Efficient Bulk Deletes in Relational Databases

A. Gärtner [1] A. Kemper [1] D. Kossmann [2] B. Zeller [1]

[1] Universität Passau
94030 Passau, Germany
⟨lastname⟩@db.fmi.uni-passau.de

[2] Technische Universität München
81667 München, Germany
kossmann@in.tum.de

Abstract

Many applications require that large amounts of data are deleted from the database – typically, such bulk deletes are carried out periodically and involve old or out-of-date data. If the data is not partitioned in such a way that bulk deletes can be carried out by simply deleting whole partitions, then most current database products execute such bulk delete operations very poorly. The reason is that every record is deleted from each index individually. This paper proposes and evaluates a new class of techniques to support bulk delete operations more efficiently. These techniques outperform the "record-at-a-time" approach implemented in many database products by about one order of magnitude.

1. Introduction

Sometimes we are confronted with more data than we can really use, and it might be wisest to forget and to destroy most of it. *(Donald Knuth, The Art of Computer Programming)*

For many companies, the cost to administrate their databases has increased dramatically in the last couple of years. This trend can be observed even though most database systems have become easier to use and database system administrators have gained more and more experience in using the systems. The reason for this cost explosion is that the sizes of databases grow at a dramatic rate. In 1999, only a few SAP R/3 databases were larger than one terabyte. It is estimated that this number will rise dramatically within the next two years.

The obvious way to limit the size of a database is to make use of archiving. That is, data which are not needed for every-day operations are demoted from the database (disks) to tertiary storage (tapes). This way the database can carry out its every-day operations efficiently and the data remains available for possible reuse. Archiving for this purpose is used, for instance, in the SAP R/3 installation of the German Telekom. In a more general context, archiving is studied as part of the SAP Terabyte project in cooperation with the University of Passau.

Archiving is a two step process. In the first step, the data to be archived are extracted from the database and written to tape. This first step involves the execution of (fairly complex) queries such as "find all orders which were processed more than three months ago". Query processing has been studied in a number of previous projects (e.g., [3]) and is not the subject of this work. In the second step, the extracted data are deleted from the database. How to efficiently delete such masses of data is the subject of this work.

In addition to archiving, there are several other applications that involve bulk deletes. For example, bulk deletes occur frequently in a data warehouse that keeps a *window* of, say, all the sales information of the last six months. The techniques presented in this paper can also be applied to speed up `UPDATE` statements; for instance, increasing the salary of above-average Employees involves carrying out a bulk delete (and bulk insert) on the *Emp.salary* index.

1.1. The State of the Art

Bulk deletes can be implemented very efficiently if the data is partitioned accordingly. For example, if we know that we are going to delete all sales information from the first quarter in September, all sales information from the second quarter in December, and so on, then we can store the sales information of each quarter in a separate partition. In this case, the bulk deletes can be implemented very efficiently by simply discarding a whole partition (including all the indices of that partition). Partitioning a database for this purpose is supported by most database products; e.g., Oracle [11]. However, partitioning only helps if the bulk deletes are planned before the data is actually created and a table can only be partitioned along one dimension. That is, partitioning will not help if, say, some bulk deletes are carried out according to the *order date* and some bulk deletes are carried out according to the *ship date*. Furthermore, partitioning does not help if there are additional restrictions; e.g., "delete old orders but only if they have been fully processed". If the data is not partitioned right, then the traditional way of deleting tuples perform very poorly on bulk deletes.

To see how the delete operations are carried out tradition-

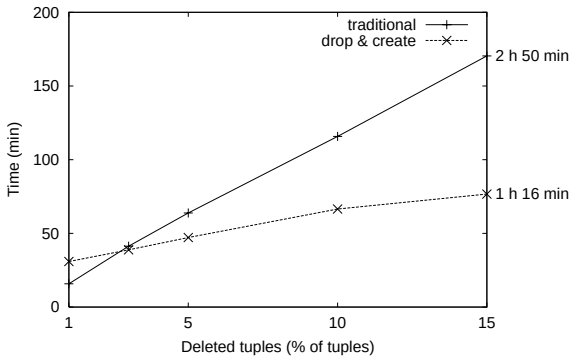

Figure 1. Bulk deletes using a commercial RDBMS: 500 MB table, 3 indices, vary number of deleted tuples

ally if the data is not partitioned accordingly, let us look at the following simple SQL statement in order to delete tuples from table R. $R.A$ is the key of table R and $D.A$ contains the keys of all records of R that are to be deleted. For instance, table D could be created as part of the first step of an archiving process. We will use this example throughout this paper, but of course the techniques devised in this work can be applied to any (bulk) DELETE statement and to any kind of application.

```
DELETE FROM R WHERE R.A IN (SELECT D.A FROM D)
```

Traditionally, this statement is carried out by scanning through table R, testing the predicate for each record of table R, and then deleting each tuple of R for which the predicate holds. Most importantly, whenever a record is deleted from table R, it is immediately removed from all indices (B-trees) of table R. As a consequence, for every record, each B-tree is traversed individually from the root to the relevant leaf resulting in overall very high costs.

Figure 1 shows the results of performance experiments that we carried out using one of the major relational database products. We ran our example statement from above using synthetic instances of tables R and D. Table R had 1.000.000 records of 512 bytes before the DELETE statement was carried out. We varied the size of table D such that 1%, 5%, 10%, or 15% of the records of table R were deleted. There were three indices on table R: one primary index and two secondary indices. The hardware was a SUN UltraSPARC 1 Creator 170E with 256 MB memory and a 4.55 GB Seagate Barracuda 4XL disk, which spins at 7200 rpm. Looking at the *traditional* graph, we observe that the running time of the DELETE statement increases sharply the more records of R are deleted. It takes almost three hours to delete 150.000 records (i.e., 15%) even for this tiny database.

