Optimizing Maintenance of Constraint-based Database Caches

Joachim Klein Susanne Braun

Databases and Information Systems, Department of Computer Science,
University of Kaiserslautern, P. O. Box 3049, 67653 Kaiserslautern, Germany
jklein@cs.uni-kl.de s_braun@cs.uni-kl.de

Abstract. Caching data reduces user-perceived latency and often enhances availability in case of server crashes or network failures. Database caching aims at *local* processing of specific *declarative* queries in a DBMS-managed cache close to the application. Query evaluation must produce the same results as if done at the *remote* database backend, which implies that all data records needed to process such a query must be present and controlled by the cache, i. e., to achieve "predicate-specific" loading and unloading of such record sets. Using cache constraints, the cache manager applies maintenance operations to appropriate record sets, called caching unit. In this way, the cache manager can, at any point in time, guarantee "predicate completeness" of the caching units currently present. We explore how cache groups can be maintained to provide the data currently needed. For various cache configurations, we design and optimize loading and unloading algorithms for sets of records keeping the caching units complete. Furthermore, we empirically identify the costs involved for this kind of cache maintenance.

1 Motivation

Caching data in wide-area networks close to the application removes workload from the server DBMS and, in turn, enhances server scalability, reduces user-perceived latency and often enhances data availability in case of server crashes or network failures. Simpler forms of caching, e. g., *Web caching*, keep a set of individual objects in the cache and deliver – upon an ID-based request – the object, if present, to the user [3]. In contrast to Web caching, *database caching* is much more ambitious and aims at declarative, i. e., SQL query evaluation in the cache which is typically allocated near by an application server at the edge of the Internet. If the cache holds enough data to evaluate a query, it can save response time for the client request, typically issued by a transaction program running under control of the application server. If only a partial or no query result can be derived, the remaining query predicates have to be sent to the original data source, the so-called *backend* database, to complete query evaluation and to return the result back to the cache.

Hence, database caching substantially enhances the usability of cached data and is a promising solution for various important and performance-critical applications. However, *local* query evaluation must produce the same answer as if

done at the *remote* database backend, which implies that all data records needed to process a specific predicate must be present and controlled by the cache. In contrast to conventional caching, the cache manager has, at any point in time, to guarantee "predicate-specific" loading or unloading of such record sets. The simplest way to accomplish such a "completeness" is to cache the whole contents of frequently visited tables by full-table caching [12]. Because such a solution does not allow dynamic cache adaptations to respond to the actual query workload, more flexible approaches are needed. Some of them are based on materialized views (or refinements of those), but limited to them, i. e., they only support single-table queries. [2,4,6,8,9,10]. In contrast, *Constraint-based database caching* (CbDBC) uses specific cache constraints, by which the cache manager can guarantee completeness, freshness, and correctness of cache contents and support multi-table queries. These constraints equip the cache manager with "semantic" knowledge to take care of "predicate completeness" and achieve effective cache maintenance – prerequisites to enable correct and efficient query evaluation. So far, several database vendors have implemented such database caches based on similar ideas for their implementations [1,9,13].

Sect. 2 briefly repeats the cornerstones of DB caching. To characterize its maintenance tasks, Sect. 3 explains the key problems, outlines the measurement environment, and outlines results achieved by known algorithms. The performance weaknesses observed lead the new maintenance algorithms introduced in Sect. 4 and Sect. 5. After the presentation of our performance gains for the maintenance tasks, we conclude the paper in Sect. 6.

2 Constraint-based Database Caching

Constraint-based database caching (CbDBC) maintains a set of *cache tables* forming a *cache group* (CG), where appropriate constraints are defined to control its content. Valid states of the cache are accomplished as soon as all cache constraints are satisfied. But, they are continuously challenged, because existing cache data has to be updated (due to modifications in the backend), unreferenced cache records have to be removed to save needless overhead for consistency preservation, and new records enter the cache due to locality-of-reference demands. Based on the rules given by the cache constraints, the cache is able to decide which data has to be kept and which queries can be answered. A complete description of the concepts of CbDBC can be found in [7]. Here, we briefly repeat the most important concepts for comprehension.

Each cache table corresponds to a backend table and contains, at any point in time, a subset of the related records in the backend table. For ease of management, cache tables have identical column distributions and column types as the respective backend tables, however, without adopting the conventional primary/foreign key semantics. Instead, cache table columns can be controlled by unique (U) and non-unique constraints (NU, for columns with arbitrary value distributions). Those columns gain their constraining effect by the value-completeness property.

Definition 1 (Value completeness). *A value v is value complete (or complete for short) in a column $S.c$ if and only if all records of $\sigma_{c=v}S_B$ are in S.*

Here, S is the cache table, S_B its corresponding backend table, and $S.c$ a column c of S. Completeness of a value in a NU column requires loading of multiple records, in general, whereas appearance of a value in a U column automatically makes it value complete. Apparently, value completeness supports the evaluation of equality predicates on the related columns.

A further mechanism enables the evaluation of equi-joins in the cache. A so-called *referential cache constraint* (RCC) links a source column $S.a$ to a target column $T.b$ (S and T not necessarily different) and enforces for a value v appearing in $S.a$ value completeness in $T.b$. Therefore, values in $S.a$ are called *control values* for $T.b$.

