Making the Most of Cache Groups

Andreas Bühmann and Theo Härder

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

Abstract. Cache groups are a powerful concept for database caching, which is used to relieve the backend database load and to keep referenced data close to the application programs at the "edge of the Web". Such cache groups consist of cache tables containing a subset of the backend database's data, guided by cache constraints. If certain query types are anticipated in the application workload, specifically designed cache groups can directly process parts of incoming declarative queries. The main class of such queries, project-select-join queries, can be supported by specifying a proper set of referential cache constraints.
Cache groups should be managed in the most cost-effective way. Hence, redundant constraints should not be respected during cache loading and consistency maintenance to avoid unnecessary overhead. On the other hand, because as much queries as possible should be processable in the cache, all redundant relationships implied by the set of specified cache constraints should be made explicit to help the query optimizer.

1 Database Caching with Cache Groups

Often, performance of data-intensive applications over wide-area networks (e. g., of transactional Web applications or Web information systems) is limited by the (backend) database (DB), especially by its processing power, its resource availability, and the communication delays for serving user requests. A proven remedy for such situations is the use of caching to substantially increase scalability and availability of the system as well as to drastically reduce the user-perceived delays of information requests.

Web caching, as another kind of caching in this context, [1] typically keeps static Web objects (XML fragments or images) in some of the caches in the user-to-server path and only enables identifier-based requests for cached objects. In contrast, DB caching is intended to deliver correct results for declarative DB queries (e. g., in SQL) from the current cache contents, thereby relieving the backend DB of some of its workload. Latency of user requests is supposed to be noticeably reduced by allocating these caches close to the application servers at the edge of the Web. This, however, only happens if user queries can be completely evaluated in the cache to save the travel times of messages (query shipping, result transmission) to the backend DB through wide-area networks. Hence, analysis of workloads must help to determine the future data reference behavior of applications and, in this way, prepare for appropriate locality of reference for them, to enable cache-based answering of frequent queries. This task is often facilitated by geographic contexts, which frequently determine the workload of application servers and, in turn, the data to be kept in their DB caches. Because these caches need powerful functionality for query optimization and processing, storage management, indexing, etc., they are often managed by full-fledged DBMSs and are therefore called frontend DBs, too.

Differing from approaches that make use of (stacked) materialized views [2], our cached data is organized in cache groups [3], which consist of a set of cache tables. These cache tables contain a subset of the backend DB's data, whose selection is guided by cache constraints – the approach therefore also being called *constraint-based DB caching*. It primarily rests upon referential cache constraints (RCCs), which specify the data sets needed to run selected project-select-join (PSJ) queries in the cache. Such specifications may be redundant or may contain RCC cycles, which imply cache groups exhibiting non-minimal maintenance or excessive loading or unloading [4].

Our specific contribution in this paper is to introduce a set of rules for proper cache group design and usage. In case of redundantly specified RCCs, our rules identify these redundant constraints, which will reduce cache maintenance overhead. On the other hand, our rules derive all redundant relationships implied by the set of specified cache constraints and make them explicit. This facilitates the query optimizer's task of figuring out all (parts of) queries that can be evaluated in the cache – besides those ones the cache groups are designed for.

The rest of the paper is organized as follows: In the following Sect. 2, we illustrate how cache groups are designed, how they are loaded, and how they are probed in order to determine whether a given query can be processed in the cache. Section 3 derives the set of rules that govern the optimization of cache groups, whereas we apply these rules to a sizeable example in Sect. 4 to demonstrate the course and the effects of this optimization process. Finally in Sect. 5, we summarize our results and give an outlook on our future work.

2 Designing Cache Groups

The key idea of constraint-based database caching is to accomplish *predicate completeness* for some given types of query predicates P in the cache such that all queries matching P can be evaluated correctly. This technique does not rely on static predicates: Parameterized constraints make the specification adaptive; so-called candidate values (CVs) are used to instantiate the corresponding 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. Hence, the candidate values should be carefully chosen, because they determine the set of cache-evaluable predicates. They describe the future reference locality anticipated in the cache and, therefore, serve as a kind of "loading directives" for the cache manager.

