A Middleware-Based Approach to Database Caching

Andreas Bühmann, Theo Härder, and Christian Merker

Department of Computer Science, University of Kaiserslautern,
P. O. Box 3049, D-67653 Kaiserslautern, Germany
{buehmann, haerder, merker}@informatik.uni-kl.de

Abstract. Database caching supports declarative query processing close to the application. Using a full-fledged DBMS as cache manager, it enables the evaluation of specific project-select-join queries in the cache. In this paper, we propose significant improvements and optimizations – as compared to the well-known DBCache approach – that make our caching concept truly adaptive. Furthermore, we describe an adaptive constraint-based cache system (ACCache) relying on middleware components as a DBMS-independent realization of this approach.

1 Motivation

While Web caching is concerned with reducing response time and bandwidth consumption for service requests in the user-to-server path, *database (DB) caching* focuses on request optimization in the remaining path from the Web server to the *backend* database, which keeps the dynamic up-to-date data used by transactional programs to derive user query results. In contrast to Web caching, which can only answer identifier-based cache requests, DB caching provides declarative query processing, which makes it much more powerful but also complex.

To accelerate service requests of Web users and, at the same time, to improve scalability of applications accessing the backend DB, application servers frequently migrate to data centers closer to the user "at the edge of the Internet". Special algorithms enable Web clients to select one of the replicated servers close to them thereby minimizing response times of Web services. However, this is only true if locality of data reference can be provided by such application servers – often achieved through geographical contexts of these services. Otherwise, frequent round-trips to the remote backend DB may degrade the performance of DB-based services to a level much worse than without application server migration. Therefore, it is vital for the entire migration approach to keep prevalently used data close to the application in database *caches* (also called *frontend* DB servers).

In Sect. 2, we present an adaptive constraint-based caching concept supporting the evaluation of project-select-join (PSJ) queries. This mechanism must be entirely transparent to application programs such that turning caching on or off only affects query performance. Because (any type of) caching always has inherent trade-offs as far as cache consistency and maintenance is concerned, only DB contents exhibiting high locality of reference should be kept in the cache. Therefore, only a few tables containing selected records are maintained in a typical cache, arranged into *cache groups*, although the backend DB may consist of hundreds of tables. Moreover, caching is always kind of

speculative, because it should anticipate changing workload needs in the future. Thus, caching adaptivity is of utmost importance. As compared to [1], we propose a much more flexible mechanism enabling orthogonality of parameter specification (by candidate values) and cache filling as well as evaluation of more query types.

Section 3 describes an implementation of this mechanism based on middleware concepts. While cache management is rather straightforward for simple cache groups (e. g., *Director* → *Movie*), query processing power is limited in such cases. Thus, to reveal the strengths and weaknesses of ACCache, we have chosen a rather complex running example (Fig. 1). Section 4 summarizes our results and identifies future work.

2 Constraint-Based Database Caching

Constraint-based database caching promises a new quality for the placement of data close to their application. The key idea is to accomplish *predicate completeness* in the cache for some given types of query predicates P such that all queries matching P can be evaluated correctly.

A database cache is a database consisting of cache tables. Cache tables represent selected backend tables in the cache and contain subsets of their records[1]. All records (of various types) in the backend DB that are needed to evaluate a predicate P are called the *predicate extension* of P. If a collection of cache tables contains the predicate extension of a predicate P, it is said to be *predicate complete* with respect to P. Note that a predicate extension in the sense used here consists of all records from the backend tables needed to reconstruct the query result. For an aggregate query, the predicate extension would not be the aggregate (as the query result) but all records to be aggregated.

Cache constraints enable cache loading in a constructive way and guarantee the presence of their respective predicate extensions in the cache. This technique does not rely on static predicates: Parameterized constraints make the specification adaptive; it is completed when specific values instantiate the parameters: An "instantiated constraint" then corresponds to a predicate and, once the constraint is satisfied (i. e., all related records have been loaded), it delivers correct answers to eligible queries. Note, the set of all present predicate extensions flexibly allows combined evaluation of their predicates in the cache.

Given suitable cache constraints, there are no or only simple difficulties in deciding whether certain predicates can be evaluated. At run time, only simple existence queries are required to determine whether suitable predicate extensions are available.

The primary task of this constraint-based caching approach is to support local processing of queries that typically contain simple projection and selection operations as well as equi-joins (PSJ). Because all columns of the corresponding backend tables are kept, all *project* operations possible in the backend DB can also be performed in the cache. Other operations like *selection* and *join* depend on specific cache constraints. Furthermore, since full DB functionality is available, the results of these PSJ queries can be subjected to further arbitrary selections and transformations.

[1] In the present state of our model, we deal with whole records only and do not consider projections of certain sets of columns, as DBProxy [2] does, for example.

2.1 Completeness

For predicates we would like to evaluate in the cache, we must guarantee predicate completeness. Considering a cache table S, we denote by S_B its corresponding backend table, by $S.c$ a column c of S.