Definition 2 (Referential cache constraint, RCC). *A referential cache constraint $S.a \rightarrow T.b$ from a source column $S.a$ to a target column $T.b$ is satisfied if and only if all values v in $S.a$ are value complete in $T.b$.*

Only records frequently referenced by queries, i. e., those having high locality in the cache, are beneficial for caching. Therefore, we have designed a special filling mechanism based on a so-called *filling column*, e. g., $T1.a$ in Fig. 1a. For filling control, we define for it a fill table $ftab(T1)$, an RCC $ftab(T1).id \rightarrow T1.a$, and a set $cand(T1)$ which contains the desired *candidate values* eligible to initiate cache filling.[1] Upon a query reference of value v listed in $cand(T1)$, e. g., by $Q = \sigma_{a=v}T1$, it is inserted into $ftab(T1).id$, if not already present, and, hence, called *fill value*. Via the related RCC, such a value implies value completeness of v in $T1.a$ and, therefore, loading of all records $\sigma_{a=v}T1_B$ into $T1$. To satisfy all cache constraints, RCCs emanating from $T1$ may trigger additional load operations. As a consequence of v's completeness, values – so far not present in their source columns (e. g., $T1.a$ and $T1.b$) – may have entered $T1$ and, in turn, imply completeness in their target columns. The newly inserted records in those target tables may again trigger – via outgoing RCCs (e. g., $T2.a$) – further load operations, until all cache constraints are satisfied again. An example of a cache group together with its constituting components is shown in Fig. 1a.

3 Cache Loading and Unloading

To describe cache loading in detail, we need some further terminology. For any RCC $S.a \rightarrow T.b$ defined in a cache group, the set of records to be (recursively) cached due to the existence of v in $S.a$, is called *closure* of v in $S.a$. Refer to Fig. 1b: Loading a record with value $T1.a$ = *'Jim'* to $T1$ implies to satisfy $T1.a \rightarrow T2.b$, which adds records *('p4711','Jim', ...; 'p4810', 'Jim', ...)* to $T2$. In turn, these records insert new values *'p4711'* and *'p4810'* to $T2.a$, which enforce satisfaction of RCC $T2.a \rightarrow T3.a$. Hence, the new values have to be made value

[1] In contrast, DBCache [1] uses the *cache key* concept, which implies caching of any value referenced in the related column.

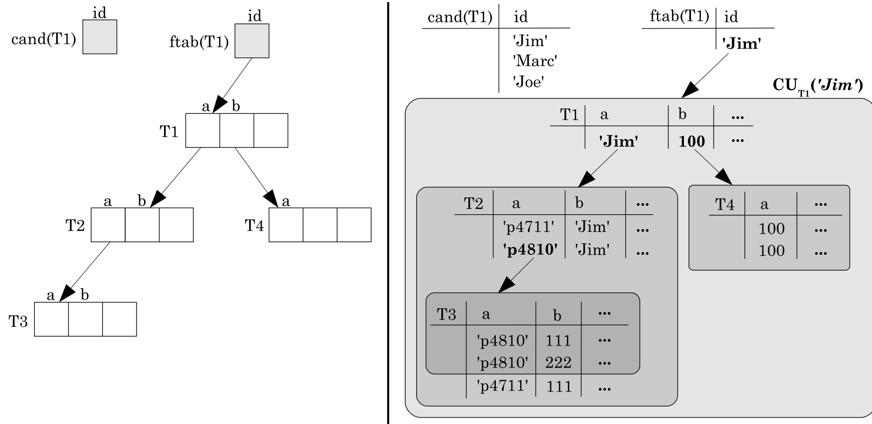

Fig. 1. Cache group: constraint specification (a), load effect of value *'Jim'* (b).

complete in $T3.a$, for which the example in Fig. 1b assumes that the records *('p4810',111,...; 'p4810', 222,...)* have to be inserted into $T3$. Note, the closure of *'Jim'* in $T1.a$ contains the records in $T2$ controlled by *'Jim'* and, in turn, all dependent closures recursively emanating from control values included, e. g., the closure of *'p4810'* in $T2.a$ contains records in $T3$, as illustrated in Fig. 1b.

Of special importance is the loading/unloading effect of a fill value, because it initiates cache loading or is subject to cache removal. The respective set of records is, therefore, called *caching unit* (CU). A fill value (e. g. *'Jim'* for the filling column $T1.a$) is managed by the *id* column of its fill table $ftab(T1)$ which is the source column of a special RCC $ftab(T1).id \rightarrow T1.a$. Hence, inserting/removing *'Jim'* to/from $ftab(T1).id$ implies loading/unloading of an entire caching unit CU_{T1}(*'Jim'*).

The set of records addressed by the CU_{T1}(*'Jim'*) is not necessarily the actual set to be considered by the cache manager for load/unload actions. Constraints of different CUs in a cache group may interfere and may refer to the same records such that record sets belong to more than one CU. Assume loading of CU_{T1}(*'Joe'*) causes the insertion of record *('p4810','Joe',...)* into $T2$. Because value *'p4810'* in $T2.a$ is already present and, in turn, $T3.a$ is already value complete for *'p4810'*, no further loading of $T3$ is necessary. On the other hand, the closure of *'p4810'* must not be removed in case CU_{T1}(*'Jim'*) is unloaded by the cache manager. Hence, when loading/unloading a caching unit, only records exclusively addressed by this CU – also denoted by *CU difference* – are subject to cache maintenance. This requirement ensuring correct query evaluation in the cache adds quite some complexity to cache management.