2.1 Basics of Cache Groups

A cache contains a collection of cache tables, which represent backend tables and which can either be isolated or related to each other in some way. For simplicity, let the table and column names be the same in the cache and in the backend DB: Considering a cache table S, S_B designates its corresponding backend table, $S.c$ a column c of S. All records (of various types) in the backend DB that are needed to evaluate predicate P are called the *predicate extension* of P.

For comprehension, let us repeat some definitions from [4]: The simplest form of predicate completeness is *value completeness*. A value v is said to be value complete (or *complete* for short) in a column $S.c$ if and only if all records of $\sigma_{c=v} S_\text{B}$ are in S. Hence, if we know that a value v is value complete in a column $S.c$, we can correctly evaluate $S.c = v$, because all records from table S_B that carry this value are in the cache. Furthermore, if we know that all values occurring in a column $S.c$ are complete, we call $S.c$ *column complete*. This property allows to evaluate all simple equality predicates $S.c = x$ in the cache as soon as a value x is found in $S.c$.

To answer PSJ queries in the cache, we must be sure that their extensions are present. Specific *equi-join predicates* can be evaluated only if all corresponding join partners are in the cache, which is enforced by using *referential cache constraints* (RCCs) [3]. An RCC is defined between two cache columns not necessarily belonging to separate tables. An RCC $S.a \rightarrow T.b$ 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 $T.b$.

This RCC ensures that, whenever we find a record s in cache table S, 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 perform this join in the cache correctly: 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 complete value can serve as an *entry point* into the cache for the evaluation of a query; 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 column is non-unique (NU) by default, but it can be declared unique (U) via the SQL constraint *unique* in the backend DB schema. Depending on the types of source and target columns, RCCs of types $1:n$, $n:1$, and $n:m$ may occur.

Probing is the process of finding out whether, given an equality predicate $S.c = v$ in a query, the value v is complete in column $S.c$. This knowledge is the foundation for applying RCCs along the join directions that occur in the query. There are basically two approaches to probing that can be combined to form probing strategies:

- If $S.c$ is known to be column complete, it suffices to check whether v exists in $S.c$. If it exists, it is complete.
- Otherwise RCCs can be exploited: If v exists in one of the source columns of RCCs leading to $S.c$, the value v is complete (in $S.c$).

2.2 Loading the Cache

How do we fill the cache? To initiate cache loading, we have to specify some filling columns $S.f$: Assume $x \in S_\text{B}.f$ is in the CV list and the cache manager wants to instantiate a cache constraint containing $S.f = x$. In a first step, x is made complete, which

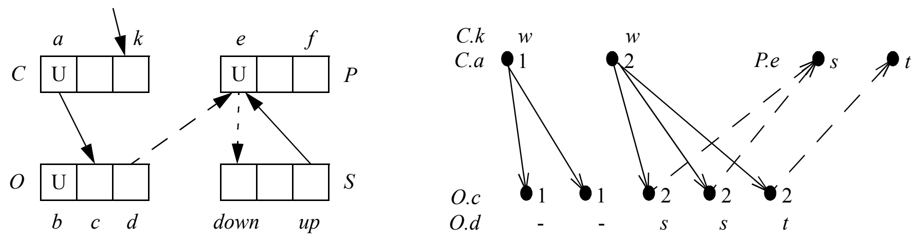


Fig. 1: Cache group *COPS*: Construction of a predicate extension for *COP*

loads a number of records into S. Then for each RCC $S.a \rightarrow T.b$ emanating from S, the newly inserted values in $S.a$ have to be made complete in $T.b$. Hence, new records are inserted into all target tables T_i reached by RCCs originating from S. In the same way, RCCs emanating from T_i provoke loading actions in further cache tables, until all RCC constraints are satisfied.

