Database Caching – Towards a Cost Model for Populating Cache Groups

Theo Härder and Andreas Bühmann

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

Abstract. Web caching keeps single Web objects ready somewhere in caches in the user-to-server path, whereas database caching uses full-fledged database management systems as caches to adaptively maintain sets of records from a remote database and to evaluate queries on them. Using so-called cache groups, we introduce the new concept of constraint-based database caching. These cache groups are constructed from parameterized cache constraints, and their use is based on the key concepts of value and domain completeness. We show how cache constraints affect the correctness of query evaluations in the cache and which optimizations they allow. Cache groups supporting practical applications must exhibit controllable load behavior for which we identify necessary conditions. For such safe cache groups, the cost trade-off for record loading and predicate evaluation saving has to be observed during their design. Therefore, we analyze their load overhead and propose a population estimation algorithm to be used for a cache group advisor.

1 Introduction

Transactional Web applications (TWAs) in various domains (often called e*-applications) dramatically grow in number and complexity. At the same time, each application faces increasing demands regarding data volumes and workloads to be processed efficiently. In such situations, caching is a proven concept to improve response time and scalability of the applications as well as to minimize communication delays in wide-area networks. For this reason, a broad spectrum of techniques has emerged in recent years to keep static Web objects (like HTML pages, XML fragments, or images) in caches in the user-to-server path (client-side caches, proxies of various types, CDNs).

As the TWAs must deliver more and more dynamic and frequently updated content, this so-called *Web caching* [9,11] should be complemented by techniques that are aware of the consistency and completeness requirements of cached data (whose source is dynamically changed in backend databases) and that, at the same time, adaptively respond to changing workloads. Attempts targeting these objectives are called *database caching*, for which several different solutions have been proposed in recent years [2,3,4]. Currently many database vendors are developing prototype systems or are just extending their current products [8,10].

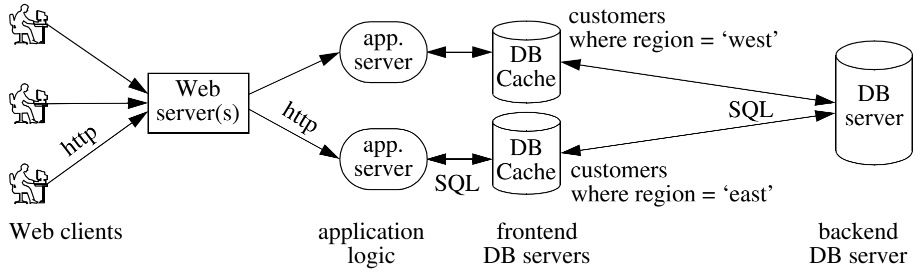

Fig. 1. Database caching for Web applications

What is the technical challenge of all these approaches? When user requests require responses to be assembled from static and dynamic contents somewhere in a Web cache, the dynamic portion is generated by a remote application server, which in turn asks the backend DB server for up-to-date information, thus causing substantial latency. An obvious reaction to this performance problem is the migration of application servers to data centers closer to the users: Figure 1 illustrates that clients select one of the replicated Web servers "close" to them in order to minimize its response time. This optimization is amplified if the associated application servers can instantly provide the expected data – frequently indicated by geographical contexts. But the displacement of application servers to the edge of the Web alone is not sufficient; conversely it would dramatically degrade the efficiency of DB support because of the frequent round trips to the then remote backend DB server. As a consequence, primarily used data should be kept close to the application servers in so-called DB caches. A flexible solution should not only support database caching at mid-tier nodes of central enterprise infrastructures [10], but also at edge servers of content delivery networks or remote data centers.

Another important aspect of a practical solution is to achieve full *cache transparency* for the applications, i. e., modifications of the application programming interface are not tolerated. Such a property gives the cache manager the choice at run time to process a query locally or to send it to the backend DB server, e. g., to comply with strict consistency requirements. Cache transparency typically requires that each DB object is represented only once in a cache and that it exhibits the same properties (name, type, etc.) as in the backend.

