To do this without corrupting the snapshots of running queries, it must be waited until all queries evaluating predicates on the first epochs finish. Their number is tracked with the respective reference counter and strictly decreasing because new queries are only started using the second epochs.

Update operations have to be extended slightly to set values for the inactive epoch columns which might still be referenced by old queries that are still running in the system.

Update without concurrent old queries: When a tuple is inserted, its inactive create epoch is set to the min and its inactive delete epoch to the max value. Marking a tuple as deleted sets the inactive delete epoch to the min value.

Update with concurrent old queries: Newly inserted tuples must remain hidden and newly deleted ones must remain valid for old queries. Therefore, new tuples get the maximum possible inactive create epoch value and the inactive delete epoch is not modified, neither for inserts nor for deletes. For deletes, only the active delete epoch column is set to the update epoch. The reset algorithm will handle these tuples.

As soon as the last blocking old query finishes and a possibly running concurrent update adapted its modification behavior accordingly, the reset operation can be triggered. The reset works similar to the update without concurrent old queries but scans all tuples. This avoids any synchronization between updates and the reset. The inactive create epoch of a tuple is always set to the min value. The inac-

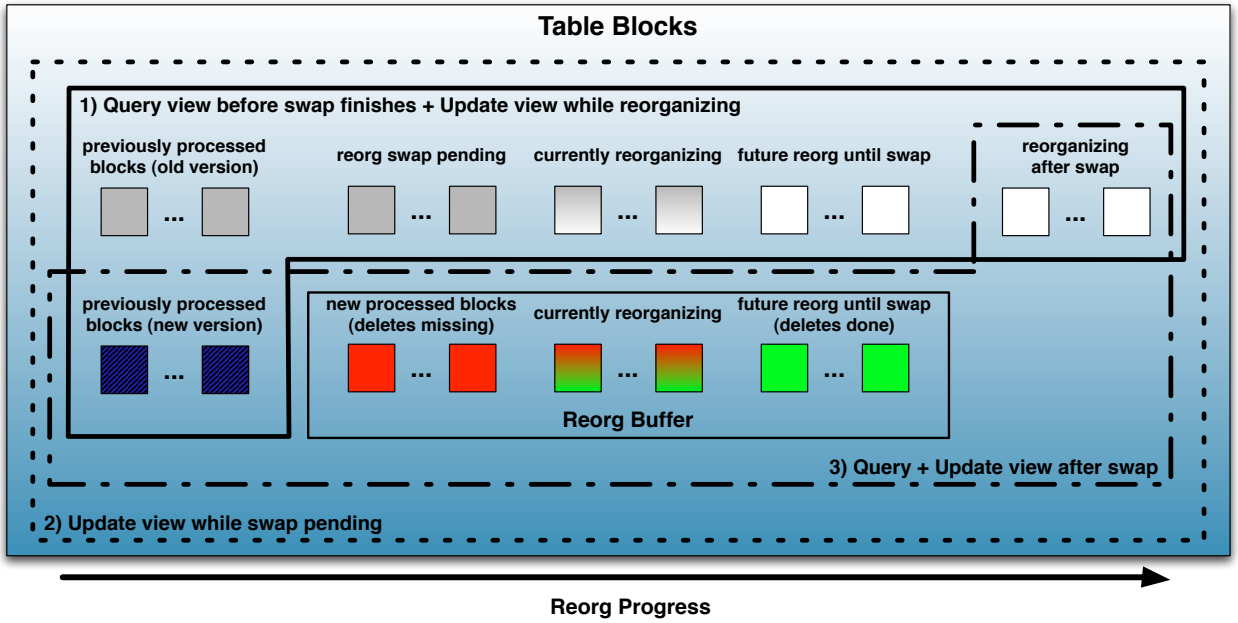

Figure 4: Views in different Reorganization Phases

tive delete epoch is set to min if any of both delete epochs has a value less than max. Otherwise the inactive delete epoch remains max. According to the epoch predicate the tuple therefore will be valid/invalid from the point in time on when the active epoch pair switches again.