We can use cache tables, filling columns and RCCs to specify *cache groups*, which is our unit of design to support a specific predicate type in the cache. 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* and holds one or more filling columns. The remaining cache tables are called *member tables* and must be reachable from R via RCCs.

With these definitions, we are able to introduce predicate extensions for PSJ queries. First let us discuss the loading process in detail by an example: Cache group *COPS* (Customer, Order, Product, Structure) in Fig. 1, which includes two member and two owner constraints. For a moment forget table S and both RCCs between S and P. Then assume the predicate of a PSJ query to be evaluated on *COP* is

$$Q = (C.k = w \wedge C.a = O.c \wedge O.d = P.e)\,.$$

An example of Q's predicate extension is sketched in Fig. 1, where dots represent records, lines value-based relationships. To establish value completeness for the value w of filling column $C.k$, the cache manager loads all records of $\sigma_{k=w} C_{\mathrm{B}}$ in a first step. For each of these records loaded, the RCC $C.a \rightarrow O.c$ must be fulfilled (PK/FK relationships, solid lines); that is, all values of source column $C.a$ $(1,2$ in the example) must be made complete in the target column $O.c$. Finally, for all values present in $O.d$ (s,t), the RCC $O.d \rightarrow P.e$ makes their counterparts complete in $P.e$ (FK/PK relationships, dashed lines). Hence, we have constructed the predicate extension needed for Q exactly.

To make cache group design more elegant, we simplify our specification concepts: Those values of the CV list that have already initiated cache loading may be considered as values in artificial control columns and their relationships to filling columns may be described by RCCs. (For example, the RCC stub leading from nowhere to $C.k$ in Fig. 1 indicates such an RCC; we leave out the artificial columns in our figures.) With this unification of cache group specification, cache tables are loaded only via RCCs. Following the RCCs, the cache manager can construct predicate extensions using only simple loading steps based on equality of values. Accordingly, it can correctly evaluate the corresponding queries locally.

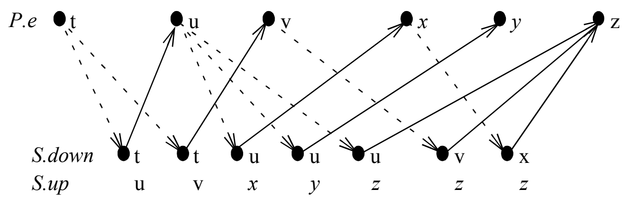


Fig. 2: Unsafe loading of products in *COPS*. Dots represent records, lines value-based relationships along RCCs.

We will show by an example that, for reasons of "safe" cache loading and maintenance, not all cache groups are acceptable: Assume we continue to load *COPS*, where tables *P* and (now) *S* contain the bill-of-material representation of products. As soon as value *t* is made complete in *P* (via RCC $O.d \rightarrow P.e$), it initiates loading in *S* via $P.e \rightarrow S.down$ to make *t* complete in *S.down*. In turn, this action loads values *u* and *v* into *S.up*, which enforces completeness for these value in *P.e* via $S.up \rightarrow P.e$. As illustrated in Fig. 2, cache loading recursively iterates over the RCC cycle and causes product *t* and its entire composed-of hierarchy to be loaded into the cache. Such excessive load situations are called *unsafe* and are prohibited when designing cache groups [3].

An RCC cycle is classified as *homogeneous* or *heterogeneous*, if it involves only a single column or more than one column in some participating table, respectively. If several cycles occur in a cache group and influence each other, some records loaded via a cycle may smuggle values into other cycles, which may keep these cycles running. Therefore, as proven in [4], while isolated homogeneous RCC cycles are acceptable, other cyclic RCC specifications must be prohibited to prevent unsafe cache groups:

- Isolated heterogeneous RCC cycles are not allowed.
- Heterogeneous RCC cycles with non-compensating smuggler relationships are not allowed.