Figure 1 also shows the running time to delete records from table R using an alternative approach (referred to as *drop & create*). In this approach, we first dropped the secondary indices of R and re-created them after the DELETE statement was executed. Due to the high cost to re-create the indices, this approach has a very high running time, too, but in fact, this approach outperforms the *traditional* approach significantly if more than 5% of the records of table R are deleted.

The goal of this work is to devise new ways to carry out bulk delete operations that significantly outperform the traditional ways of database systems to carry out DELETE statements. The two key ideas are to take a more holistic approach to optimize such DELETE statements and to delete records from indices in a batch rather than a record at a time.

1.2. Related Work

To the best of our knowledge, an algorithm for the implementation of bulk deletes—aside from partitioned tables—has not been published yet. However, there has been work on the analysis and implementation of delete operations from indices and a number of other related issues. Probably the best known algorithm to delete records from a B^+-tree was proposed by Jannink [7] and an improvement to this algorithm was proposed by Olivie and Maelbrancke [13]. Shasha and Johnson studied the overall performance of B^+-trees in the presence of deletes and inserts [8]: their conclusion is that leaf pages should not be merged after deletions. Bulk loading, the dual of bulk deletes, was studied in a number of papers. Seeger et al., for instance, propose an algorithm to bulkload generalized search trees [22]. Furthermore, Wiener and Naughton show how object-oriented databases can be loaded efficiently [24, 25]; they propose to carry out a *join* between the table of all OIDs and the table of all objects in order to initialize inter-object references in the object-oriented database. Deleting records from tables and the management of free space is described in [6, 14].

There also has been a line of work on creating and reorganizing indices while other (update, insert, delete) operations are carried out without blocking those operations; e.g., [17, 26, 21]. We will discuss that work and show how it can be applied in our context in Section 3.

1.3. Overview

The remainder of this paper is organized as follows. Section 2 presents new algorithms to carry out bulk deletes. This section describes the overall idea to process deletes in a set-oriented way rather than a record-at-a-time and it gives several examples. Section 3 discusses concurrency control and recovery issues in the context of bulk deletion. Section 4 contains the results of performance experiments that show that our new approach to carry out bulk deletes significantly outperforms traditional ways in various different situations. Section 5 contains conclusions and our plans for future work.

2. Algorithms for Bulk Deletes

In this section, we will show how bulk deletes can be processed efficiently by a relational database system. We will first present the general idea and then discuss several examples. Throughout this work, we will assume that B^+-trees are the only kind of indices supported and, thus, we will focus on showing how bulk deletes can be carried out in the presence of B^+-trees. We will study bulk deletes for other kinds of index structures (e.g., hash tables, grid files, and R trees) as part of future work.

2.1. Overall Approach

Processing a DELETE statement involves deleting records from the base table as well as from all indices of that table. Furthermore, referential integrity constraints from other tables must be tested. In a traditional system, the whole DELETE process is *horizontal*: whenever a tuple is deleted from the base table, it is immediately deleted from all indices before the next record is deleted. In other words, records are deleted one-at-a-time and from each index individually.

In order to implement bulk deletes more efficiently, we propose to carry out bulk deletes *vertically*. That is, we propose to delete all records from the base table first, to delete all records from the first index next, and so on. The most important advantage of such a vertical way to process bulk deletes is that the *physical* structure of the base table and indices can be taken into account. For instance, it is possible to sort the records before deleting them from an index in order to avoid random disk I/O. In addition, as we will see, a number of additional optimizations become possible if the records are deleted from the indices using such a vertical approach. Furthermore and for the same reasons, integrity constraints can be processed more efficiently using a vertical approach. Since the same mechanisms are applicable for processing integrity constraints with the help of indices as for the deletion of records from indices, we will ignore the processing of integrity constraints in the remainder of this paper.

To delete a set of records from a base table or an index, we introduce a new (logical) *bulk delete operator*, represented by a $\downarrow\bowtie$ symbol. The bulk delete operator takes two inputs: (1) a table or index, and (2) a list of records containing the keys or $RIDs$ (row identifiers)[1] of the records that are to be deleted. The bulk delete operator carries out two steps. In the first step, the records that are to be deleted are located in the base table (or index) and removed from the table (or index). In the second step, the table (or index) is reorganized; for instance, empty pages are reclaimed or two nodes of a B^+-tree can be merged (Section 2.3). The output of the $\downarrow\bowtie$ operator is a list of deleted records; this way the output of one $\downarrow\bowtie$ operator can serve as the input of another $\downarrow\bowtie$ and we generate *plans* with several $\downarrow\bowtie$ operators - one for each index and one for the base table - in order to execute bulk deletes.

From this discussion it should have become clear that the $\downarrow\bowtie$ operator is closely related to the relational *join* operator. In fact, the $\downarrow\bowtie$ operator carries out a *pointer based join* [19, 1] in order to find the records of the base table or index which should be deleted. Accordingly, the $\downarrow\bowtie$ operator can be implemented in many different ways and the best choice is made by the query optimizer depending on the size of the table/index, the number of records to be deleted, and the size of the main memory buffer pool. In all, the optimizer faces the following decisions:

1. $\downarrow\bowtie$ **method**: just like the ordinary relational join, the $\downarrow\bowtie$ operator can be carried out using many different techniques including nested-loops, merging, and hashing. However, it should be noted, that not every join method is applicable because records must be deleted from one of the "join" arguments: for instance, an adapted version of the grace or hybrid hash join which would partition the table/index is not viable. Only those join algorithms are applicable that operate on the original data pages ("in place") of the base relation or leaf node pages of the indices—not on copies generated during partitioning or sorting. This restriction is obviously necessary for deleting the tuples (or index entries) from their home pages.