Let us begin with single cache tables. For simple equality predicates like $S.c = v$, where v is a value, the predicate completeness takes the shape of *value completeness*.

Definition 1 (Value completeness). *A value*[2] *v is said to be 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.*

Obviously, *if* we know that a value v is value complete in a column $S.c$, we can correctly evaluate $S.c = v$ in the cache, because all records from table S_B that carry this value are there. Determining which values actually are complete is the task of *probing*, which will be introduced in Sect. 2.2.

To obtain the predicate extensions of PSJ queries we use *referential cache constraints* (RCCs) between cache columns to specify all records needed to satisfy specific equi-join predicates.

Definition 2 (Referential cache constraint). *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 $T.b$.*

An RCC $S.a \rightarrow T.b$ guarantees, whenever we find a record s in cache table S, that all join partners of s with respect to $S.a = T.b$ are in T, too. Note, the RCC alone does not allow us to correctly perform this join in the cache: Many records of S_B that have join partners in T_B may be missing from S. But using an equality predicate with a complete value in column $S.c$ as an *anchor*, we can restrict this join to pairs of records that are present in the cache: The RCC $S.a \rightarrow T.b$ expands the predicate extension of $S.c = x$ to the predicate extension of $S.c = x \wedge S.a = T.b$. In this way, a column with a complete value can serve as an *entry point* for a query into the cache; it allows us to start reasoning about predicates evaluable in the cache: Once the cache has been entered in this sense, reachable RCCs show us where joins can correctly be performed. Of course, the application of RCCs can be chained: A second RCC $T.d \rightarrow U.e$ could expand the predicate extension to $S.c = x \wedge S.a = T.b \wedge T.d = U.e$.

Figure 1 shows a cache setup for a movie database, including many RCCs used to connect the selected cache tables. Let us assume we know that the name 'Bond' is complete in $A.name$[3], which means that *all* actors named 'Bond' are in the cache. We can then safely evaluate the predicate $A.name =$ 'Bond' in the cache, because it is predicate complete with respect to this predicate. Furthermore, since we guarantee that all specified RCCs hold at any time, we are allowed to evaluate

$$A.name = \text{'Bond'} \wedge A.id = P.aid \wedge P.mid = M.id \wedge M.zip = C.zip$$

in the cache, too. Of course, this is only a skeleton of a possible query and could be enriched with further selection predicates such as $M.title =$ 'Dr. No'.

[2] As SQL's *null* indicates the absence of a value, we do not regard *null* in itself as a value.

[3] In formulas like this one, we like to abbreviate the table names.

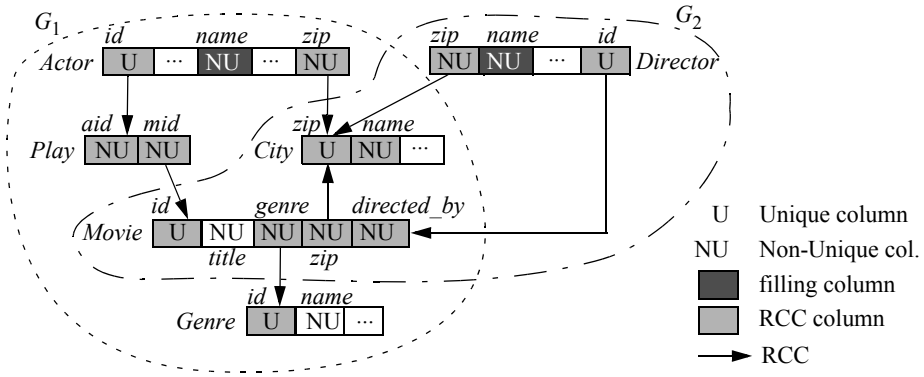

Fig. 1. Cache groups G_1 and G_2

Using RCCs we implicitly introduce a value-based table model intended to support queries. Despite similarities to the relational model, RCCs are not identical to the PK/FK (primary key / foreign key) relationships contained in the backend schema. A PK/FK relationship can be processed symmetrically, whereas our RCCs can be used for join processing only in the specified direction. There are other important differences: $n:m$ RCCs (NU $\rightarrow$ NU) have no counterparts in the backend DB, and a column may be the source of n and the target of m RCCs. In contrast, a column in the role of a primary key may be the starting point of k, but in the role of a foreign key the ending point of only one (meaningful) PK/FK relationship.

2.2 Probing for Entry Points

RCCs allow us to draw conclusions about predicate extensions that are in the cache, but only if we can rely on some value being complete and serving as an entry point. Considering some column $S.c$, how do we know that a value v is complete there? Obviously, our goal ought to be to provide simple and efficient means for deciding about the completeness of values in the cache: The process of using simple (existence) queries on the cache to decide about completeness is called *probing*; the queries used are called *probe queries* accordingly.