The RCC cycle in cache group *COPS* (Fig. 1) is heterogeneous and isolated and should hence not be part of a cache group design.

3 Optimizing the Design

Making the most of a given cache group has two facets: First, when answering queries, we would like the query evaluation in the cache to be as powerful and flexible as possible. Second, when maintaining the cache contents – in order to fulfill the defined cache constraints – or when probing, we try to get by with the least possible effort.

3.1 Utilizing Redundancy

The path to both of these optimization goals lies in discovering redundancy in the cache group: Excluding redundant paths of loading steps during maintenance avoids unnecessary costs; including all possible (redundant) join directions enables the query analysis to use the cached predicate extensions for a greater variety of queries. Therefore, we need to know where redundant RCCs are.

Additionally, knowledge about column-complete columns as well as about redundant RCCs offers more and probably cheaper possibilities for probing [4]: Redundant RCCs need not be used during probing, and using a column-complete column, one is able to avoid considering RCCs altogether.

An RCC is called redundant if dropping it from the cache group does not change the cache group's behavior with regard to record loading: The same sets of records will be present in the cache in any situation after any number of loaded CVs, with or without the redundant RCC. Every RCC is either a *redundant RCC* (RRCC) or a *non-redundant RCC* (NRCC).

3.2 Optimization rules

In summary, given a cache group, we would like to find out

- which redundant RCCs can be added,
- which user-defined RCCs are in fact redundant and which are not,
- and which columns are complete.

Our goal is, on the one hand, to find an irreducible core of cache constraints that minimizes maintenance costs. On the other hand, we try to extend this core with a maximum of information that is useful during non-maintenance tasks.

To this end, we transform the user-specified cache group by applying a number of rules. These rules match certain situations in a cache group and may mark a column as column complete or introduce redundant RCCs. It is important that no rule ever changes the behavior of the cache group.

At the beginning of this optimization, due to the lack of better knowledge, we consider all user-defined RCCs non-redundant. When a newly discovered RRCC coincides with a user-defined NRCC, it effectively degrades the NRCC to an RRCC.

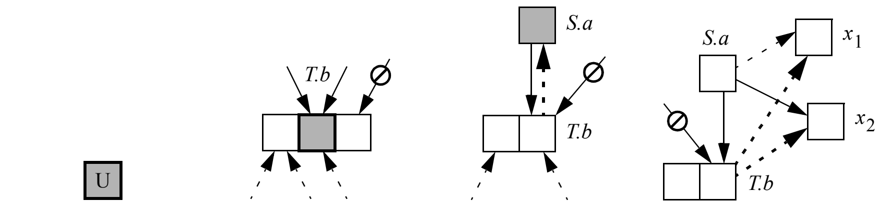
Figure 3 illustrates the situations in which our rules apply. The depicted tables and columns match the ones used in the textual descriptions of the rules below. We will walk through them one by one.

Unique Columns. We have two rules to discover complete columns. The first one is trivial, but it is needed nonetheless, because finding all complete columns is a prerequisite for successful application of some of the subsequent rules.

Rule 1. Every unique column is column complete. (Fig. 3a)

Every value in a unique column is complete as soon as it appears in the cache. Obviously, the column must always be complete then.

Induced Column Completeness. Our second rule deals with complete columns that are *induced* by RCCs and the loading mechanism.



(a) Unique columns (b) Induced column completeness (c) Inverse RCCs (d) RCCs to siblings

Fig. 3: Rules. Changes in the cache group are highlighted with thicker lines. The prohibition sign (⊘) marks exemplary RCCs that are not allowed for the rule to apply. (Complete columns are gray, redundant RCCs dotted.)