Finally, after resetting the inactive epochs completed, the inactive table epoch is set to the minimum value. This leads to a similar situation to the initial one after the load where only the epoch roles have been switched. When further updates reach the next overflow point, the handling is performed analogous, hiding the next reset operation from new queries. With this approach an update needs to be blocked if an epoch overflow happens before the last handling completes.

At first sight, the proposed approach doubles memory requirements for storing epoch values. But the epoch overhead can be reduced. Because the reset is a simple scan operation, it should complete very fast. Thus the code length for each epoch column can be reduced to the minimum number of bits that is required to represent an update sequence between two overflows without blocking any update. Furthermore, when epochs are used on block level - which is the case for the snapshot reorg anyway - it is possible to omit tuple creation epochs, nearly halving the memory overhead. For doing this, all newly inserted tuples must be written into separate blocks which will become available for the creation of query snapshots atomically after update commits. These *trailing blocks* might be merged with a reorganization for avoiding waste of memory due to incompletely filled blocks. Deletion epochs must be kept since random access is needed for them.

As a reorganization task, the number of bits needed to store epoch column values might be computed dynamically based on workload statistics estimating the required times and frequencies. The table data could therefore be transformed to optimize the storage layout. This epoch code transformation, as well as the reset, can be easily piggybacked with other reorg tasks that transform tuples anyway.

4.3 Reorganization Tasks

In the following we describe the reorganization tasks we have developed for ISAOPT and which are executed by the framework described above. Executing other tasks with our framework may be possible too, e.g. for redistributing data on cluster nodes according to a new distribution function. In general, the framework should work well for all tasks that can be divided into several steps of reorganizing old and swapping with new data while being able to operate on both – due to reorg memory constraints possibly incomplete – subsets of the entire data. Especially for tasks like schema evolution this is not trivially given and is left for future research in this direction.

4.3.1 Garbage Collection

The first transformation function for *SnapshotReorg* is used to physically remove tuples that have been marked as deleted and are not required anymore. For this purpose, old blocks are scanned and non-deleted tuples are copied into new blocks. The negative impact on query execution times through records marked as deleted was examined in [2] for disk based systems. The costs have been modeled with page misses and grew, as expected, linearly with respect to the number of deleted tuples. But as mentioned above, the same arguments can be applied for MMDBS since cache utilization is decreased. Since target blocks use the same layout and will contain less than or equal data as the source blocks, the memory overhead is constant for this reorganization. It can easily be combined with other approaches like recompression or bank reassignment.

4.3.2 Recompression

For recompressing tables, several strategies exist. Different compression algorithms may be applicable or can even be combined to improve compression rates or data throughputs, depending on data distributions or current bottlenecks in the system. Whether applying a new compression algorithm would be beneficial, and which one should be chosen has to be determined by the decision algorithm.

Generally, when a table T is recompressed incrementally while queries are running, its entire data is partitioned into two disjoint subsets T_{old} and T_{new}, where T_{old} refers to all tuples using the old compression scheme, and T_{new} to those using the new one. It must be guaranteed that queries return correct results on the entire data set $T = T_{old} \cup T_{new}$, i.e., no tuple is missed or processed twice. This can be easily guaranteed with the snapshot semantics where the currently active epoch defines which tuple should be processed in which partition. Therefore, each query can be split up into two sub queries, one for each partition, and a final phase where the results are merged. This additional overhead will slow down query processing, but it avoids a system downtimes in environments where storing the entire table data twice – once for each compression scheme – is infeasible.

The choice of transformation function for a given table depends on the source and target compression scheme. Newly inserted tuples of concurrent updates can be ignored for reorganization because they can be inserted using the new layout. For recompressing dictionary encoded data where only a new dictionary is applied, the transformation is a simple mapping of old codes to new ones. This mapping can be precomputed before the reorganization. Thus, we focus on the following on recompressing frequency partitioned data to address negative impact of shifts in the data distribution and values that have to be stored in extension and catch-all cells after updates (Sect. 3.3). When the number of tuples in the latter exceeds a threshold or major changes in column value histograms are detected, the new partitioning is recomputed. New column values are integrated in the dictionary and optimal code lengths can be determined. Due to that, a better compression rate is achievable and query response times can be improved by allowing to operate directly on compressed data again, which was impossible in those special cells before.