In contrast to DBCache [1], we use a new probing approach [3], which does not require new constraints and thus does not load extra records into the cache (as DBCache's cache keys do). The fundamental insight is that RCCs already provide guarantees about complete values in the cache: The source column $S.a$ of an RCC $S.a \rightarrow T.b$ (or more precisely, the values therein) controls which values are complete in its target column $T.b$. We therefore call $S.a$ a *control column* of $T.b$.

In general, any given column $S.c$ can have zero or more control columns. Whenever a column $S.c$ we would like to use as an entry point for a predicate $S.c = v$ has at least one control column, we can probe (i. e., check for the existence of value v) in the control columns of $S.c$. If we find v in one of these columns, we know that it is value complete in $S.c$ and that we can correctly evaluate the predicate in the cache.

In our example in Fig. 1 the following five columns possess control columns and could thus serve as entry points: *P.aid*, *M.id*, *M.directed_by*, and *G.id* have one control column each, *C.zip* even three (*A.zip*, *D.zip*, and *M.zip*), which would require – in the worst case – to probe in all the three columns for a value.

Probing can be optimized if we can deduce that, at all times, a column can contain complete values only.

Definition 3 (Column completeness). *A cache column S.c is said to be column complete (or complete for short) if and only if all values v in S.c are value complete.*

Given a complete column $S.c$, if a probe query confirms that value v is present in $S.c$ (a single record suffices), we can be sure that v is value complete and thus evaluate $S.c = v$ in the cache. Unique columns of a cache table are complete per definition. In contrast, non-unique columns are only complete under special conditions (or if completeness is enforced through additional cache constraints[4]).

You can show that a column $T.c$ is complete (at all times) if

- it is a U column,
- it is a column with an (self-)RCC $T.c \rightarrow T.c$, or
- it is the only column in table T with incoming RCCs.

In our example, we have five U columns and one additional complete NU column, namely *P.aid*. Column *M.directed_by* is not complete, because table *M* is reached by another incoming RCC on column *M.id*.

Probing Strategies. When looking for an entry point for a predicate $S.c = v$, we have two kinds of probing operations at our disposal:

- If $S.c$ is column complete, we can probe directly in $S.c$.
- If $S.c$ has at least one incoming RCC, we can probe in a control column of $S.c$.

We may choose between these two, based on the probing costs (e. g., is there an index on the probed column?). We may even apply a number of successive probing operations for a single entry point, thereby forming probing strategies. In this case, the order of the probing operations and their probabilities of success determine the average costs of the whole probing strategy.

2.3 Loading Predicate Extensions

To be able to evaluate a predicate Q in the cache, the cache manager must guarantee predicate completeness for Q by loading all required records into the cache tables. Following the RCCs, the cache manager can construct predicate extensions using only simple loading steps based on equality of values.

Obviously, there must be some way to tell the cache manager which predicate extensions to load. In essence, this means placing single values into specific cache columns, from where the cache manager will fill the cache, guided by the cache constraints.

[4] For example, DBCache's cache key columns are forcibly complete.

Candidate Values in Filling Columns. Besides RCCs, a second type of cache object is needed in order to establish a parameterized loading mechanism: Attached to selected *filling columns* are sets of *candidate values* (CVs), which alone initiate the loading of predicate extensions when they are referenced by user queries.

The set of all candidate values of a filling column $S.f$ is denoted by $C_{S.f}$ and is always a subset of $S_\mathrm{B}.f$'s domain. Whenever a candidate value v in $C_{S.f}$ occurs in an equality predicate of a query ($S.f = v$), the cache manager probes the respective cache table as usual to see whether this value is present: A successful probe query (the value v is found) implies that the predicate extension for the given equality query is in the cache and that this query can be evaluated locally. Otherwise, the query is sent to the backend to continue processing.

As a further consequence of this cache miss attributed to v, the cache manager satisfies the value completeness for v asynchronously by fetching all required records from the backend and loading them into the respective cache table. It then proceeds to restore the validity of all RCCs by loading the necessary records into the remaining tables. Hence, the cache is ready to answer the corresponding equality query locally from then on as well as all queries anchored by it.

Apparently, a reference to a candidate value v serves as a kind of indicator that, in the immediate future, locality of reference is expected on the predicate extension determined by v. Candidate values therefore carry information about the future workload and sensitively influence caching performance. Hence, candidate values must be selected carefully. In an advanced scheme, the cache manager itself takes care, by monitoring the query load, that only those values with high re-reference probability become and stay candidate values. In a straightforward case, the database administrator (DBA) specifies the set of candidate values (e. g., as the domain itself, an enumeration, a range, or as other predicates) positively or negatively (stop-words).

Master Control Columns. The subset of candidate values of a filling column $S.f$ that have already been referenced and therefore actually are in the cache controls which values are complete in $S.f$ and, hence, behaves similar to the contents of a control column. To allow uniform treatment of all cache columns with regard to probing and filling, we introduce an *artificial* control column $\mathrm{ctrl}(f)$ for each filling column f.