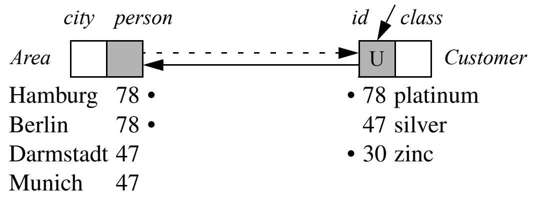


Fig. 4: Induced column completeness (of column *Area.person*) and inverse RCC (*Area.person* $\rightarrow$ *Customer.id*). Records marked with a dot (•) are in the cache.

Rule 2. Let $T.b$ be the only column of a table T that is reached by incoming NRCCs. Then $T.b$ is column complete. (Fig. 3b)

Every value that is loaded into T through one of the incoming NRCCs is complete in $T.b$. Since records are not loaded into T in another way (possibly existing RRCCs do not contribute to the loading), $T.b$ is column complete.

Let us look at a little example: Figure 4 shows a cache group comprising two cache tables *Customer* and *Area* (with their backend and cache contents) as well as an RCC *Customer.id* $\rightarrow$ *Area.person*. Table *Customer* is filled via column *id*. Customers 78 and 30 have been inserted, for each of which the corresponding *Area* records have been loaded: two records for 78 (making 78 complete), none for 30 (assuming 30 is not in $Area_{\mathrm{B}}$) which is therefore complete in *person* nevertheless. Therefore, *person* is column complete.

Column *person* would stay complete if another incoming RCC were added to it (and made 47 complete, for example). But if (Munich, 47) were loaded because of an RCC to *city*, it could not be guaranteed that the other 47 record would get into the cache, too. Hence, 47 would not be complete and neither would *person*. Note that any number of incoming RRCCs are acceptable; RRCCs do not contribute to the loading of cache tables and, thus, are unable to challenge column completeness.

Inverse RCCs. An RCC $x \to y$ expresses that every value in x is complete in y. We can discover additional RCCs if we are able to control the set of values present in x (and can then show that these values have to be complete in y). The simplest situation where it is clear which values appear in x is when x's table is loaded only via a single RCC $s \to x$ pointing to x. Then the values in x depend directly on the values in s: More precisely, x can only contain a subset of values of s. Therefore, we say that a column $T.b$ is *column dependent* on a column $S.a$ iff the only NRCC targeting table T is $S.a \to T.b$.

By comparing this definition with Rule 2, it is obvious that every column-dependent column is complete, but not every complete column is column dependent.

Let us return to our example in Fig. 4: There we have two column-dependent columns, *person* and *id*. We will concentrate on *person*: Due to the incoming RCC, it contains a subset (78) of the values in *id* $(30,78)$, which we know to be complete in *id*, because *id* has a unique constraint. Therefore, every value in *person* is complete in *id* and we can add an *inverse RRCC person* $\to$ *id*.

The colum *person* must not be reached by another NRCC (as opposed to our previous case of only column completeness), because a so-loaded 47 in *person* would not necessarily become complete in *id*.

We wrap up the sketched situation in our next rule:

Rule 3. Let $T.b$ be column dependent on a column $S.a$ due to an NRCC $S.a \to T.b$. If $S.a$ is column complete, then an inverse RRCC $T.b \to S.a$ holds. (Fig. 3c)

RCCs to Siblings. In special situations, two or more columns are in some sense synchronized due to RCCs originating from a common column. In Fig. 3d, this common column is $S.a$ and we have got three RCCs leading from it to some other (child) columns $T.b$, x_1, and x_2. (The RCC $S.a \to x_1$ is redundant, the other two are not.) This means that all the values in $S.a$ are complete in all of these three columns; let $V_{S.a}$ denote this set of values.

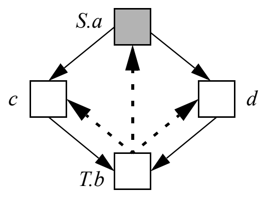
As we know from the discussion of Rule 3, column dependency of a column, say $T.b$, restricts the set of values in this column to a subset (of $V_{S.a}$). Hence, every value in column $T.b$ is complete in the children of $S.a$, which we can express by redundant RCCs from $T.b$ to its siblings. (Strictly speaking, we could also add a redundant RCC from $T.b$ to itself. But because such an RCC can be equivalently replaced with a columcompleteness label, we omit it: This would just be a special case of Rule 2.)