2. $\downarrow\bowtie$ **order**: the optimizer must decide which indices to consider first and at what point the records should be deleted from the base table. As we will see, the optimizer must also decide which of the two inputs of a $\downarrow\bowtie$ operator is to be considered as the *inner* and *outer*. Again not every join order is applicable for the above mentioned reason.

3. primary $\downarrow\bowtie$ **predicate**: an index can be seen as a list of $\langle$*key, RID*$\rangle$ records. Given a list of keys and $RIDs$ of records to delete, the entries of the index can be looked up either by their key (and their RID to distinguish duplicate keys) or by their RID. Looking up entries in an index by their $RIDs$ might sound counterintuitive, but as we will see, sometimes this approach makes indeed sense in order to implement bulk deletes. Also, the choice of the primary $\downarrow\bowtie$ predicate (key or RID) can impact the choice of the $\downarrow\bowtie$ method and order.

Being aware of all these options, it is quite straightforward to extend an existing optimizer to make these decisions; for example, a query optimizer based on dynamic programming [4, 12] can easily be extended for this purpose. We will present and discuss the tradeoffs of three viable plans in the next subsection.

2.2. Examples

To demonstrate alternative ways to execute bulk deletes, we will go back to the example of the introduction. Table R from which records are supposed to be deleted is shown in Figure 2. Among others, table R has three attributes A,

[1] A RID can be thought of as a pointer to a record of a base table. Typically, such a RID is composed of a file or volume number, page number, and a slot number. However, to keep the examples readable the $RIDs$ in our examples consist only of a page number X and a slot number Y, e.g., 4.2.

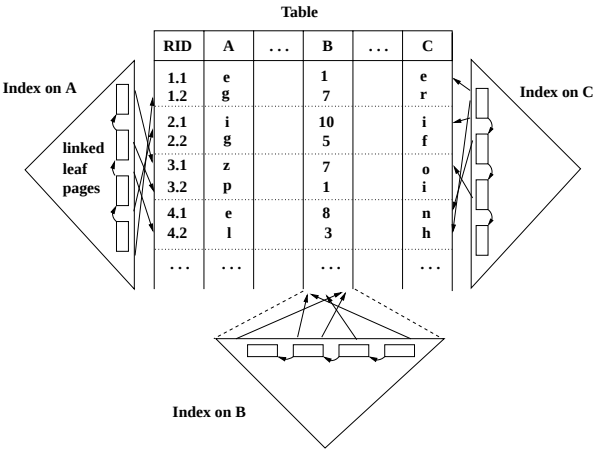

Figure 2. Example for a relation R with 3 indices

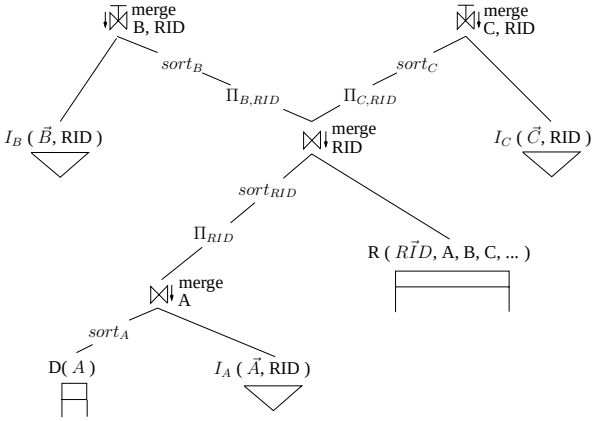

Figure 3. Using sorting and merging

B, and C and three indices I_A, I_B, and I_C on these attributes. As mentioned earlier, we assume that all indices are B^+-trees [23] in which all $\langle$ *key, RID* $\rangle$-entries are in the leaf pages. The inner nodes of the B^+-tree contain only reference keys for navigation purposes—no real data values. Furthermore, we assume a B-link-tree organization [10]; in a B-link-tree, the leaf nodes are chained in order to facilitate sequential scanning at the leaf node level. Records in table R are uniquely identified (and located) via a RID value which contains the page ID and the slot number of the record within that page. In addition, we assume that a second relation, called D, exists; D stores the A-values of all records that are supposed to be deleted from R. In summary, we have the following storage structures:

- $R(R\vec{I}D, A, B, C, ...)$
  $R\vec{I}D$ indicates, that the relation R is clustered (i.e., sorted) on RID values.
- $D(A)$
- $I_A(\vec{A}, RID)$
  $\vec{A}$ indicates that the leaf pages of the index are clustered on A.
- $I_B(\vec{B}, RID)$
- $I_C(\vec{C}, RID)$

We will also use the query from the introduction, but the discussion is relevant for any kind of bulk delete statement. (Any kind of SQL sub-query could be used as part of the DELETE statement.)

```
DELETE FROM R WHERE R.A IN (SELECT D.A FROM D)
```

Using our new bulk delete operator $\bowtie\!\downarrow$ such a DELETE operation corresponds to the following logical plan: $D \bowtie\!\downarrow I_A \bowtie\!\downarrow R \bowtie\!\downarrow I_B \bowtie\!\downarrow I_C$. In the following, we will discuss alternative ways to execute this DELETE plan efficiently: the first approach is based on sorting, the other approaches make use of hashing and partitioning. As mentioned in Section 2.1, many more possibilities are conceivable (e.g., considering different $\downarrow\!\bowtie$ orders) and the choice which option to use must be made by the query optimizer. We will also discuss special cases such as clustered indices and unique indices. Compound indices on several attributes can be treated just like indices on a single attribute. However, we will ignore integrity constraints in the following examples; as mentioned in the previous subsection, integrity constraints can also be processed more efficiently using a vertical approach. We propose to check integrity constraints in such a vertical way as early as possible and before deleting records from the table and the indices so that no work needs to be undone if an integrity constraint fails.