This *master control column* $\mathrm{ctrl}(f)$ is a U column of a separate, anonymous (master control) table with an RCC $\mathrm{ctrl}(f) \rightarrow f$ pointing to the filling column f.

Having made this step, we can simply regard the domain of $\mathrm{ctrl}(f)$ as the set of candidate values of f, whereas the actual contents of column $\mathrm{ctrl}(f)$ (i. e., some of the candidate values) determines which predicate extensions are in the cache. When looking for an entry point for a predicate $f = v$, we can use our regular probing strategies (and probe in the control column $\mathrm{ctrl}(f)$, for instance); in case of a cache miss, the value v is inserted into the master control column $\mathrm{ctrl}(f)$ from where the cache manager will start its loading steps to reestablish the validity of all cache constraints.

Now the only special thing about filling columns is their sensitivity to references of values in equality predicates, which leads to new values in their artificial control columns. With respect to probing, query evaluation, and even filling via RCCs they behave exactly like any other column.

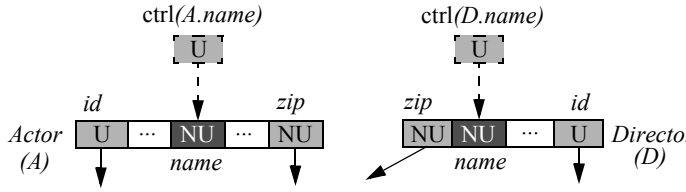

Fig. 2. Master control columns of filling columns

Figure 2 shows the master control columns ctrl($A.name$) and ctrl($D.name$) for the two filling columns $A.name$ and $D.name$ (dark gray) of our example. Assume $A.name =$ 'Bond' is part of a predicate. Hence, the filling column $A.name$ is our potential entry point. We can now probe for 'Bond' in the control column of $A.name$, which happens to be the master control column ctrl($A.name$). If we find the value there, we can evaluate the predicate in the cache. If we do not, we must pass the predicate on to the backend, but can prepare for subsequent cache-based evaluations of the predicate by inserting 'Bond' into ctrl($A.name$).

With the master control columns in place, we can add the NU columns $A.name$ and $D.name$ to our set of potential entry points gathered in Sect. 2.2, which yields a total count of nine.

2.4 Cache Groups and Federations

In general, our caching mechanism supports PSJ queries that are characterized by predicate types of the form $(EP_1 \vee \ldots \vee EP_n) \wedge EJ_1 \wedge \ldots \wedge EJ_m$, where the EP_i, $1 \leq i \leq n$, are equality predicates on filling columns of a specific cache table called *root table* and the EJ_j, $1 \leq j \leq m$, correspond to RCCs that (transitively) connect the root table with the remaining cache tables involved. The resulting structure is called *cache group*, which is our unit of design to support a specific predicate type in the cache.

Definition 4 (Cache group). *A cache group is a collection of cache tables linked by a set of RCCs. A distinguished cache table is called the root table R of the cache group and holds one or more filling columns. The remaining cache tables are called member tables and must be reachable from R via RCCs.*

Whenever more than one basic predicate type should be supported in a cache, we have to consider the *federation* of cache groups overlapping in some tables. On the one hand, memory space may be saved in shared cache tables, but, on the other hand, implicit extension of one cache group by RCCs of another one may lead to the loading of many unwanted records into the cache.

In our example, cache group G_1 and G_2 are designed for the two predicate types

$$(A.name = v_1) \wedge A.id = P.aid \wedge P.mid = M.id \wedge M.genre = G.id \wedge A.zip = C.zip$$
$$(D.name = v_2) \wedge D.zip = C.zip \wedge D.id = M.directed\_by \wedge M.zip = C.zip$$

and share the member tables C and M in the federation (see Fig. 1).

2.5 Related approaches

At first sight, DBCache [1,4] uses similar concepts to perform database caching with cache groups: The concept RCC and the basic method of determining predicates evaluable in the cache are the same. But DBCache does not use the concept of predicate extensions or predicate completeness to explain why the cache is structured as it is. It has no notion of master control columns or of probing in control columns in general and is restricted to complete columns (DBCache term: domain-complete columns) as potential entry points. To make at least filling columns complete, further constraints called cache keys are employed – they fail to separate values referenced and wanted to be complete in the cache (contents of our master control columns) from values that are in the cache because of other constraints and may thus lead to unwanted cache loading.

Our master control columns have been inspired by the *control tables* in the MTCache project [5,6], which are used in quite a similar way: There a set of stacked materialized views is used to describe the cache contents, each dependent on the contents of another view (which resembles RCCs) or ultimately on the contents of a control table.