4.3.3 Cluster Reorganization

Finally, a further reorganization task is to merge partially filled blocks of each cell. This is useful to address the problem described in Sect. 3.3 which occurs when data is distributed with a round robin strategy among cluster nodes. These trailing cell blocks can decrease the system scalability if the number of cells containing any data - and therefore at least one trailing block per node - is considerably high, which is the case for data sets containing a lot of weakly correlated high cardinality columns. This results in a large number of partitions and possible combinations of them. In order to reduce memory and processing overhead these blocks are merged and redistributed in the cluster. After the reorganization only one partially filled block remains per cell.

Several additional distributed reorganization tasks are possible for workload balancing, handling changes in the cluster when nodes fail or new nodes are added or if other compression and distribution techniques are used. But this is out of the scope of this paper and left for future work.

5. EVALUATION

In this section we want to evaluate our reorganization approach. First, we show that the reorganization framework can be used in the background to query processing without major negative performance impacts. Second, we analyze the negative impact of tuples marked as deleted on memory utilization and query response times, which can be removed by a reorganization. Last, we evaluated a first approach of a data redistribution among nodes in the cluster for reducing the overall memory consumption.

As test benchmark, TPC-DS [23] was chosen. It was specially designed as successor for the commonly used TPC-H benchmark with intend for a better modeling of decision support systems that are used in real world scenarios. For simplifying measurements, only one part of the entire schema, the STORE_SALES snowflake, was used. The data set was generated with the provided benchmark tool using scale factor 300. This results in a mart size of approximately 100 GB uncompressed csv files. This is sufficient to show impacts and improvements through reorganizations. Since we used the single snowflake and in the current development state ISAOPT only supports a subset of the operations defined in the SQL standard, only a subset of the benchmark queries[1] could be run.

All experiments have been conducted on an ISAOPT test cluster consisting of three HS22 blades [17]. Each blade was equipped with two Intel Xeon quad-core processors with 2.66 GHz, and 8 MB of shared L2 cache. 48 GB of DDR3 RAM was provided for each of the blades. The nodes within the cluster were connected using a 10 GB Ethernet network.

5.1 Reorganization Overhead

The spirit and purpose of an online reorganization algorithm is to execute the maintenance operations in the background to the normal system workload without major performance impacts on latter. In this section we want to evaluate how our proposed *SnapshotReorg* algorithm meets this requirement. Therefore we measured the runtimes of the TPC-DS benchmark query set while a reorganization is concurrently executed.

To analyze the impact of the concurrent system workload we conducted two measurements, the first with a high workload on the system and a second with a low one. We executed the benchmark queries serially in an infinite loop (query stream) and measured the overall benchmark runtime after each iteration. For the low system workload one query stream has been executed, for the high workload four streams were running in parallel. For the concurrent reorg process we marked 20% of the tuples as deleted before starting the experiments and executed a garbage collection after an initial phase, using four reorg threads and a reorg buffer of 400 MB. The results are plotted in fig. 5.

As we can see in fig. 5 the reorg process caused some overhead because the runtimes for the benchmarks increased as soon as the reorg has been triggered. This is due to the fact that the reorg process requires CPU power which can not be used for query processing in the meantime. When the reorg processes finished, the benchmark performance hence increased rapidly resulting in lower response times for each iteration. Another observation is that the reorg process

[1]queries 3, 19, 34, 42, 46, 48, 55, 68, 73, 79, 96 have been executed; only inner query blocks of nested queries were offloaded