2.2.1. Bulk Deletes by Sorting and Merging

Figure 3 shows how sorting can be used to implement bulk deletes. The arrow in the $\bowtie\!\downarrow$ operator points to the table or index from which records are deleted (left or right). In this example, the table or index is always the *inner* which is indicated by the use of a $\bowtie\!\downarrow$ symbol; for layout reasons, however, the bulk delete from index I_B is shown with a $\downarrow\!\bowtie$ symbol although I_B is the inner of this bulk delete operation. The two top-level $\bowtie\!\downarrow$ operators produce no output (indicated by $\overline{\bowtie}$); the output of the other $\bowtie\!\downarrow$ operators is piped into other relational operators (i.e., projections in Figure 3). The plan of Figure 3 is based on sorting and merging in order to implement bulk deletes. That is, in every step the list of keys (or $RIDs$) of the records that are to be deleted is sorted according to the clustering of the table or index. The tables and indices on which the bulk deletion is carried out and which are typically much larger than the list of keys need not be sorted. Also, we can apply projections before each sort in order to minimize the volume of data that needs to be sorted. Carrying out bulk deletes in this way is usually much better than the traditional (horizontal) approach to carry out deletes because sorting avoids random disk I/O to locate and delete the records; in other words, we use sorting in order to adjust the list that specifies the records to delete to the physical layout of the table and indices. Essentially, this plan is often attractive just as sort/merge joins are often more attractive

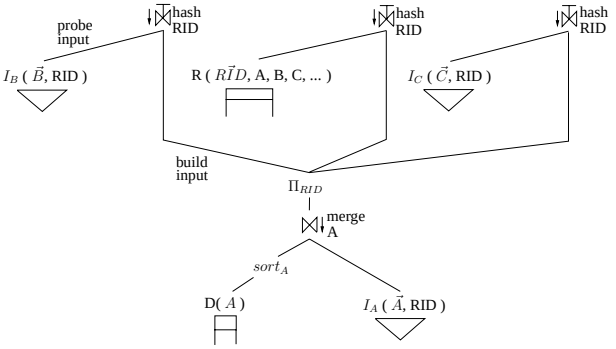

Figure 4. Using hashing

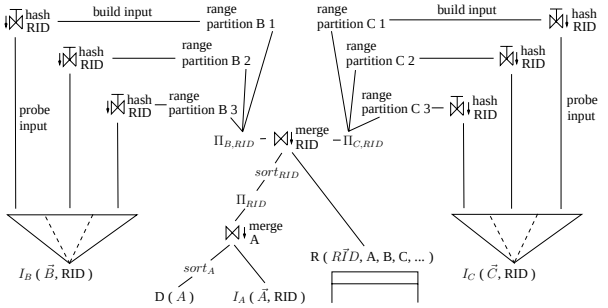

Figure 5. Using hashing and range partitioning

than nested-loop joins. Again, it is also important to keep in mind that only the (small) lists of keys and $RIDs$ need to be sorted.

The plan of Figure 3 is not a tree, it is a graph. The result of the bulk delete operation with table R is split and piped into two different query trees in order to carry out bulk deletes for the indices I_B and I_C. That is, the result of the "join" of D, I_A and R is a common subexpression for the subsequent bulk delete operations on the other indices. Splitting output streams in a query plan has also been used in different contexts; i.e., in [2]. Here, it makes it possible to process the bulk deletes on I_B and I_C independently and in parallel.
In the following, we will discuss special cases that impact the way bulk deletes are carried out:

Clustered Index I_A
If I_A is a clustered index, then we need not sort the list of $RIDs$ because the result of the first $\bowtie\downarrow$ operation (i.e., deleting records from I_A) is already sorted by RID in Figure 3. In this case, the sort/merge approach to carry out bulk deletes becomes even more attractive. This is analogous to *interesting orders* which make sort/merge joins more attractive in regular join processing [5].

Clustered Index I_B (or I_C)
Analogously, if I_B (or I_C) is a clustered index, then we can save the *sort*$_B$ (or *sort*$_C$) operators in the plan of Figure 3 because an order on RID implies an order on B (or C) in this case.

Unique Indices
For the plan of Figure 3 this constraint has no direct effect. However, it does have an effect on concurrent update/insert transactions which need to consult the index to enforce the uniqueness constraint. Such transactions cannot proceed while the unique index is off-line. We will revisit this issues in Section 3.

2.2.2. Bulk Deletes by Hashing

As an alternative to sorting, hashing can be used in order to detect which records should be deleted. Hashing is particularly attractive if the list of keys and $RIDs$ that specify which records should be deleted fits into main memory; in fact, it is only necessary that the $RIDs$ (without any keys) fit into main memory. Figure 4 shows how hashing can be used to carry out bulk deletes.

As in the plan of Figure 3, sorting is used in order to delete the records from index I_A and to find the $RIDs$ of all records that should be deleted. This RID list is then piped into three hash-based $\downarrow\bowtie$ operators that are used to delete the records from table R and indices I_B and I_C. More precisely, a main-memory hash table is constructed from the RID-list and the leaf pages of the indices and all pages of table R are scanned and the RIDs of each record is probed with the hash table in order to see whether the record should be deleted. Probing can be done independently and in parallel for each index and table R, but on a single-processor machine the same hash table can be used. This approach corresponds to the *classic hash join* of [18] and is particularly attractive if the hash table really fits into physical main memory.

If the RID list is very large and the size of the hash table exceeds the size of the available main memory, then range partitioning can be applied. Figure 5 shows an example that demonstrates how range partitioning and hashing can be used to implement the bulk deletes for indices I_B and I_C. For the range partitioning phase, the key values of B and C of all records that should be deleted need to be known so that range partitioning can only be applied after the bulk delete for table R has been carried out. The idea is to partition the RID-list into partitions that fit into main memory and then carry out the bulk delete for each partition individually using the main-memory hash-based approach described above. In Figure 5, three partitions are created for each bulk delete. I_B and I_C can be range partitioned without any cost because index I_B is clustered (ordered) by B and I_C is clustered by C.
It is of course possible to mix hash-based and sort-based $\downarrow\bowtie$ operators in many more ways in a single plan. For instance in the plan depicted in Figure 5, sorting and merging (instead of hashing) could be used to process table R.