3 Architecture of ACCache

The key idea of DB caching is to provide – close to the application server – a query processing facility, which must be transparent for the transaction programs requesting DBMS services. For developing an adequate architecture, it is reasonable to strive for a solution which is independent of a specific DBMS and exclusively rests on the availability of some SQL engine. Hence, it became obvious that we should go for a flexible solution based on middleware concepts. In this way, our work does not rely on the goodwill of a single manufacturer (which would require to massively modify and expand the code of an existing DBMS) and gains flexibility and openness thereby enabling the use of different DBMS engines at minimal porting costs. Furthermore, we have the opportunity to avoid the trade-offs and to combine – based on our concepts described in Sect. 2 – the advantages of different existing systems [1,5].

3.1 Component Architecture

Figure 3 illustrates the main tasks of our adaptive constraint-based caching system by its components providing the required services and their interaction. Cache transparency for the user is achieved through the JDBC interface, which accepts SQL statements and delivers results in the way the application program expects. All requests are passed on to some Query Worker, which analyzes them, regarding the cache's configuration and its current contents, and – if processing in the cache is possible – transforms them such that references to cache and backend tables can be separated. Hence, the native federated query facility of the DBMS used [7] can distribute (appropriate parts of) the query statement to the cache DB and backend DB. Sending DB requests and receiving their results are handled by the middle tier thereby providing a uniform interface (to all ACCache components) and controlling all accesses to the underlying DBMSs.

The Setup and Initializer components perform the initial cache creation using a configuration file and possibly some initial cache filling. Cache maintenance and adaptivity

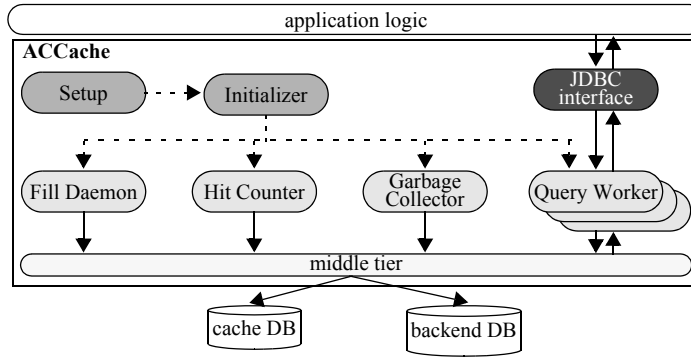

Fig. 3. Constraint-based caching system: Overview

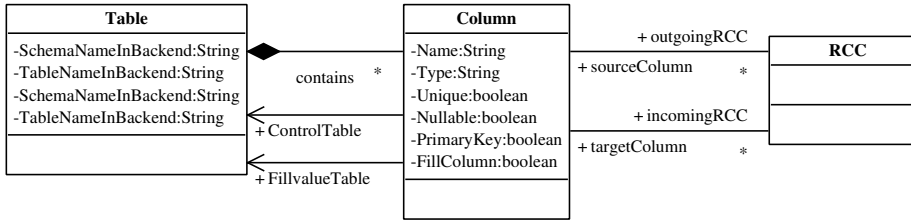

Fig. 4. Meta-data of the cache-related object types

is primarily accomplished by the Fill Daemon and the Garbage Collector whereas the Hit Counter is collecting reference statistics to enable accurate load/unload decisions.

3.2 Initializing DB Cache Processing

The Setup and Initializer components provide an administrator interface to ACCache. They enable the setup of a specific cache DB configuration and, for each operating session, the creation and initialization of appropriate data structures within ACCache. These are used for keeping meta-data for the cache tables (table and column names, column types, RCCs, filling columns, etc.) as well as statistics for cache operation control. The kernel part of the ACCache-internal data structure contains the object types illustrated in Fig. 4. The kind of information recorded is dependent on the role a column embodies. For example, if a filling column is specified, two additional table objects are created and referenced by this column object. The first one is a master control table carrying information about cached values in a filling column (value, loading time stamp, most recent reference, etc.), whereas the second one keeps all candidate values for the filling column (filling value table).

Cache DB setup requires the following essential steps:

– allocation of the specified cache tables and their related control tables: They can be created in any sequence, because FK relationships are not maintained in the cache (but only RCCs)

- specification of filling value tables
- creation of appropriate indexes (on source columns of RCCs and on U columns) to speed up probe queries.

An important optimization feature is the use of prepared statements for probing, filling, and other housekeeping operations on the cache. Because of their frequency, these SQL operations should be highly optimized and ready for running when needed. Because all possible operations are known in advance – once a cache DB configuration is fixed –, they can be prepared in the form of query execution plans (QEPs) and kept ready as soon as the cache DB is set up. The ACCache components accessing the cache DB via the middle tier require four different types of SQL statements:

- existence queries, primarily used by the Query Workers for probing
- insert statements used by the Fill Daemon to load new records into cache tables
- update queries to modify information in control tables
- delete statements to unload records from cache tables.