The use of SQL implies another challenge because of its declarative and set-oriented nature. This means that, to be useful, the cache manager has to guarantee that queries can be processed in the DB cache, i. e., the sets of records (of various types) satisfying the corresponding predicates – denoted as *predicate extensions* – must be completely in the cache. This *completeness condition*, the so-called *predicate completeness*, ensures that the query evaluation semantics is equivalent to the one provided by the backend.

A federated query facility [2,8] allows cooperative predicate evaluation by multiple DB servers. This property is very important for cache use, because local evaluation of some (partial) predicate can be complemented by the work of the backend DB server on other (partial) predicates whose extensions are not in the cache. Hence, in the following we refer to predicates meaning their portions to be evaluated in the cache.

2 Constraint-based Database Caching

We take a look at the concepts developed and realized in the DBCache project [2] and explore the underlying ideas. This work has lead us to a class of techniques which we term *constraint-based database caching* [7]. In particular, we analyze techniques which support the evaluation of specific PSJ queries (projection-selection-join queries) in the cache.

For the specification of cache contents, we refer to a particular approach called *cache groups*. In short, a cache group is a collection of related cache tables. Cache constraints defined on and between them determine the records of the corresponding backend tables that have to be kept in the cache. The technique does not rely on the specification of static predicates: The constraints are parameterized, which makes this specification adaptive; it is completed when the parameters are instantiated by values of so-called cache keys. An "instantiated constraint" then corresponds to a predicate and, when the constraint is satisfied – i. e., all related records have been loaded – the predicate extension delivers correct answers to eligible queries.

The key idea of constraint-based caching is to start with very simple base predicates (here equality predicates) and to extend them by other types of predicates (equi-join predicates in our case) in a constructive way, such that cache maintenance can always guarantee the presence of the corresponding predicate extensions in the cache. Hence, there are no or only simple decidability problems whether a complete predicate evaluation can be performed. Only a simple probe query is required in the cache at run time to determine the availability of eligible predicate extensions. Furthermore, because all columns of the corresponding backend tables are kept in the cache, all project operations possible in the backend can also be performed in the cache thereby enabling PSJ queries. Since full DB functionality is available, the results of these queries can further be refined by operations like group-by, having, or order-by.

2.1 How Do Cache Groups Work?

As introduced above, a cache group is a collection of related cache tables. For simplicity, the names of tables and columns are identical in the cache and in the backend DB. Considering a cache table S, we denote by S_{B} its corresponding backend table, by $S.c$ a column c of S. Note, a cache usually contains only subsets of records pertaining to a small fraction of backend tables. Its primary task is to support query processing for TWAs which typically contain up to 3 or 4 joins [2]. Hence, we expect the number of cache tables – featuring a high degree of reference locality – to be in the order of 10 or less, even if the backend DB consists of hundreds of tables.

If we want to be able to evaluate a given predicate in the cache, we must keep a collection of records in the cache tables such that the completeness condition for the predicate is satisfied. For simple equality predicates like $S.c = v$ this completeness condition takes the shape of *value completeness*.

Definition 1 (Value completeness, VC). *A value v is said to be value complete in a column $S.c$ if and only if all records of $\sigma_{c=v}S_{\mathrm{B}}$ are in S.*

If we know that a value v is value complete in a column $S.c$, we can correctly evaluate $S.c = v$, because all rows from the corresponding backend table S_{B} that carry that value are in the cache. But how do we know that v is value complete? This is easy if we maintain *domain completeness* of specific table columns.

Definition 2 (Domain completeness, DC). *A column $S.c$ is said to be domain complete (DC) if and only if all values v in $S.c$ are value complete.*

Given a domain-complete column $S.c$, if a probe query confirms that value v is 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. Note that unique (U) columns of a cache table (defined by SQL constraints "unique" or "primary key" in the backend DB schema) are DC per se (*implicit* domain completeness). Non-unique (NU) columns in contrast need extra enforcement of DC.