These thoughts leave us with the following rule:

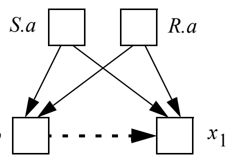
Rule 4. Let $T.b$ be column dependent on a column $S.a$ due to an NRCC $S.a \to T.b$. Then for every column c_i that is reached by an RCC $S.a \to c_i$ from the same source column (i. e., a sibling of $T.b$), an additional RRCC $T.b \to c_i$ holds. (Fig. 3d)

Possible Extensions. Our rules do not find every redundant RCC possible. We will discuss two conceivable generalizations of existing rules that would enable us to find more redundant RCCs.

The example shown in Fig. 5a generalizes the situation that is covered by our Rules 3 and 4: Column $T.b$ is reached by *two* different homogeneous paths (where there is no change of column in any table on the path), both emanating from $S.a$.



(a) Multiple paths from the same column



(b) Multiple synchronizing columns

Fig. 5: Harder optimization RCCs

This means that $T.b$ is not column dependent on $S.a$ according to our simple definition, but in a more general sense it is: The values in $T.b$ are still determined only by the values in $S.a$; on the paths towards $S.a$, more values may get lost than in our simple single-RCC case, but we still have a subset relationship. This means that an inverse RCC $T.b \rightarrow S.a$ is possible as well as RCCs from $T.b$ to the direct children of $S.a$ (which are no longer siblings of $T.b$).

Figure 5b shows a different kind of generalization of the concept of dependency: This time, the values in column $T.b$ depend on both the values in $S.a$ and the values in $R.a$ (i. e., at any time, $T.b$ contains a subset of the union of those values). This setting still permits RCCs to siblings to be added, as long as these siblings (e. g., x_1) are reached by RCCs from each of these synchronizing columns.

Rules expressing the sketched situations are not as easily checked as our chosen ones, which can consider a column and its immediate neighborhood locally. In contrast, here we would have to collect information about paths of any length and compare sets of influencing columns. It is questionable whether these special situations occur often enough to justify the added complexity of the rules for their optimization.

3.3 Applying the Rules

How do we apply our four rules to a given cache group? The basic idea is simple: Keep applying the set of rules until no further match occurs and the cache group is in a stable and, with regard to our rules, optimized state. Obviously, we must be sure that this will happen eventually: Our rule application algorithm should not run into endless cycles.

Let us analyze the dependencies among our rules: Rules 3 and 4 produce RRCC, which may override NRCCs. NRCCs eventually embody the irreducible core of the constraints; they are not produced by any rule. Since RRCCs are not removed, their number is only increasing, the number of NRCCs decreasing. This may at most lead to further columns becoming column dependent, which might make Rules 3 and 4 applicable again. This process is bounded by the number of feasible RCCs.

The first two rules only produce column-complete columns: Only Rule 3 depends on these column-complete columns. Since no rule removes the column-completeness status of a column, no cyclic behavior is possible – as long as we are careful enough to check whether a rule application did actually change the cache group.