3.1 Key Problems

The structure of a cache group can be considered as a directed graph with cache tables as nodes and RCCs as edges (see Fig. 1a). Handling of cycles in such graphs is the main problem and, for that reason, considered separately in Sect. 3.1. To

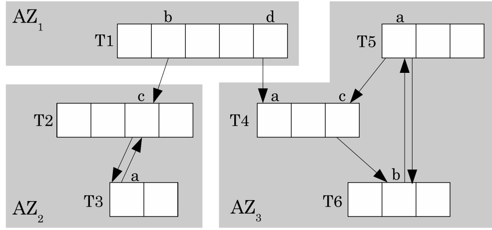

Fig. 2. Separation of a cache group in atomic zones.

gain a directed acyclic graph (DAG), we isolate cycles in so-called *atomic zones* (AZ) to manage them separately. Hence, in the simplest case, every cache table is a single atomic zone (*trivial atomic zone*). Otherwise, tables belonging to a cycle are assigned to the same atomic zone (*non-trivial atomic zone*). Fig. 2 shows a cache group example with segmentation into atomic zones.

Separation into atomic zones allows us to consider cache group maintenance in the resulting DAG from a higher level of abstraction [5]. Each atomic zone has to be loaded in a single atomic step, i. e., under transaction protection, to guarantee consistent results of concurrent queries. Reconsider Fig. 2: When a caching unit CU_{new} is loaded, top-down filling, i. e., AZ_1 before AZ_2 and AZ_3, would imply all affected atomic zones had to be locked till loading of CU_{new} is finished, because use of AZ_1, when related AZ_2 and AZ_3 are unavailable, would risk inconsistent query results. In contrast, a bottom-up approach allows to consistently access AZ_2 or AZ_3 for records in CU_{new} (e. g., when evaluation of a query predicate is confined by the atomic zone), although loading of the corresponding AZ_1 is not finished.

The reversed sequence can be used during unloading. After having removed its fill value from $ftab(T1)$, AZ_1 can be "cleaned" before AZ_2 and AZ_3 (within three transactions).

Cycles. By encapsulating cycles in atomic zones, we are now ready to consider their specific problems. An RCC cycle is said to be *homogeneous*[2], if it involves only a single column per table, for example, $T2.c \rightarrow T3.a$, $T3.a \rightarrow T2.c$ in Fig. 2. Loading of a homogeneous cycle is safe, because it stops after the first pass through the cycle is finished [7].

Unloading, however, may be complicated in homogeneous cycles due to an interdependency of records, as shown in the following example.

Example 3.1. [Dependencies in homogeneous cycles]
Fig. 3 represents a homogeneous cycle where *'Jim'* should be deleted from AZ_3. If we now try to find out whether or not *'Jim'* can be removed from the cycle, we have to resolve the cyclic dependency in $T1.a \rightarrow T2.a \rightarrow T3.a \rightarrow T4.a \rightarrow T1.a$. A standard solution is to mark records to identify those already visited. However, records cannot only be involved in an *internal dependency* within the cycle, but

[2] Heterogeneous cycles may provoke recursive loading and are, therefore, not recommended in [7].

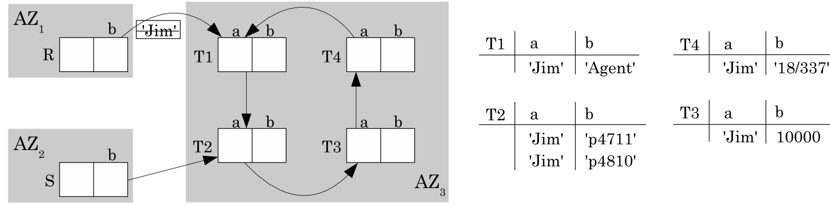

Fig. 3. Internal vs. external dependencies within an homogeneous cycle.

also in an *external dependency*. Such a dependency would exist if value *'Jim'* would be present in $S.b$. Then, due to the RCC $S.b \rightarrow T2.a$ in Fig. 3, value *'Jim'* would be kept in $T2.a$ and no records would be deletable in this example. On the other hand, a table in a cycle may have no matching records. For example, if records such as *('Jim', '18/337')* would not exist in $T4_B$, the cycle is broken for this specific value. Assume a broken cycle for *'Jim'* in $T4.a$ and simultaneously the existence of *'Jim'* in $S.b$, then only *('Jim', 'Agent')* could be deleted from $T1$.

Due to the illustrated problems, contribution [7] recommends deletion of the complete cache content, which implies that caching units with high locality would be reloaded immediately. Therefore, *selective unloading*, executed as an asynchronous thread, can save refill work and provide more flexible options to maintain the cache content. Sect. 5 presents the concepts to provide proper unloading of caching units and describes in addition some implementation details used in our prototype *ACCache*.