3.3 Query Worker

The Query Worker component (QW) is responsible for processing user queries and therefore provides the key functionality of ACCache. Several QW instances are managed in a pool at run time; a free QW is assigned to an arriving query and put back in the pool when finished. At first, a QW validates the request against a grammar of a subset of SQL. If no match is obtained, the query is passed on to the backend DB. Otherwise, local processing is initiated which is only sketched in its essential steps. Assume the following query is a potential candidate for cache processing (see Fig. 1):

```
SELECT d.name, m.title, g.name FROM Director d, Movie m, Genre g
WHERE d.id = m.directed_by AND m.genre = g.id AND d.id = '711'
ORDER BY g.name ASC
```

After checking for correct SQL syntax, the query is decomposed into its different clauses. At first, all table (and alias) names from the `FROM` clause are extracted. Then the `WHERE` clause is analyzed. For predicates of the form $column_i = column_j$ (equi-join predicate), QW checks whether an RCC exists between these columns. The data structure illustrated in Fig. 4 greatly supports the analysis. When an RCC is identified, QW creates/expands a directed graph – the so-called cache group evaluation graph (CEG) –, which receives the table names of the related columns as vertices. These table vertices are connected by a directed edge representing the direction of the RCC.

Figure 5 shows the result of the analysis process for our example. After two join predicates, QW extracts a predicate of the form $column_k = value$ which is considered as a potential entry point. Hence, QW initiates a probing process. If $column_k$ is a U column (like $d.id$), a probing query is sent to the cache DB. Otherwise, probing is performed on the source columns of incoming RCCs (see Sect. 2.2). The related existence queries probing potential entry points have the form

```
SELECT 1 FROM TABLE (VALUES 1) AS tmp
WHERE EXISTS (SELECT * FROM ⟨cache table⟩ WHERE ⟨column⟩ = ?)
```

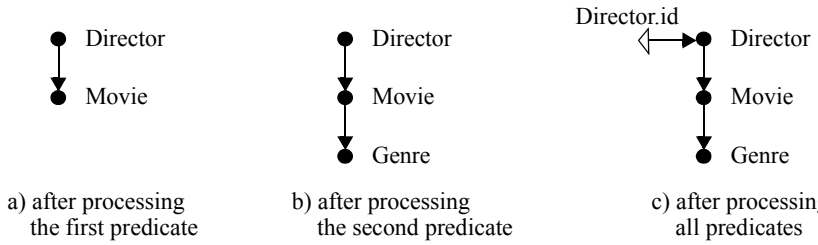

Fig. 5. Anchoring of cache tables

As soon as a complete value is determined (assume, $d.id =$ '711' is in the cache), probing stops successfully. In this case, the column can be used as an entry point for the query: It is taken as an anchor for the related table and added/connected to the CEG. Depending on the query analyzed, several entry points attached to table vertices may exist. Obviously, all table vertices reachable from an entry point are automatically anchored. Hence, CEG enables the generation of modified queries that are to be (partially) evaluated in the cache DB. For our example, the original query is rewritten to

```
SELECT d.name, m.title, g.name
FROM CA_Director d, CA_Movie m, CA_Genre g
WHERE d.id = m.directed_by AND m.genre = g.id AND d.id = '711'
ORDER BY g.name ASC
```

where the prefix `CA_` indicates a reference to a cache table. When the middle-tier component forwards the transformed query to the federated query facility, the entire query evaluation is performed in the cache DB in this case.

If probing fails, the value looked up is not complete in the cache. QW then checks whether the related column is a filling column and whether the value belongs to the candidate values. If so, a message is sent to the Fill Daemon to load this value into the cache.

3.4 Fill Daemon

Loading of records must be accomplished very carefully, that is, caching of duplicate records must be prevented and – after the filling process as a consequence of a CV reference is finished – all cache constraints must be satisfied by the state of the cache. The principal approach to loading predicate extensions has been discussed in Sect. 2.3. Here we outline its implementation.

Assume Actor name 'Bond' is included in the list of CVs, was referenced in a query, and was not found in the cache (see Fig. 1). Hence, the Fill Daemon will receive a message to make Actor name 'Bond' complete thereby loading the resp. predicate extension. Inserting 'Bond' into the control table implies loading the related Actor records which force Play and City records into the cache. The inserted Play records require the filling of Movie records and these, in turn, Genre and City records.

We have already mentioned that the insert statements for such a filling process are prepared by the Initializer component. However, these statements necessarily carry so-called markers (for actual parameters values) instead of concrete values. Hence, starting with the control table, we insert value 'Bond' and request all Actor records with name 'Bond' from the backend DB. These records are then inserted into the Actor table in the cache (bewaring of duplicates). Furthermore, they deliver the values replacing the markers in the prepare statements for Play and City, and so on.