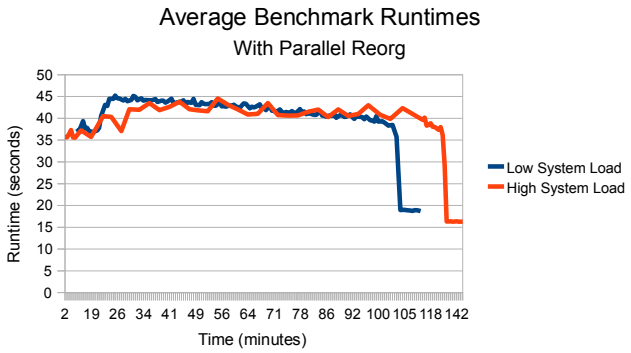

Figure 5: Benchmark Runtimes with concurrent Reorganization

running concurrently to a high system workload requires a longer time to complete in comparison to the low system utilization because more resources can be utilized. Note that the average runtime for a query stream in the high workload case is slightly lower than the one for a low workload. Because query blocks of the concurrent streams can be processed in parallel some time can be saved in phases that require a serial execution in a single stream like grouping.

Another fact that can be observed in fig. 5 is that the time required for processing a benchmark iteration is decreasing slightly while reorg is running. Because *SnapshotReorg* processes the mart data incrementally, new queries started after each swap can benefit from the already achieved cleanup that is analyzed more detailed in Sect. 5.2. Despite the fact that we used 25% of the available cores only for reorg, the overhead was quite low in comparison to the achieved benefits. Considering the incremental decrease of the overhead, one can say that *SnapshotReorg* can run without major negative performance impacts on concurrent queries.

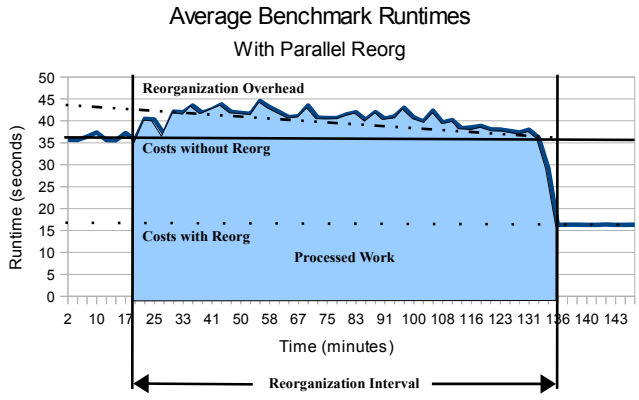

Figure 6: Sample Reorganization Overhead

The reorganization overhead can be considered in the decision algorithm to decide whether a reorg is beneficial or not. fig. 6 illustrates relevant parameters. The work required for executing the queries is indicated by the area under the curve. Before reorg started, it is higher since for each query a longer time is required for processing it (indicated by solid horizontal line). After reorg finished, each query is executed faster (dotted line). For executing the reorg, resources are required which can not be used for query execution. Thus, their execution times increase. This overhead is indicated by the dash-dotted line in fig. 6. Depending on the amount of system resources that are provided for reorg, the shape of the curve can be modified, e.g. more resources may be spent in times of low a system load. If all the parameters can be modeled, estimated, or measured e.g. with a calibration phase, they can be used in a model similar to the one developed by Yao et al. [24] to determine reorganization points where the saved work outweighs the overhead.

Note that fig. 6 is a simplification for the whole reorg process. The benchmark runtimes drop significantly at the end of the reorganization. This is due to the fact that in our setup all tables have been reorganized serially. The process started with some small dimension tables and most of the time reorganized the fact table. This caused the slight decrease in benchmark runtimes until $T = 130$. At the end of the reorganization, the larger dimension tables for customer data have been processed that were used in the longer running queries. Latter showed significant performance improvements after the garbage collection finished (cf. sect. 5.2). The reorg overhead remained nearly constant over the whole reorg interval since we did not implement the workload management yet and used a constant number of threads. A lesson from that is that the reorganization benefit strongly depends on the current system workload which should be considered in the decision phase.