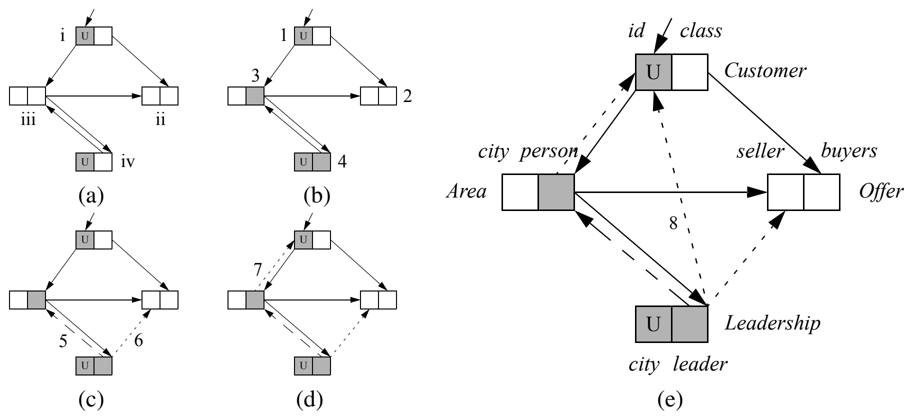


Fig. 6: Optimizing a cache group. The dashed lines indicate user-defined but redundant RCCs.

Unique columns cannot be created during optimization: Therefore, Rule 1 can be independently applied in advance, before the other rules are applied repeatedly until there are no further changes to the cache group.

In a Java implementation [5], we have chosen to apply our rules according to a depth-first search of the cache group, starting at the filling columns and stopping when cycles are detected. This is sufficient, because all tables not reachable this way will not be filled and used either. Furthermore, we are able to analyze the cycles encountered during rule application and see whether they lead to controllable loading behavior or make for unsafe cache groups.

4 Example

Let us see our rules acting in concert to optimize a given cache group. Figure 6 depicts our object of optimization (a) as well the optimization result (e); we will show step by step how this result has been derived.

We start with a cache group that has been specified by someone who wants to use our caching system for his online selling platform (Fig. 6a): We have four cache tables, two unique columns *Customer.id* and *Leadership.city*, one filling column *Customer.id* and six user-defined RCCs, which we have to consider non-redundant until further investigation. (In Fig. 6e these are the five solid RCCs and the dashed one.)

In a preparing step, we apply Rule 1 to every column in the cache group. The order in which we visit the cache tables in this and all the following steps is given in Fig. 6a in Roman figures: a depth-first search starting at the filling column $C.i$. (In the following, we abbreviate table and column names by their first letters.) In this way, we find the unique columns $C.i$ and $L.c$ and mark them as column complete.

We then begin to apply Rules 2–4 to the cache group (Fig. 6b):

1. $C.i$ is the only column of C that is reached by NRCCs; therefore, by Rule 2, it is column complete (which we already know, so this does not change the cache

group). $C.i$ is column dependent as well, but because it is dependent on an artificial column outside of our main cache group, we skip the other rules.

2. In table O no column is induced column complete or column dependent, because there are incoming NRCCs on two columns: None of our rules matches.
3. $A.p$ is reached by two NRCCs, but not any other column is: $A.p$ is column complete. Note that $A.p$ is *not* column dependent, because it is influenced by both $C.i$ and $L.l$.
4. $L.l$ is induced column complete because of the only NRCC $A.p \rightarrow L.l$, which makes $L.l$ column dependent, too.
5. Since $L.l$ is column dependent, we can add an inverse RRCC $L.l \rightarrow A.p$ (Rule 3). This degrades the already existing NRCC (indicated by a dashed line in Fig. 6c), recognizing it as redundant.
6. According to Rule 4 we can finally add redundant RCCs from $L.l$ to each of its siblings (with respect to the common father column $A.p$): Here this makes only for one RRCC $L.l \rightarrow O.s$.

This concludes our first run through the cache group; we have visited each table once. Since we have made four changes (two column-complete columns and two RRCCs), and because these might have established the preconditions for further rule applications, we have to start a second time: In tables C and O we come across the same states as before, but in A we find something new:

7. Column $A.p$ has become column dependent on $C.i$ due to the degradation of the former NRCC $L.l \rightarrow A.p$. This means – according to Rule 2 – that we can add an inverse RRCC $A.p \rightarrow C.i$ (Fig. 6d). Furthermore, we could add RRCCs to children of $C.i$ if there were any besides $A.p$.
8. In table L column $L.l$ is still column dependent on $A.p$ – as discovered in step 4. (A column can never lose its column dependency.) In step 6, we have already applied Rule 4 and introduced RRCCs to all siblings of $A.p$ – but wait, there is a new sibling, namely $C.i$, due to the recently created RRCC $A.p \rightarrow C.i$. Hence, we can add an RRCC $L.l \rightarrow C.i$ back to the *Customer* table (Fig. 6e).