Top-Down Filling. The filling process sketched so far iteratively loads a sequence of cache tables starting with the control table. This table sequence can be computed by recursively following the outgoing RCCs of each table visited. As an example, we list the insert statement for the Actor table:

```
INSERT INTO CA_Actor SELECT * FROM Actor a WHERE a.name = 'Bond'
AND a.name NOT IN (SELECT name FROM CA_Actor)
```

The entire filling process must be executed by a transaction whose insertions have to be protected by locks. Otherwise, parallel QWs could see inconsistent cache states which could lead to wrong query evaluations. For example, when inserting Actor record having $a.id =$ '007', the corresponding records are not present in cache table Play. Hence, "long" X locks must be kept until the filling process is successfully finished which, in turn, may block reader transactions for long time spans.

Bottom-Up Filling. A more sophisticated filling mechanism may avoid such situations. The key observation is that loading the cache tables bottom-up, we can fill each table in a separate transaction thereby providing cache consistency and only need to lock until the resp. cache table is loaded. More precisely, we have to define so-called atomic zones which can be loaded independently. In the simplest case, if no cycles are present, every cache table is an atomic zone. Due to space limitations, we cannot discuss cycle issues in detail and refer to [8]; suffice it to say that all tables belonging to an allowed cycle end up in the same atomic zone.

Figure 6 illustrates the atomic zones for the filling process of cache group G_1. The loading sequence of these zones can be determined by topological sorting which results for our example in: (Genre, City), Movie, Play, Actor, and finally the control table for *A.name*. Hence, after having finished loading of, say, cache table Genre, we can release the locks on Genre and let concurrent QWs run reader transactions on this table, and so on. However, to start the filling process with table Genre, we need to determine the records to be inserted. Therefore, we need to travel along the reverse RCC path from Genre up to Actor to select the Genre records depending on a CV to be filled in.

In the general case, the reverse RCC path be $R_n, R_{n-1}, \ldots, R_1$ where the target table of R_n is the cache table to be filled and the source table of R_1 is the root table. Then, the prepared insert statements have the following generic form:

```
INSERT INTO ⟨cache table⟩ (
  SELECT * FROM ⟨corresponding backend table⟩ WHERE ⟨R_n target col.⟩ IN (
    SELECT ⟨R_n source col.⟩ FROM ⟨R_n source table⟩ WHERE ⟨R_n-1 target col.⟩ IN (
    ... (SELECT ⟨R_1 source col.⟩ FROM ⟨R_1 source table⟩ WHERE ⟨filling col.⟩ = ?))))
```

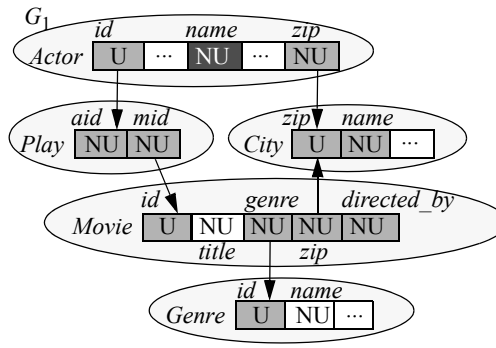

Fig. 6. Loading cache tables bottom-up

If a cache table is reachable by several RCC paths, it may receive records via all these paths. Therefore, prepared insertion statements are generated for all these paths – done in a stereotypical way, as shown below. Again, we illustrate the insertion statement for the first table to be loaded. The only marker to be replaced is 'Bond'.

```
INSERT INTO CA_Genre SELECT * FROM Genre g WHERE g.id IN (
  SELECT m.genre FROM Movie m WHERE m.id IN (
    SELECT p.mid FROM Play p WHERE p.aid IN (
      SELECT a.id FROM Actor a WHERE a.name = 'Bond'
))) AND g.id NOT IN (SELECT id FROM CA_Genre)
```

Hence, bottom-up filling provides a trade-off between potentially higher concurrency during the filling process and the need for more complex queries to be evaluated in the backend DB.

3.5 Hit Counter and Garbage Collector

The Hit Counter (HC) is responsible for recording statistical data used by the Garbage Collector (GC) for its cache replacement strategy. It is implemented as a separate process owning a queue continuously monitored and emptied. QWs fill this queue with messages recording each entry point found while a query was analyzed. In all control tables, HC maintains statistical information on the (candidate) values which triggered the load of those values identified as potential entry points for a query. In particular, the columns *hitcounter* and *lastaccess* are incremented or modified.

GC is responsible for controlling the size of the cached data by periodically checking whether or not a pre-specified cache filling level (high-water mark) is observed. If this level is reached, GC initiates one or more deletions of cache instances by removing a CV from a control table. As a consequence, the entire predicate extension for the removed CV has to be deleted from the cache thereby preserving the cache constraints. Thus, records belonging to multiple predicate extensions must not be deleted. In such cases, records can leave the cache only if the last predicate extension they belong to is removed from the cache.

Again, the prepared statements for delete operations are generated by the Initializer. The concrete CVs to be replaced, however, are chosen by means of an LRU algorithm. As a victim, GC selects the entry from a control table which has the least recent time stamp in the *lastaccess* column. This CV replaces the marker in the prepared statement.