5.2 Garbage Collection

After we have shown that the reorg process can be executed with a small overhead in parallel to normal system tasks, we now analyze how it can be utilized for counteracting degenerations caused by updates. Due to the snapshot semantics in ISAOPT all updates are either inserts or deletes (Sect. 3.2). For our measurements we kept the amount of stored data constant and concentrated on tuples marked as deleted because they cause gaps in the data structures and waste memory. We simulated different extends of update operations with loading the benchmark data set and randomly marking different percentages of tuples as deleted. Assuming a random distribution of deleted tuples is valid since ISAOPT distributes incoming tuples among all nodes with a simple round robin fashion during the load and they are not stored clustered.

We expect that the appliance benefits from this reorganization twofold - first, the memory which is wasted for storing tuples marked as deleted will be freed and second, we expect improvements of query runtimes because the deleted tuples do not have to be considered anymore when executing query plan operators. To analyze the impact on queries we measured the benchmark runtimes twice for several percentages of deleted tuples, once before starting the garbage collection and once after latter has been finished. Thus, we eliminated the reorganization overhead and obtain two baselines for showing the pure runtime benefit. In addition to that we scanned the data set and measured how much memory was used for storing tuples marked as deleted.

As expected the reorganization successfully cleaned up the data structures like shown in figure 7. Due to the equal distribution of deletes, approx. the same amount of blocks was freed like tuples have been marked as deleted before. From memory perspective a reorganization therefore is beneficial as soon as a certain threshold of deleted tuples is

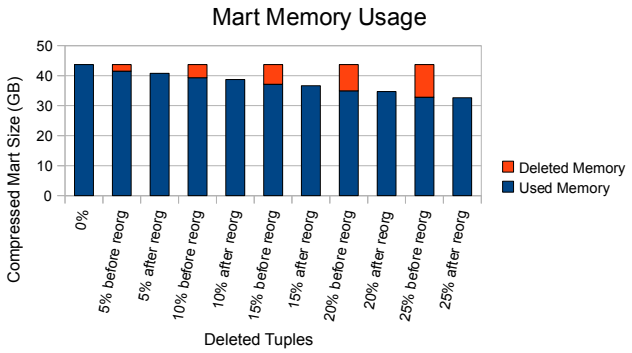

Figure 7: Memory Utilization Improvement through Garbage Collection

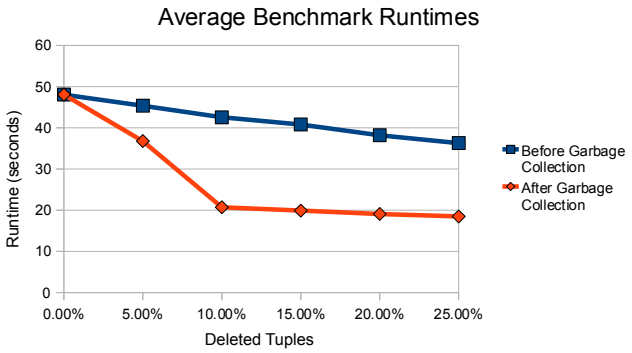

Figure 8: Benchmark Runtime Improvement through Garbage Collection

exceeded. The value of this threshold depends on the application's memory requirements. If a lot of memory is required for storing temporary data during query executions a lower threshold should be chosen. Otherwise, the reorg can be postponed to clean up a large amount of tuples for keeping the reorganization overhead low.

Now turning to the runtime impact of garbage collection shown in figure 8. The benchmark runtimes before a garbage collection decrease slightly with increasing percentage of deleted tuples since latter are skipped in the pipeline for query operators after scanning the compressed values, e.g. in the scan on decompressed values or the grouping and aggregation phase. Regarding the benchmark runtimes after reorg, it is interesting that they initially decrease strongly until a kink in the approx. linear improvement occurs where the 10% deleted tuples have been cleaned up. After this percentage the runtimes only improve slightly. This can be explained with caching effects. To minimize cache misses due to branch mispredictions, the table scan on the compressed values is executed completely branch free, i.e. all predicates that can be evaluated directly on compressed columns are evaluated in this initial step. For each predicate a bitmask is created which are merged at the end of each tuple batch. The final bitmask is then used for filtering tuples and pass them to following operators in the query plan pipeline. Removing deleted tuples therefore has two benefits.