3.2 Measurements

The main objective of our empirical experiments is to gain some estimates for the maintenance of some basic cache group structures. In all measurements, we have stepwise increased the amount of records to be loaded or deleted, where the given number corresponds to the size of the CU difference caused by the related fill value. In all cases, we have distributed the affected records as uniformly as possible across the cache tables involved.

Important cache group types. We have previously argued that the lengths of RCC chains and homogeneous cycles are interesting for practical cache management. Using this directive, we have measured the maintenance costs of some basic cache groups, as illustrated in Fig. 4.

Data generator. To provide suitable data distributions and cache group contents, a data generator – tailor-made for our experiments – analyses the specific cache group and generates records corresponding to a CU difference of a given size and assigns them uniformly to the backend DB tables involved. All tables have seven columns; if a column is used in an RCC definition, data type `INTEGER` is used, whereas all other columns have data type `VARCHAR(300)`.

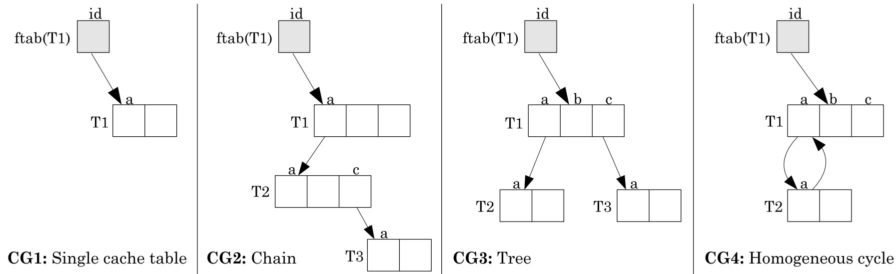

Fig. 4. Important cache group types

Measurement environment. For all measurements, we use the ACCache system [5] based on an existing DBMS in front- and backend (i. e. DB2), which was extended by the functionality described in Sect. 4 and 5. Applications participating in a test run are specified by means of three *worker nodes*: the simulated client application triggering loading of CUs by sending SQL queries, the ACCache system, and the backend DBMS. We implemented a tailor-made measurement environment which enabled controlled test runs: each test run was repeated 6 times with newly generated CU data where the sizes of the CU differences remained stable, but with data of varying content. Hence, all results presented are average response times of dedicated measurements.

Because we wanted to explore the performance and quality of load methods separated from network latency, we run the applications in a *local-area* network where data transmission costs are marginal. In the Internet, these costs would be dominated by possibly substantial network delays.

4 Loading of Caching Units

To preserve cache consistency, entire caching units, i. e., all records implied by the insertion of a fill value, must be loaded at a time. Of course, duplicates are removed if records – also belonging to other CUs – are already present. Because SQL insertions are always directed to single tables, records to be loaded are separately requested for each participating table (which coincides with an atomic zone in non-cyclic cases) from the backend.

4.1 Direct Loading

The first method *directly* inserts the records into the cache tables where the atomic zones are loaded bottom-up. The quite complex details are described in [5]. In principle, the cache manager requests the data by table-specific predicates, which reflect the RCC dependencies of the table in the cache group, from the backend DBMS. For each table involved, the record set delivered is inserted observing the bottom-up order, thereby dropping duplicate records. While cache group $CG1$ in Fig. 4 can be loaded by a simple backend request, i. e.,

Q1: `select * from` $T1_B$ `where` $T1_B.a =$ 'v',

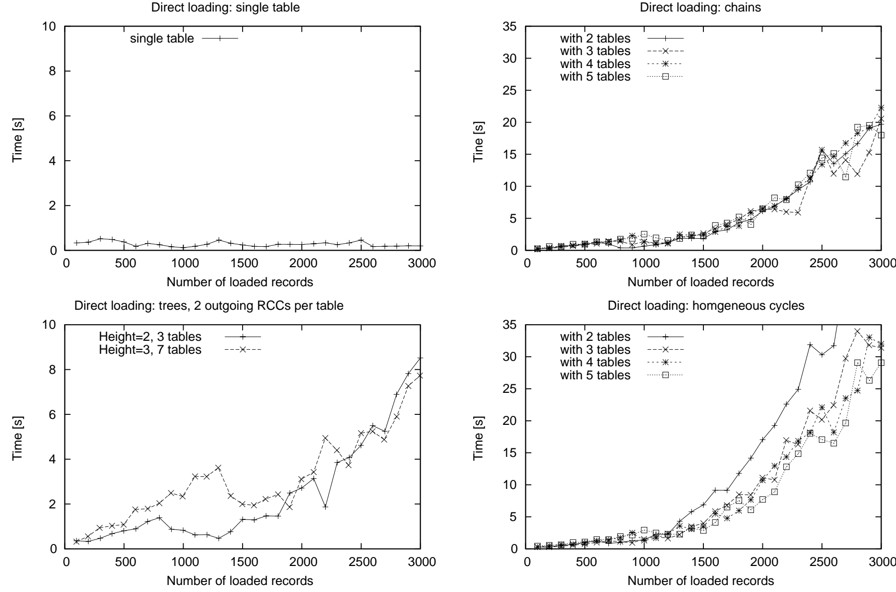

Fig. 5. Results for direct loading: single node, chains, trees, homogeneous cycles