Our second run through the cache group is finished. We have added two RRCCs and must therefore perform a third run to see whether these changes have opened up further possibilities. You should be able to verify that this is not the case. Accordingly, the state in Fig. 6e is our optimized version of the cache group the user has defined:

– We have identified three additional RCCs, which, during query analysis and evaluation, allow for more join directions in the cache. For example, the predicate $L.c =$ 'Berlin' $\wedge L.l = C.i$ can be evaluated in the cache, given that $L.c$ can be probed successfully for 'Berlin'.
– We have revealed that RCC $L.l \rightarrow A.p$ is actually redundant and thus need not be checked during cache loading or probing operations. We could also warn the user about this redundancy in his design, either when loading his complete specification into our caching system or in advance, when the user is designing his cache group assisted by a cache group adviser that implements our rules.
– Finally we have discovered four column-complete columns (among them admittedly two trivial ones): These promise more flexibility in choosing the cheapest probing strategy.

5 Conclusion

In this paper, we have presented four simple optimization rules that can be applied to a cache group after it has been designed. These rules do not touch the loading behavior, but make redundant information explicit that is contained in or derivable from the given cache group design. Furthermore, during optimization, unsafe cache groups can be detected. This stock of information allows the cache manager to perform his tasks of loading, unloading, probing, and query evaluation more efficiently.

Alternatively, this information could be fed back interactively to the designer of a cache group to make him aware of the consequences of his decisions. Another type of information that would be useful in this setting is estimates about the loading costs of predicate extensions.

Our rules find the most useful redundant RCCs in situations that occur frequently. We have demonstrated which constellations in cache groups lie beyond the capabilities of our rules and how the rules could be extended to cope with those.

We have already implemented a DB-caching prototype called ACCache [6], which relies on our constraint-based caching model. It is realized on top of an existing relational DBMS and leverages its federated query execution capabilities. Within ACCache we can fill the cache; analyze, rewrite, and execute queries (partially) in the cache or in the backend DB; collect statistics about the usage of specific predicate extensions; and we can perform garbage collection based on these statistics. The making use of redundant RCCs and column-complete columns during this tasks is still to be added.

At the moment, we are developing an automated measurement environment, which will enable us to perform comparative benchmarks in order to assess quantitatively the actual benefit of our cache group optimization rules presented in this paper – among other aspects, such as the costs of loading and unloading predicate extensions or the overhead of probing, always in comparison to the lower latencies or reduced backend loads achievable.

References

1. Podlipinig, S., Böszörmenyi, L.: A survey of web cache replacement strategies. ACM Computing Surveys **35**(4) (2003) 374–398
2. Larson, P., Goldstein, J., Zhou, J.: MTCache: Transparent mid-tier database caching in SQL server. In: ICDE Conference, IEEE Computer Society (2004) 177–189
3. 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
4. Härder, T., Bühmann, A.: Value complete, column complete, predicate complete – Magic words driving the design of cache groups. VLDB Journal (2006) Accepted for publication.
5. Scholl, W.: Cache-Group-Optimierung zur Effizienzsteigerung von Datenbank-Caches. Project thesis, TU Kaiserslautern (2006) http://wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Sch06.PA.pdf.
6. Bühmann, A., Härder, T., Merker, C.: A middleware-based approach to database caching. In Manolopoulos, Y., Pokorný, J., Sellis, T., eds.: ADBIS 2006. Volume 4152 of LNCS., Thessaloniki (2006) 182–199