2.3. Reorganization

One important advantage of our approach to carry out bulk deletes is that the B^+-trees can be reorganized with very little extra cost. In all three plans described in the previous section, the leaf pages are scanned from the beginning to the

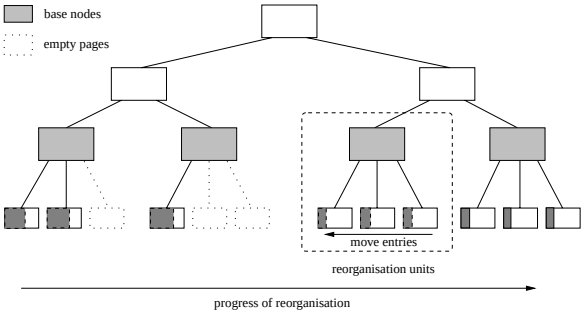

Figure 6. Reorganization of a B^+-tree during bulk deletion

end; therefore, leaf pages can be *compacted*, *compressed* and merged with neighbor pages (i.e., clustered) during the bulk delete. The inner nodes of the B^+-tree can be updated (and reorganized) *after* or *while* the leaf pages are processed. One way to update and reorganize the inner-nodes afterwards is to process each layer individually (a full-fledged B-link tree organization is necessary for this approach). An alternative is to update the inner-nodes on the fly by adapting the algorithm presented in [26] as shown in Figure 6. The idea is to identify one inner-node as a base node; typically, such a base node will be chosen in such a way that the sub-tree rooted by the base node fits into the available main memory buffers. First, bulk deletion and reorganization for the leaves of this sub-tree is carried out and the inner-nodes of this sub-tree are updated. Then the next base node (i.e., the right sibling of the previous base node) is chosen and the leaves and inner-nodes of that sub-tree are processed. At the end, if necessary, the nodes of the "base-node" level and of higher levels of the tree are reorganized in the same fashion. Since table R is also scanned from the beginning to the end in all our example plans, table R could also be reorganized while processing the bulk deletion for table R. Reorganizing table R, however, involves updating the entries of all indices because the $RIDs$ of (almost) all records of R change. In the plan of Figure 3, for instance, we would have to update most entries of index I_A after the bulk deletes for table R have been carried out if table R is reorganized as part of the bulk delete; as a result, index I_A would be processed twice.

Note that the procedure sketched in Figure 6 may generate some "holes" in the storage area of the leaf nodes. In order to retain (or generate) a contiguous storage area for the entire set of leaf nodes it is also possible to shift all entries "to the left"—beyond base node delimiters.

3. Concurrency Control and Recovery Issues

In this section we will briefly address concurrency control methods that have to be applied if the bulk deletion process runs concurrently with other transactions accessing and modifying the same relation and its indices. We will also briefly outline the recovery methods to guarantee the fault tolerance and restartability of a bulk deletion process.

3.1. Concurrency Control Issues

So far we have described the bulk deletion process without consideration of other, concurrent transactions accessing the same relation and/or indices. That is, we assumed that the bulk deletion process has obtained exclusive access on the entire relation R as well as on its indices. Even though our bulk deletion approach is very efficient—typically up to an order of magnitude faster than the traditional approach as we will show in the performance experiments of Section 4—it may still be beneficial to allow concurrent transactions while bulk deletion is still in progress.

However, we see no benefits in allowing concurrent transactions while the bulk deletion processes the base table R: bulk deletion is employed if a vast number of tuples is removed from R. Therefore, database systems employing lock escalation would switch to an exclusive lock on the base table, anyway. Other database systems will set a very large number of locks leading to a very severe number of conflicts (deadlocks) with other transactions. Therefore, our bulk deletion process locks table R exclusively and switches all indices on R off-line.

As soon as table R and all unique indices are processed (and the bulk deletion is committed) the lock on R is released and the unique indices are brought on-line in order to allow other concurrent read and update transactions on R. However, the indices without a unique constraint remain off-line while the bulk deletion process propagates the deletions to the index structures. This increased concurrency is facilitated by our vertical processing technique that propagates deletes to the table and the indices separately. This vertical approach allows us also to process the indices with a uniqueness constraint first. Processing the unique indices first is necessary to ensure that the uniqueness constraint isn't violated. Trying to ensure the uniqueness constraint while the unique index is off-line can lead to inconsistencies because no locking is possible. Furthermore it is difficult to decide at the time an insertion is made whether a conflict exists or not because a potentially conflicting entry may be deleted by the bulk deletion later on. However, we will study if the restriction of processing the unique indices first can be relaxed or not as part of our future work.

Of course, the off-line indices cannot be used as access paths or for predicate locking before the deletes have been installed. Update transactions modifying base relation R have to do extra work in order to guarantee the consistency of indices after the whole bulk delete is finished. Two approaches are possible, which are derived from Mohan and Narang's work on online index creation [17]:

- *Side-file*: Update transactions log their changes to R in side-files which are propagated to the indices.
- *Direct propagation / no side-file*: The updates are directly installed in the indices while bulk deletion processing is still active.

3.1.1. Side-File

When a side-file is used the remaining indices I_B and I_C aren't changeable directly by any other transaction. In this case, all changes made to this indices by a updater transaction are logged in side-files (one for each index). When the bulk deletion has processed an index the side-file is applied to the index but still the index is off-line and still other transactions can append the side-file. When nearly the whole side-file is processed, the bulk deletion quiesces all updates to the index, processes the last entries of the side-file and brings the index on-line again. This approach is also described in [20].