So far, we can evaluate only equality predicates, i. e., simplest selection queries, in the cache. To enhance such queries with equi-join predicates, we introduce *referential cache constraints* (RCCs), which guarantee the correctness of equi-joins between cache tables. Such RCCs are specified between two cache table columns: a source column $S.a$ and a target column $T.b$. The tables S and T need not be different, not even the columns themselves.

Definition 3 (Referential cache constraint, RCC). *RCC $S.a \rightarrow T.b$ between columns $S.a$ and $T.b$ is satisfied if and only if all values v in $S.a$ are value complete in $T.b$.*

RCC $S.a \rightarrow T.b$ ensures that, whenever we find a record s in S, all join partners of s with respect to $S.a = T.b$ are in T. Note, the RCC alone does not allow us to correctly perform this join in the cache: Many rows of S_{B} that have join partners in T_{B} may be missing from S. But using an equality predicate on a DC column $S.c$ as an "anchor", we can restrict this join to records that exist in the cache: RCC $S.a \rightarrow T.b$ expands the predicate extension of $S.c = x$ to the predicate extension of ($S.c = x$ and $S.a = T.b$). In this way, DC columns serve as *entry points* for queries.

Domain completeness of a column $S.c$ is equivalent to an RCC $S.c \rightarrow S.c$, a so-called *self-RCC* on its defining column $S.c$. By specifying such a self-RCC, the DBA can enforce domain completeness of $S.c$ and thus create an entry point for query evaluation explicitly.

How do the records that constitute a predicate extension get into the cache? And how are these predicate extensions actually chosen? For these tasks, we introduce the second kind of cache constraint, the so-called *cache key.*

Definition 4 (Cache key). *A cache key column $S.k$ is always kept domain complete. Only values in $\pi_k S_{\mathrm{B}}$ initiate cache loading when they are referenced by user queries.*

You can imagine that the specification of a cache key includes a self-RCC; similar to it, a cache key can always be used as an entry point. (In both cases the columns get *explicitly* domain complete.) But in addition, a cache key serves as a *filling point* for a distinguished *root table* R (the only table in a cache group that contains cache keys) and – via the (paths of) RCCs specified between R and related cache tables – for the *member tables* of the cache group. Whenever a query references a particular cache key value that is not in the cache, the backend DB must evaluate this query. But as a consequence of this cache miss attributed to a cache key, the cache manager satisfies the value completeness for the missing cache key value by fetching all required records from the backend and loading them into the cache table R (thus keeping the cache key column domain complete). To satisfy the RCCs, the member tables of the cache group are loaded in a similar way (details are provided in [2]). Hence, a reference to a cache key value x serves as something like an indicator that, in the immediate future, locality of reference is expected on the predicate extension determined by x. Cache keys therefore carry information about the future workload and sensitively influence caching performance. Hence, DBAs must select them carefully[1].

2.2 Types of RCCs and Their Use in Cache Groups

Depending on the types of the source and target columns, we classify RCCs as (1: 1), (1: n), (n: 1), and (n: m) and denote them as follows:

– U $\rightarrow$ U or U $\rightarrow$ NU: member constraint (MC)
– NU $\rightarrow$ U: owner constraint (OC)
– NU $\rightarrow$ NU: cross constraint (XC).

Using RCCs we implicitly introduce something like a value-based table model intended to support queries. Despite similarities, MCs and OCs are not identical to the PK/FK (primary key/foreign key) relationships in the backend DB: Those can be used for join processing symmetrically, RCCs only in the specified direction. XCs have no counterparts at all. Because a high fraction of all SQL join queries refers exclusively to PK/FK relationships – they represent real-world relationships captured by the DB design –, almost all RCCs are expected to be of type MC or OC; accordingly XCs and multiple RCCs ending on a NU column seem to be rare.