$CG2$ and $CG3$ obviously need three load requests. Although $CG4$ consists of only a single atomic zone, up to three requests are necessary to load all tables participating in the cycle. Essentially, the table maintenance cost is caused by the predicate complexity required to specify the records to be inserted. While insertion into $T1$ of $CG1$ in Fig. 4 is very cheap, the records, e. g., to be inserted into $CG2$ have to be evaluated by three queries: filling $T1$ is similar to Q1 above, whereas the queries Q2 and Q3 for $T2$ and $T3$, respectively, are more complex:

Q2: `select` * from= $T2_B$ `where` $T2_B.a$ `in`
    (`select` $T1_B.a$ `from` $T1_B$ `where` $T1_B.a =$ 'v')

Q3: `select * from` $T3_B$ `where` $T3_B.a$ `in`
    (`select` $T2_B.c$ `from` $T2_B$ `where` $T2_B.a$ in
        (`select` $T1_B.a$ `from` $T1_B$ `where` $T1_B.a =$ 'v')).

In the example, the inherent complexity is needed to determine all join partners for the CU. When inserting records with value $T1_B.a =$ 'v' into $T1$ of $CG2$, Q2 delivers all join partners needed in $T2$ for $T1$ (to satisfy RCC $T1.a \rightarrow T2.a$) and, in turn, Q3 those in $T3$ for $T2$. Apparently, an RCC *chain* of length n requires $n - 1$ joins and one selection.

Measurement results. Our experiments reported in Fig. 5 correspond to the cache group types sketched in Sect. 3.2 and primarily address scalability of the load method. In each case, we continuously increased the number k of records to be loaded up to 3000. Because of the simple selection predicate in $CG1$ and the missing need for duplicate elimination, Fig. 5a scales very well and the cost

involved in selecting, comparing, and inserting of data was hardly recognizable in the entire range explored.

The remaining experiments were coined by the counter-effect of smaller result sets per table and more load queries with more complex predicates to be evaluated. When n tables were involved, the load method had to select $\sim k/n$ records per table. In Fig. 5b, e. g., the experiments for $k = 3000$ and chains of length 2, 3, 4, and 5 were supplied by 1500, 1000, 750, and 600 records per table, respectively.

While the load times quickly entered a range unacceptable for online transaction processing, the existence of cycles augmented this negative performance impact once more. In summary, if the amount to be loaded is higher than several hundred records, direct loading cannot be applied. Hence, a new method called *indirect loading* was designed to avoid these problems encountered.

4.2 Indirect Loading

Indirect loading reduces the high selection costs of direct loading using so-called *shadow tables*. Before making a requested CU available for query evaluation, it is entirely constructed at the cache side and then merged with the actual cache content. This proceeding allows arbitrary CU construction asynchronously to normal cache query processing. Therefore, it implies much simpler predicates of load queries, because the CU fractions of the participating atomic zones can be loaded top-down, for which simple selections on single backend tables are sufficient. For each cache table, a corresponding shadow table (indicated through a subscripted S) with identical column definitions is created which holds the collected record of a requested CU (see Fig. 6b).

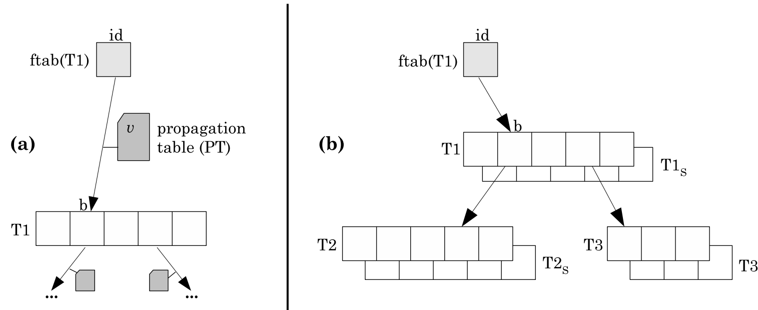

Fig. 6. New concepts: propagation tables (a), shadow tables (b)

Before these records are merged bottom-up, the preceding top-down working collection is implemented through a simple recursive algorithm based on so-called *propagation tables* (PT). These tables, defined for each RCC, consist of only a single column and control the propagation of distinct RCC source values (also denoted as control values) to be loaded to the shadow tables.[3] We

[3] In Sect. 5, PTs are also used to control propagation of unloading from cache tables.

denote the values propagated trough PTs as *propagation values*. To load a *CU* (see Fig. 6a), its fill value v is inserted into $ftab(T1).id$. The PT attached to RCC $ftab(T1).id \rightarrow T1.b$ obtains value v and trigger value completeness for it in $T1.b$. In turn, newly loaded control values in $T1$ are again propagated along PTs of outgoing RCCs. As long as propagation values are present in PTs, the respective records are collected according to the principles described in Sect. 3. The process stops if all control values are satisfied, i. e. if all propagation values are processed/consumed. Subsequently, the freshly loaded CU is merged in bottom-up fashion with the related cache tables thereby eliminating duplicate records and observing all RCCs.