Deletion starts from the control table removing the selected LRU CV and proceeds to all connected cache tables via outgoing RCCs. As in case of cache loading, we exclude the discussion of cycles. Assume, we want to remove the predicate extension for $D.name =$ 'Spielberg' in cache group G_2 (see Fig. 1). After 'Spielberg' is not in master control table, say K_1, anymore, the records in cache table Director are removed by:

```
DELETE FROM CA_Director
WHERE (name IS NOT IN (SELECT name FROM K1))
```

The deletion procedure has to follow all RCC paths starting from the root table. The prepared statements to be applied have the following generic form where corresponding expressions have to be generated and ANDed for each incoming RCC of a cache table. Hence, the base template is

```
DELETE FROM ⟨cache table⟩ WHERE (⟨RCC target column⟩
  IS NOT IN (SELECT ⟨RCC source column⟩ FROM ⟨RCC source table⟩))
```

which can easily applied to cache table Movie. Removing records from City with two incoming RCCs requires the following statement:

```
DELETE FROM CA_City
WHERE (zip IS NOT IN (SELECT zip FROM CA_Movie))
  AND (zip IS NOT IN (SELECT zip FROM Director))
```

At this point, you might wonder whether or not the deletion procedure is complete. What happens to RCC-dependent records in table Genre?

3.6 Savings and Penalties in Cache Group Federations

So far, we have discussed the management of single cache groups. As our running example in Fig. 1 reveals, it may be sometimes beneficial to allocate multiple overlapping cache groups in a federation. This design was influenced by the transparency requirements for cache tables which demand that each table (logically) appears only once in the cache.

For example, G_1 and G_2 share the tables Movie and City, which may save multiple representations of the same records. However, loading of records intended for one cache group may unintentionally cause records to be loaded in other cache groups. For example, table Genre only belongs to G_1. However, it is RCC-connected to table Movie to be loaded in G_1 and G_2. Hence, to preserve the RCC constraint, Genre may have to be filled whenever new records appear in table Movie. Therefore, loading a new predicate extension in G_2 may enforce records into Genre (in G_1) – only to satisfy all cache constraints.

Hence, if we load a new predicate extension into G_2, ACCache may have to insert Genre records, too, that is, new records in Movie may imply via RCC $M.genre \rightarrow G.id$

the insertion of (unwanted) Genre records. Symmetrically, deletion of a predicate extension in G_2 may remove records from Movie. To keep the cache consistent, Genre records may have to be deleted, too. Thus, deletion statements must cover all cache tables reachable by RCC paths starting from the root table of G_2.

The penalty each group in a federation must pay can be considered as a "membership fee". Separate allocation of cache groups, however, does not offer a perfect solution either. In such cases, we necessarily create copies in the cache which have to be kept consistent. For these reasons, savings and penalties of different solutions should be quantified, before a specific cache group design is chosen. Such an approach requires quantitative models for loading and unloading cache tables depending on specific workloads. A so-called cache group adviser could be a valuable tool for such design decisions. First steps in this direction are proposed by the authors in [9].

4 Summary

In this paper, we have primarily discussed the design and implementation issues of a middleware-based solution for database caching. For this reason, we have sketched our model for adaptive constraint-based caching and have emphasized the benefits and added value of this model as compared to the DBCache approach. The main part of our work has addressed our ACCache system which provides a database caching solution kept independent from specific DBMSs.

Our future work concentrates on optimization in ACCache. This includes the design of a suitable benchmark enabling representative performance measurements with comparable results and providing a refined exploration of federation issues. Moreover, these results could empower an adviser to support the specification of adequate configurations for cache groups and federations.

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 Conference. (2003) 718–729
2. Amiri, K., Park, S., Tewari, R., Padmanabhan, S.: DBProxy: A dynamic data cache for web applications. In: ICDE Conference. (2003) 821–831
3. Bühmann, A.: Einen Schritt zurück zum negativen Datenbank-Caching (A step back towards negative database caching). In: BTW Conference, Karlsruhe (2005) 107–124
4. Bornhövd, C., Altinel, M., Mohan, C., Pirahesh, H., Reinwald, B.: Adaptive database caching with DBCache. Data Engineering Bulletin **27**(2) (2004) 11–18
5. Larson, P., Goldstein, J., Zhou, J.: MTCache: Transparent mid-tier database caching in SQL server. In: ICDE Conference, IEEE Computer Society (2004) 177–189
6. Zhou, J., Larson, P., Goldstein, J.: Partially materialized views. Technical Report MSR-TR-2005-77, Microsoft Research (2005)
7. IBM: DB2 Universal Database (V 8.2) (2005)
8. Merker, C.: Konzeption und Realisierung eines Constraint-basierten Datenbank-Cache. Master's thesis, TU Kaiserslautern (2005)
9. Härder, T., Bühmann, A.: Database caching – Towards a cost model for populating cache groups. In: ADBIS Conference. Volume 3255 of LNCS., Budapest, Springer (2004) 215–229