The side-file approach has the advantage that no latches on index pages during bulk deletion are necessary (except for unique indices, see below) and it is easy to implement.

3.1.2. No Side-File / Direct Propagation

When no side-file is used the changes are propagated directly to the off-line indices. Therefore the bulk deleter as well as the updater transactions have to set latches on the index pages in order to avoid conflicts. However, latching the pages is not enough. To avoid conflicts between updater transactions and the bulk delete transaction an inserted entry $\langle$ `k,RID` $\rangle$ has to be marked as undeletable. This will prevent the following race condition: the bulk deleter may have the corresponding RID in its delete-set. This RID may have been re-used by the database management system for an insert.

However, an undeletable entry can be removed as part of rollback processing for the transaction that inserted it.

3.1.3. Index Processing Order

There exist many possible orders in which the indices can be processed due to the vertical approach of our new bulk delete algorithm. Therefore indices which are critical for the performance of applications can be processed first while the processing of non-critical indices can be delayed. Especially the unique indices can be processed first.

3.2. Checkpoints and Recovery

We propose to make use of checkpoints to minimize the loss of work during a system failure. A checkpoint could be established at any time by flushing all pages to stable storage with a LSN equal or less to the actual one. Additionally the last processed RID or key-value respectively can be stored in the log. This will speed up recovery because the storage structures are clustered regarding to $RIDs$ and key-values, respectively. So the already processed values can easily be recognized. Also the results of the join variants described in Section 2 should be materialized to stable storage. Checkpoints are especially advisable when the processing of one structure (R, I_A, I_B, or I_C) is finished.

To take full advantage of checkpointing and to save the work done even after a system failure we propose to finish the bulk deletion instead of rolling it back as done during traditional recovery [15, 16]. Further attention is needed when side-files are used. If the bulk deletion is finished during recovery the side-files are appended. The side-files are applied to the indices when the bulk deleter has finished. This is necessary because the changes logged in the side-files were made by transactions that were triggered after the commit of the bulk deletion and have therefore to be made durable after the bulk deletion changes to avoid inconsistencies.

4. Performance Experiments and Results

In this section, we present the results of performance experiments that show that our *vertical* approach to implement bulk deletes outperforms the traditional *horizontal* approach if a significant portion of a table is deleted. We will only present results that were obtained using sorting and merging to implement the $\downarrow\bowtie$ operator. The tradeoffs between hashing and sorting for bulk deletes are the same as for regular joins [5], and the differences in performance are much smaller than the differences between the horizontal and vertical approach.

4.1. Benchmark Environment

To study the performance of bulk deletes, we used a prototype database system implemented on top of a UNIX file system. The implementation of the B^+-trees is based on the code developed by Jan Jannink [7] so that the deletes in the traditional, horizontal approach were carried out as best as possible using Jannink's algorithm. However, we adapted the B^+-tree so that the nodes in each level are linked (B-link tree [10]) because this organization was necessary in order to implement our horizontal bulk delete approaches (Section 2). The whole prototype was implemented in the C++ language.

Our prototype is installed on a SUN Ultra 10 workstation with a 333 MHz SPARC processor and a 9.1 GB Seagate Medialist Pro hard disk. The operating system is Solaris 2.7. The size of the main memory is 128 MB; however, if not reported otherwise, our prototype uses only 10 MB of main memory The traditional algorithm uses this main memory as input/output buffers in order to read chunks of several pages from disk (i.e., chained I/O) or to cache pages of indices and/or base tables. The bulk deletion algorithm uses this main memory not only for caching but also to carry out sorting. The page size for tables and indices is 4096 bytes and we use Solaris' direct I/O feature in order to avoid caching effects of the operating system.

The database consisted of one table *R* with eleven attributes *A,B,...,K*. In all experiments, table R has initially 1.000.000 tuples, each of size 512 bytes. The first 10 attributes are random integers and the last attribute (i.e., K) is a string field containing garbage data for padding. Each attribute is free of duplicates because Jannink's B^+-tree implementation does not support duplicates. However, the exis-

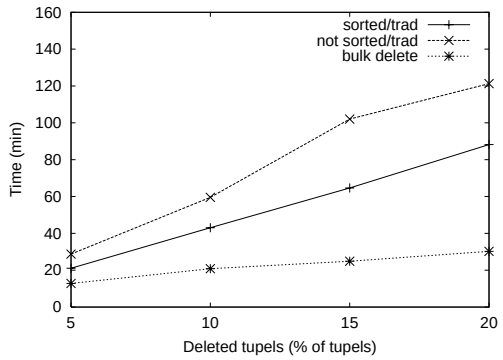

Figure 7. Running time [min], vary deletes, 1 unclustered index, 5 MB memory

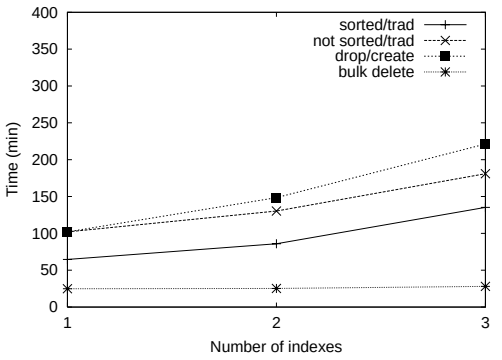

Figure 8. Running time [min], vary indices, uncl. indices, 5 MB memory, 15% deletes

tence of duplicates does not impact much the tradeoffs of our new bulk delete techniques.