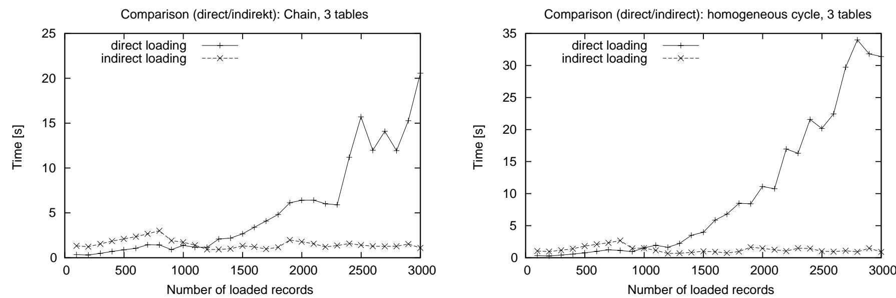

Fig. 7. Improvements achieved by indirect loading

Comparison: direct/indirect loading. Fig. 7 measures the performance of direct and indirect loading for two cache group types. We have empirically compared again those cache group types (chains and cycles) which achieved worst performance with the previous load method. Indirect loading was primarily designed to solve such performance problems and, indeed, the results are quite clear: In both cases, the costs involved for indirect loading were often lower than one second and did not exceed 3 seconds. Note, because CU preparation in shadow tables is aynchonous, only short locks are necessary for the merging phase. Therefore, the timings are acceptable, because concurrent queries are not severely hindered.

The performance reached for loading seems to be further improvable: So far, both methods are executed by the cache DBMS. This means that record selection needs multiple requests to the backend DBMS and, in turn, is burdened by multiple latencies between cache and backend. Therefore, the so-called *prepared* loading tries to avoid these disadvantages.

4.3 Prepared Loading

This method entirely delegates the collection of CU records to the backend DBMS. As a prerequisite, the backend DBMS needs to maintain additional metadata about the cache groups supplied. The cache manager requests the data for a new caching unit by sending the corresponding fill value to the backend. The way the data is collected is similar to indirect loading, but happens at the backend.

A prepared CU is then packaged and transferred to the cache. Cache merging only has to observe uniqueness of records.

Because the goal of caching is usually to off-load the server DBMS, this "optimization" partly yields the opposite and requires the server to maintain cache-sided metadata and to perform extra work. This method, however, may be very useful in case of cache groups having n atomic zones and high latency between cache and backend, because only a single data transfer to the cache is needed instead of n (and even more in the presence of cycles). Effects of latency, however, are not considered in this paper.

5 Unloading of Caching Units

After having explored various options for cache loading, we consider selective unloading of cached data. Note, keeping unused data in the cache increases maintenance costs to preserve consistency and freshness without bringing benefits in terms of reduced query response times. Therefore, it is important to control data references in the cache and to possibly react by removing fill values and their implied CUs whose reference locality degraded. Of course, replacement algorithms in cache groups are more complicated than those for normal database buffers.

To unload a fill value together with its CU, the atomic zones involved are traversed top-down (*forward directed unloading*), as sketched in Sect. 3.1. The control values to be deleted are propagated using the same PTs already introduced in Sect. 4.2. Note, because records in a CU may be shared by other CUs, actually only the CU difference must be removed, which implies checking whether or not the records considered for replacement exclusively belong to the CU to be unloaded. In the following, we outline our replacement provisions, before we describe unloading in a trivial atomic zone and the more complex procedure for non-trivial atomic zones.

5.1 Replacement Policy

We used as replacement policy the well known LRU-k algorithm, for which we record the timestamps of the k most recent references to a CU in the related control table (see [11]). The replacement decision for CUs refers to extra information recorded in each control table. The first is a high-water mark concerning the number of related caching units to be simultaneously present in the cache. The second characterizes the minimum fill level observed for CU unloading. The current fill level is approximated by the number of rows in the control table (the number of CUs actually loaded) divided by the number of candidate values. When the fill level reaches the high-water mark, a delete daemon has to remove records to make room in the cache. Such a strategy allows to separately control the cache space dedicated for each filling column, which enables fine-tuning of locality support and is much more adaptive compared to a single occupancy factor assigned to the whole cache group.

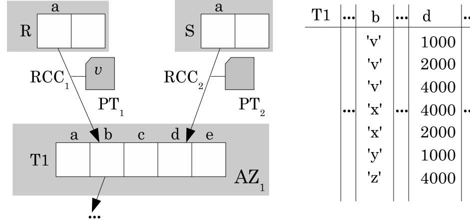

Fig. 8. Unloading in trivial atomic zones

5.2 Unloading in Trivial Atomic Zones

Consider the cache group fragment shown in Fig. 8. The given processing sequence for atomic zones (see Sect. 3.1) ensures that the related PTs of incoming RCCs obtain all propagation values if control values were removed from preceding atomic zones. In the example, deletion in $T1$ is initiated by value v propagated through PT_1. Hence, value v defines the *starting point* for the deletion process in table $T1$.