First, the time is saved which would be required for evaluating the predicates on the compressed columns. This explains the only slight improvements after the kink because this evaluation is processed pretty fast with simple SIMD vector operations.

The second improvement comes with a better cache utilization. After the predicates have been evaluated in the first pass, the tuples reside in the processor cache. Following query plan operators in the pipeline benefit from this. But if tuples are marked as deleted, they do not pass the epoch predicate of the first pass and therefore only waste cache memory causing further cache misses. Therefore the runtimes improve strongly until the result sets can be cached completely which was in our case at 10% deleted tuples.

According to this, only queries benefit from the improved cache efficiency that produce large (intermediate) result sets. This can be clearly seen in fig. 9. There we plotted the percental runtime improvement per query which is achieved through garbage collection, i.e. the difference between the

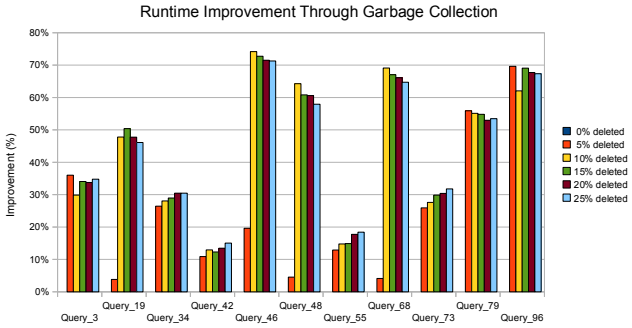

Figure 9: Runtime Improvement of Single Queries through Garbage Collection

upper and the lower curve in fig. 8. Queries 42 and 55 benefit only slightly from the reorg since they are simple and join only two small dimension tables with the fact table. More complex queries with larger joins like queries 79 and 96 already benefit from reorg when only a small amount of deleted tuples (5%) is removed. On contrary, the queries with the largest number of joins (19, 56, 48, 68) and therefore the largest intermediate result sets have only slight improvements until the 10% deleted tuples are cleaned up.

These experiments show that the reorganization benefit for query runtimes depends on the kind of queries which are executed in the appliance's common workload and the data they touch. This makes the decision algorithm more complex than regarding just the memory utilization because the workload might shift over time. Furthermore, the reorg efficiency depends on hardware parameters like the cache size which requires further tuning to derive optimal reorganization points.

5.3 Cluster Reorganization

For measuring the negative impact of incompletely filled fixed size cell blocks, we counted them for each cell. As expected through the round robin data distribution, nearly all cells contained one trailing block per node in the cluster.

Our experiments with the TPC-DS data set showed that the trailing block overhead was quite low. Only ≈150 MB of the compressed data (44 GB) was wasted in the entire cluster. Through a reorg ≈50 MB could be saved per node. But

the same measurements with a customer-specific data set of 300 GB raw data (65 GB compressed) using a larger cluster have shown that those overhead for trailing cell blocks accumulated to several 100 MB per node. Merging those cell blocks reduced the wasted space to just a few MB per node.

These differences result from differing data distributions in fact and dimension tables. The TPC-DS data set contained most of its data in the fact table and only a small number of cells result from the partitioning algorithm for both, fact and dimension tables. In contrast, the customer data set had larger dimension tables with a large number of cells through a widely spread column value distribution.

Since fact tables are distributed in the current ISAO implementation, each node stores only its trailing cell block per cell. A reorganization therefore could only save at most $N-1$ blocks per cell in case where N denotes the number of cluster nodes, each node has data for the cell, and all these blocks can be merged into a single one. For dimension tables the reorg effect was much larger because their blocks are replicated onto each node. First, the data is distributed among all nodes and each of them processes its part in parallel. This results in one trailing block per node for each cell containing data. After this step, the blocks are broadcasted in the cluster and loaded into main memory for being able to process collocated joins. Through the replication in the worst case, there are N^2 partially filled blocks per cell in the cluster which could be reduced to $O(N)$ after reorganization.