If not stated otherwise, we only have one index on table R: I_A, an index on attribute A. The height of I_A is three and the inner nodes contain up to 512 entries. Also, we generate a table D with random A values and use the same DELETE statement that we have been using in all examples throughout this paper; i.e.:

```
DELETE FROM R WHERE R.A IN (SELECT D.A FROM D)
```

We executed this DELETE statement in isolation (i.e., no concurrency control effects) and measured its running time. We vary the size of table D so that 5% to 20% of the records of table R are deleted. Accordingly, the size of table D varies from about 200 KB to 800 KB, and table D can always be sorted in one pass in main memory. In some experiments, we also vary the number of indices and the size of the main memory buffers. Furthermore, we study scenarios in which I_A is clustered (i.e., table R is sorted according to attribute A) and situations in which I_A is not clustered.

Typically the height of all indices is three and the height of the indices does not change even if 20% of the records are deleted. As proposed in [9], we only reorganize and garbage collect an index page if it is totally empty (We do not apply the techniques proposed in Section 2.3). In all experiments, almost no reorganization is carried out because of the random distribution of the keys of the records that are deleted.

4.2. Experiment 1: Vary Number of Deleted Records

Figure 7 shows the running times of our sort-based (vertical) bulk delete approach compared to traditional (horizontal) approaches to carry out bulk deletes. Our sort-based bulk delete approach carries out a sort/merge-based bulk delete operation with index I_A first and then a sort/merge-based bulk delete operation with table R (Figure 2). The traditional approaches probe index I_A in order to find all records to delete, and then for each match immediately delete the corresponding record from table R and index I_A. We studied two versions of the traditional approach: (a) table D is sorted before the deletes are carried out (referred to as *sorted/trad*) and (b) table D is not sorted (referred to as *not sorted/trad*). Obviously, the *sorted/trad* version performs much better because it avoids random I/O while probing index I_A for each key stored in table D; this version roughly corresponds to the way the database product studied in the introduction carries out bulk deletes. The *drop & create* method proposed in the introduction is not studied because I_A is the only index in this experiment and I_A is vital to carry out the bulk delete operation using any approach.

Figure 7 shows clearly the importance to implement bulk deletes using one of our special horizontal approaches. Even the *sorted/trad* approach is clearly outperformed by our sort/merge-based horizontal approach. The differences become larger, the more records are deleted. If 20% of the records are deleted, our horizontal approach outperforms the *not sorted/trad* approach by almost one order of magnitude: half an hour compared to more than two hours.

4.3. Experiment 2: Vary Number of Indices

Figure 8 shows the running times of the alternative approaches if we vary the number of indices. In this series of experiments, always 15% of the records are deleted from table R. We observe that our horizontal approach to carry out bulk deletes becomes more important, the more indices exist. In the extreme case of this experiment (three indices), our horizontal approach takes only about half an hour whereas the traditional approaches take more than two hours (*sorted/trad*) and more than three hours (*not sorted/trad*), respectively. (These numbers are comparable to the results described in the introduction which were obtained using a commercial relational database system; see Figure 1.) If I_A is not the only index, then the *drop & create* approach makes sense, but as shown in Figure 8, the *drop & create* approach performs even worse than the traditional (vertical) approaches, here. (Apparently, creating indices is slower in our prototype than in the commercial database system used for the experiments of Figure 1.)

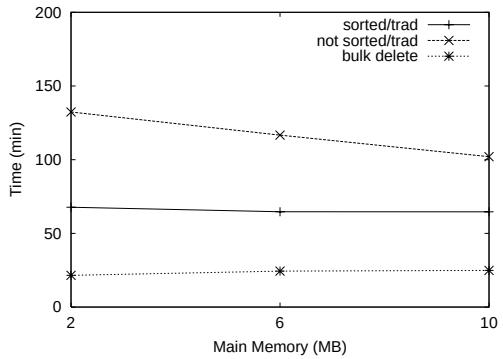

Figure 9. Running time [min], vary memory, 1 uncl. index, 15% deletes

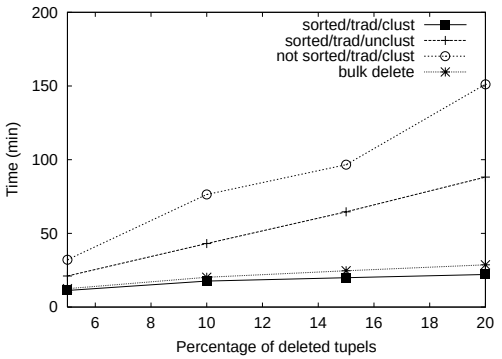

Figure 10. Running time [min], clust. index, 1 index, 5 MB memory, 15% deletes

4.4. Experiment 3: Vary the Height of the Index

We also tested the performance to carry out bulk deletes with different index heights (Table 1). To construct a version of index I_A with a larger height, we (artificially) increased the size of the keys in the inner nodes of I_A by allocating more space within a node for each key. More specifically, we store 100 keys per node in order to create an index I_A with height four; normally, we store up to 512 keys per node and index I_A has height three. As shown in Table 1, the running time of our bulk delete approach is almost independent of the height of the index. The running times of the traditional approaches, however, increase sharply with the height (size) of an index. Recall that the whole index must be traversed from the root to the corresponding leaf in order to delete a record using a traditional approach; instead our approach to implement bulk deletes directly operates on the leaf pages of an index.

Remember that the bulk delete algorithm sorts the delete set D before deleting the entries in the index and in this case the sorting can be done in main memory. Therefore sorting D before evoking the bulk delete algorithm has no significant impact on the running time of the bulk delete statement using our new bulk delete algorithm.

Table 1. Running time [min], 1 uncl. index, 15% deletes, 5 MB memory

	index height 3 (min)	index height 4 (min)
sorted/bulk	24,87	26,79
not sorted/bulk	24,87	26,79
sorted/trad	64,65	80,65
not sorted/trad	102,05	136,09

4.5. Experiment 4: Vary Size of Available Memory

Figure 9 shows the running times of the alternative approaches varying the size of the available main memory used for sorting and to cache pages of index I_A and table R. We observe that our new approach performs just as well if only very little main memory (2 MB) is available: all sorting can be carried out in one pass in main memory because only small lists of RIDs and keys need to be sorted, and the merging with table R or index I_A requires only very little main memory, too. On the other hand, the traditional approaches are more sensitive to the size of the main memory. In particular, the performance of the *not sorted/trad* approach depends strongly on the amount of available main memory: pages of the index and the base table are possibly used several times and the more memory is available, the more pages of the index and of the base table can be cached.