To determine the deletable set of records, each record with $T1.b$ = 'v' is to be checked whether or not it is eligible, i. e., whether other control values do not enforce its presence in the cache. In our example, all records can be deleted which are not restrained by control values of RCC_2. Therefore, if only the control value 1000 is present in $S.a$ (the source column of RCC_2), all records $\sigma_{(b=\text{'}v\text{'} \wedge d \neq 1000)} T1$ can be deleted. Thus, deletion within a trivial atomic zone can be performed with a single delete statement. The following statement Q4 removes all deletable records from AZ_1, where all incoming PTs (in our case PT_1 and PT_2) are observed.

```
Q4: delete from T1
     where (b in (select CV from PT1) or
            d in (select CV from PT2))
       and (b not in (select R.a from R) and
            d not in (select S.a from S))
```

As indicated by our cache group examples in Fig. 4, most tables are encapsulated in trivial atomic zones. Because unloading of them can be achieved by considering each atomic zone in isolation (thereby observing the top-down processing sequence), this maintenance task remains simple and can be effectively performed. In rare cases, however, removal of records becomes more complicated if they are involved in cyclic RCC references. Such a case will be discussed in the following.

5.3 Unloading in Non-trivial Atomic Zones

We now consider in detail the problems sketched in example 3.1, where we have differentiated internal from external dependencies. To explain their effects and to resolve them, we analyze unloading in the homogeneous cycle shown in Fig. 9.

The algorithm proceeds in two phases: *global deletion* and *internal deletion*. We denote the values in columns which form a homogeneous cycle as *cycle values*.

To resolve their dependencies as fast as possible, the key idea is to initially find all cycle values whose records are not involved in external dependencies. In Fig. 9, these are all records having value 1000, because values 2000 and 3000 are assumed to be externally referenced through $R.a$ = *'x'* and $S.a$ = *'j'*. After deletion of all records with cycle value 1000 (in phase global deletion), the atomic zone just holds records which have no more dependencies (neither internal nor external) or records which could not be deleted due to an existing external dependency. Hence, the internal cyclic dependencies are eliminated if this was possible. The remaining records are deleted within the second phase called internal deletion, which is also performed in forward direction (similar to trivial atomic zones), but only analyzes the tables within the non-trivial atomic zone. As a consequence, the records having value 3000 in $T1$ and $T2$ are deleted in our example; the corresponding record in $T3$, however, cannot be deleted due to the external dependency present. Subsequently, we consider the two phases in detail.

Global deletion. Refer to Fig. 9 and assume that the value v needs to be deleted as indicated. To find the cycle values whose records are globally deletable, a join between all tables having incoming external RCCs is performed. In our example, these are the tables $T1$ and $T3$. Hence, query Q5 returns the deletable cycle values:

Q5: `select` $T1.b$ `from` $T1$`,` $T3$
  `where` $T1.b = T3.a$
   `and` $T1.a$ `in (select` CV `from` PT_1`)`
   `and` $T1.a$ `not in (select` $R.a$ `from` R`)`
   `and` $T3.b$ `not in (select` $S.a$ `from` S`)`

The example shows that it is sufficient to perform the dependency check via the control values of incoming external RCCs. Because cyclic internal dependencies cannot be violated, such cyclic dependencies are not observed in this phase. Hence, this approach exploits the fact that the cycle values can be joined within an homogeneous cycle. When control values are affected during the deletion of records which hold the corresponding cycle values (in our example the records

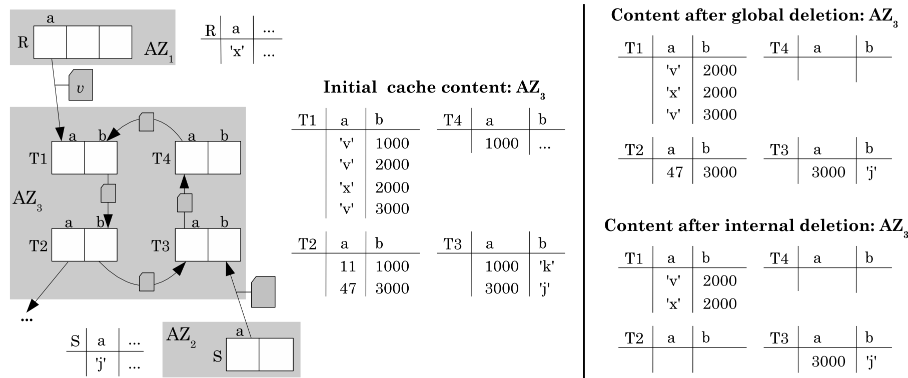

Fig. 9. Unloading in non-trivial atomic zones.

having value 1000), they have to be propagated only along external, outgoing RCCs (e. g. RCC_3 in Fig. 9 using their PTs to continue deletion in subsequent atomic zones). In Fig. 9, this is necessary for value 11 which is completely removed from $T2.a$.

Internal deletion. Internal deletion is performed in a similar way as unloading in trivial atomic zones. Starting at a table with a PT value attached, all incoming RCCs (external and internal RCCs) are checked to find deletable records. Such records are removed and all affected control values are propagated along **all** outgoing RCCs (using their PTs). The internal deletion ends if there is no PT anymore holding propagation values for the related AZ. In our example, the process stops in table $T3$, because the value 3000 still has external dependencies.