To determine whether this cluster reorganization is beneficial, the number of partially filled blocks can easily be tracked after loading the data finishes and can be adjusted incrementally after each update. The reorg can be triggered automatically after a certain threshold is exceeded or manually when more memory is required e.g. for offloading another mart to ISAOPT.

6. CONCLUSION

In this paper, we have presented an approach on online reorganization in the ISAOPT MMDBS. Our work addresses the problem of repairing degenerated data structures caused by updates by recompression and garbage collection without disrupting query performance. We achieve this by implementing snapshot semantics – eliminating the need for an explicit synchronization between queries and concurrent updates. The results of our experimental evaluation have shown the benefit of the reorganization in terms of better memory utilization and improvement query runtimes as well as the low overhead for reorganization.

While we focused in this paper on how to reorganize, an important question is also when and what to reorganize. We plan to address this issue in our ongoing work by integrating an appropriate decision model.

7. REFERENCES

[1] K. J. Achyutuni, E. Omiecinski, and S. B. Navathe. Two techniques for on-line index modification in shared nothing parallel databases. In *SIGMOD*, 1996.

[2] I. Ahn and R. Snodgrass. Performance evaluation of a temporal database management system. *SIGMOD Rec.*, 15(2):96–107, 1986.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[5] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, 2001.

[6] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. In *SIGMOD*, 1988.

[7] CURSOR Software AG. DB2 Newsletter. Technical report, CURSOR Software AG, August 2008.

[8] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.

[9] T.-l. Hu, G. Chen, X.-y. Li, and J.-x. Dong. Automatic relational database compression scheme design based on swarm evolution. *Journal of Zhejiang University - Science A*, 7(10):1642–1651, 2006.

[10] IBM Corp. *DB2 Version 9.5 for Linux, UNIX and Windows English manuals*. IBM Corp., April 2009.

[11] S. Idreos, R. Kaushik, V. R. Narasayya, and R. Ramamurthy. Estimating the compression fraction of an index using sampling. In *ICDE*, 2010.

[12] C. S. Jensen. Vacuuming in TSQL2. commentary, TSQL2 Design Committee, Sept. 1994.

[13] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *VLDB*, 2008.

[14] A. Koeller and E. A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views in SchemaSQL. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1096–1111, 2004.

[15] V. M. Markowitz and J. A. Makowsky. Incremental reorganization of relational databases. In *VLDB*, 1987.

[16] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, 2008.

[17] D. W. Randall Davis. IBM BladeCenter HS22 Technical Introduction. Redpaper, IBM Corp, 2009.

[18] G. H. Sockut and R. P. Goldberg. Database reorganization—principles and practice. *ACM Comput. Surv.*, 11(4):371–395, 1979.

[19] G. H. Sockut and B. R. Iyer. Online reorganization of databases. *ACM Comput. Surv.*, 41(3):1–136, 2009.

[20] K. Stolze, F. Beier, K.-U. Sattler, S. Sprenger, C. C. Grolimund, and M. Czech. Architecture of a Highly Scalable Data Warehouse Appliance Integrated to Mainframe Database Systems. In *BTW*, 2011.

[21] K. Stolze, V. Raman, R. Sidle, and O. Draese. Bringing BLINK Closer to the Full Power of SQL. In *BTW*, 2009.

[22] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB*, 2005.

[23] TPC. TPC BENCHMARK DS. Standard, Transaction Processing Performance Council, 2007.

[24] S. B. Yao, K. S. Das, and T. J. Teorey. A dynamic database reorganization algorithm. *ACM Trans. Database Syst.*, 1(2):159–174, 1976.

[25] M. Zukowski, P. A. Boncz, N. Nes, and S. Heman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, June 2005.