4.6. Experiment 5: Clustered Index

Figure 10 shows the running times if index I_A is clustered; i.e., table R is sorted by attribute A. As a baseline, Figure 10 also shows the runing times of the *sorted/trad* approach if I_A is not clustered. (These results are identical with those shown in Figure 7.) If index I_A is clustered and no other indices exist, this is the best possible case for the traditional approaches. In this case, the *sorted/trad* approach even outperforms our (vertical) approach to implement bulk deletes because no advantage can be achieved by sorting and by carrying out the deletes in a vertical way. The analogon is that index nested-loop joins perform very well if the index is clustered and the outer table is sorted accordingly. However, even in this extreme case and although it does not take advantage of the fact that the index is clustered and sorting is unnecessary, our bulk delete approach performs almost as well as the *sorted/trad* approach. The *not sorted/trad* approach benefits from the clustering of the index, but it shows overall very poor performance because of its high cost to probe the index in order to find the records to delete.

5. Conclusions and Future Work

In this paper, we proposed new algorithms for deleting masses of data in relational databases when B^+-trees are used for indexing. First, we described the traditional implementation of deletes; the traditional implementation is to take a *horizontal* record-at-a-time approach. Then we introduced a new and more holistic approach which deletes records from

tables and indices in a set-oriented way. We presented alternative ways to delete records in such a set-oriented way and showed that the bulk deletes can be optimized in similar ways as complex join queries: the join method, the join order, and the primary join predicates can be chosen depending on the size of the table, the number of records to delete and properties of the indices (e.g., uniqueness and clustering). The traditional, record-at-a-time way to execute bulk deletes corresponds to a plan in which nested-loops are used as a method for all joins and records are deleted from the base table first and then from the indices. We carried out performance experiments that showed that a plan which is based on sorting and merging and which uses a different join order outperforms the traditional way by up to almost one order of magnitude.

This work was restricted to B^+-trees; in our prototype, other kinds of indices are updated in the traditional way. In future work, we plan to generalize our approach and study algorithms to delete records in bulk from other index structures such as hash tables, R-trees, or grid files.

Acknowledgements

This work was partially supported by the German Research Council DFG under contract Ke401/7-1 and by SAP as part of the Terabyte project.

References

[1] R. Braumandl, J. Claussen, A. Kemper, and D. Kossmann. Functional join processing. *The VLDB Journal*, 8(3-4):156–177, 2000. Invited Contribution to the Special Issue "Best of VLDB 98".

[2] J. Claussen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Trans. Knowledge and Data Engineering*, 12(2):238–260, 2000.

[3] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[4] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, CA, USA, May 1987.

[5] G. Graefe, A. Linville, and L. Shapiro. Sort versus hash revisited. *IEEE Trans. Knowledge and Data Engineering*, 6(1):120–135, Mar. 1994.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1993.

[7] J. Jannink. Implementing deletion in B+-trees. *ACM Sigmod Record*, 24(1):33–38, Mar. 1995.

[8] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 235–246, Philadelphia, Pennsylvania, Mar. 1989.

[9] T. Johnson and D. Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *Journal of Computer and System Sciences*, 47(1):45–76, Aug. 1993.

[10] P. Lehman and S. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. on Database Systems*, 6(4):650–670, 1981.

[11] L. Leverenz, R. Mateosian, and S. Bobrowski. *Oracle8 Server – Concepts Manual*. Oracle Corporation, Redwood Shores, CA, USA, 1997.

[12] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, IL, USA, May 1988.

[13] R. Maelbrancke and H. Olivie. Optimizing Jan Jannink's implementation of B+-tree deletion. *ACM Sigmod Record*, 24(3):5–7, Sept. 1995.

[14] M. McAuliffe, M. Carey, and M. Solomon. Towards effective and efficient free space management. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 389–400, Montreal, Canada, June 1996.

[15] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Systems*, 17(1):94–162, Mar. 1992.

[16] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 371–380, San Diego, CA, USA, June 1992.

[17] C. Mohan and I. Narang. Algorithms for creating indexes for very large tables without quiescing updates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 361–370, San Diego, CA, USA, June 1992.

[18] L. Shapiro. Join processing in database systems with large main memories. *ACM Trans. on Database Systems*, 11(9):239–264, Sept. 1986.

[19] E. Shekita and M. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 300–311, Atlantic City, NJ, May 1990.

[20] G. Sockut, T. Beavin, and C. Chang. A method for on-line reorganization of a database. *IBM Systems Journal*, 36(3):411–436, 1997.

[21] V. Srinivasan and M. Carey. Performance of on-line index construction algorithms. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 580 of *Lecture Notes in Computer Science (LNCS)*, pages 293–309, Vienna, Austria, Mar. 1992. Springer-Verlag.

[22] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 406–415, Athens, Greece, Aug. 1997.

[23] H. Wedekind. On the selection of access paths in a data base system. In *IFIP Working Conference Data Base Management*, pages 385–398, 1974.

[24] J. Wiener and J. Naughton. Bulk loading into an OODB: A performance study. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 120–131, Santiago, Chile, Sept. 1994.

[25] J. Wiener and J. Naughton. OODB bulk loading revisited: The partitioned-list approach. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 30–41, Zürich, Switzerland, Sept. 1995.

[26] C. Zou and B. Salzberg. On-line reorganization of sparsely-populated B^+-trees. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 115–124, Montreal, Canada, June 1996.