5.4 Measurement Results

Fig. 10 illustrates the times needed to unload specific caching units. In all cache group types, the unloading process, using SQL statements already prepared, was very efficient (typically much faster than 200 ms). Only the initial statement preparation included in the first measurements (selecting 100 records) caused comparatively high costs. These costs are also included in preceding measurement results (see Fig. 5) where, however, this minor cost factor is insignificant for the times measured. The execution time consumed to unload a CU within an homogeneous cycle is similar to that needed to unload chains, which illustrates that we are now also able to unload homogeneous cycles with acceptable performance.

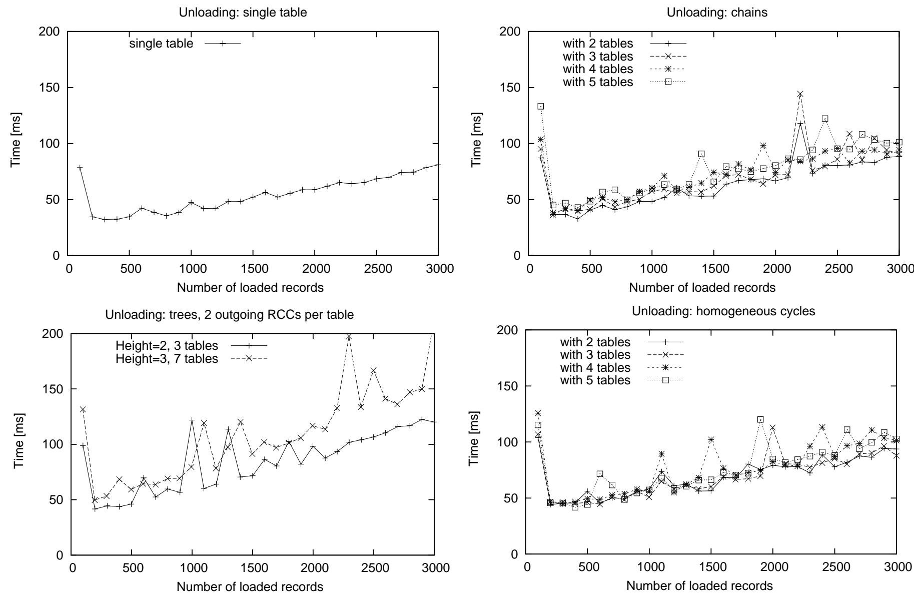

Fig. 10. Unloading of cache units

6 Conclusion

CbDBC supports declarative query processing close to applications. The cache constraints to be applied pose particular challenges for cache management and maintenance. With the help of the methods and algorithms presented, it is now possible to selectively load and unload caching units efficiently (also in homogeneous cycles). Starting from the performance problems caused by direct loading, we introduced a new method called indirect loading, which improves cache maintenance dramatically. When latency is too high, preparation of caching units within the backend DBMS could relieve the delays implied by the loading process. Finally, we presented a novel unloading mechanism, which is also able to handle unloading of homogeneous cycles. Supported by a variety of empirical measurements, we confirmed that acceptable maintenance efficiency can be reached for all important cache group types.

References

1. Altinel, M., Bornhövd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., Reinwald, B.: Cache tables: Paving the way for an adaptive database cache. In: VLDB Conf. (2003) 718–729
2. Amiri, K., Park, S., Tewari, R., Padmanabhan, S.: DBProxy: A dynamic data cache for web applications. In: ICDE Conf. (2003) 821–831
3. Anton, J., Jacobs, L., Liu, X., Parker, J., Zeng, Z., Zhong, T.: Web caching for database applications with Oracle Web Cache. In: SIGMOD Conf. (2002) 594–599
4. Bello, R.G., Dias, K., Downing, A., Feenan, Jr., J.J., Finnerty, J.L., Norcott, W.D., Sun, H., Witkowski, A., Ziauddin, M.: Materialized views in Oracle. In: VLDB Conf. (1998) 659–664
5. Bühmann, A., Härder, T., Merker, C.: A middleware-based approach to database caching. In Manolopoulos, Y., Pokorný, J., Sellis, T., eds.: ADBIS Conf.,. LNCS 4152, Springer (2006) 182–199
6. Goldstein, J., Larson, P.: Using materialized views: A practical, scalable solution. In: SIGMOD Conf. (2001) 331–342
7. Härder, T., Bühmann, A.: Value complete, column complete, predicate complete – Magic words driving the design of cache groups. The VLDB Journal (2008) 805–826
8. Larson, P., Goldstein, J., Guo, H., Zhou, J.: MTCache: Mid-tier database caching for SQL server. Data Engineering Bulletin **27**(2) (June 2004) 35–40
9. Larson, P., Goldstein, J., Zhou, J.: MTCache: Transparent mid-tier database caching in SQL server. In: ICDE, IEEE Computer Society (2004) 177–189
10. Levy, A.Y., Mendelzon, A.O., Sagiv, Y., Srivastava, D.: Answering queries using views. In: PODS Conf. (1995) 95–104
11. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The lru-k page replacement algorithm for database disk buffering. In: SIGMOD Conf. (1993) 297–306
12. Oracle Corporation: Internet application server documentation library (2008)
13. The TimesTen Team: Mid-tier caching: The TimesTen approach. In: SIGMOD Conf. (2002) 588–593