[1] Low-selectivity cache key columns may cause cache filling actions involving huge sets of records never used later. It may therefore be necessary to control the use of cache key values with stop-word or recommendation lists.

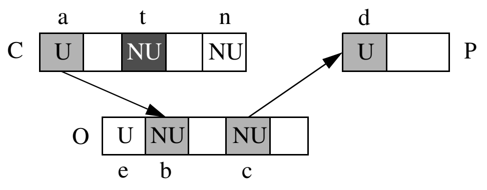

Fig. 2. Cache group OP for order processing

Assume a cache group OP with tables C, O, and P and cache key $C.t$, formed by $C.a \rightarrow O.b$ and $O.c \rightarrow P.d$, where $C.a$ and $P.d$ are U columns and $C.t$, $O.b$ and $O.c$ are NU columns (Fig. 2). In a common real-world situation, C, O, and P could correspond to backend DB tables Customer, Order and Product. Hence, both RCCs would typically characterize PK/FK relationships that will be used for join processing in the cache. Additional RCCs, for example, $C.t \rightarrow O.b$ or $O.c \rightarrow C.n$, are conceivable; such RCCs, however, have no counterparts in the backend DB schema and, when used for a cross join of C and O, their contributions to the query semantics remain in the user's responsibility.

As we know, if a probing operation on some domain-complete column $T.c$ identifies value x, we can use $T.c$ as an entry point for evaluating $T.c = x$. Now, any enhancement of this predicate with equi-join predicates is allowed if these predicates correspond to RCCs reachable from cache table T.

Assume, we find 'gold' in $C.t$ (of cache group OP), then the predicate ($C.t =$ 'gold' and $C.a = O.b$ and $O.c = P.d$) can be processed in the cache correctly. Because the predicate extension (with all columns of all cache tables) is completely accessible, any column may be specified for output. Of course, a correct predicate can be refined by "and-ing" additional selection terms (referring to cache table columns) to it; e. g. ($C.t =$ 'gold' and $O.e > 42$ and $C.n$ like 'Smi%' and ...).

3 Cache Group Design and Analysis

At this point, we know how to configure a cache group by specifying the participating tables, the RCCs connecting them, and the cache keys, which initiate the population of the cache group. We can use domain-complete columns as entry points to obtain correct query results for eligible query predicates. Is this all we need to know to design and to effectively make use of cache groups?

On the one hand, a cache group should enable as flexible use for predicate evaluation as possible: We should not leave any entry point or RCC unexploited. This requires that we know about all of them, not just about those we specified explicitly. On the other hand, we want to design safe cache groups which exhibit controllable load behavior.

Definition 5 (Reachability graph). *A reachability graph γ is a directed graph implicitly defined for a cache group G. It has G's tables and RCCs as nodes and edges. γ is composed by starting from G's root table and following all RCCs transitively thereby connecting all reachable tables (as nodes of γ).*

Definition 6 (Paths and cycles). *A path in a reachability graph starts at a source table and ends at a sink table. It connects a collection of cache tables via a sequence of RCCs. No RCC may appear twice in a path. A cycle is a path that starts and ends at the same table.*

Definition 7 (Homogeneous and heterogeneous cycles). *A cycle is homogeneous, if only a single column per table is involved, heterogeneous otherwise.*

Heterogeneous RCC cycles can lead to excessive population of cache groups primarily caused by recursive filling actions. Such "dangerous" load behavior must clearly be identified and prevented.

3.1 Entry Points for Query Evaluation

So far, we have argued that a cache table column can be tested and used by an equality predicate correctly only if it is domain complete. But how do we know that? Of course, cache table columns that carry either a self-RCC or a cache key (i. e., at least all filling points) are explicitly domain complete; columns of type U are implicitly domain complete. Cache-supported query evaluation gains much more flexibility and power, if we can correctly decide that other cache columns are domain complete as well.

Let us refer again to OP. Because $C.a \rightarrow O.b$ is the only RCC that induces filling of O, we know that $O.b$ is domain complete (denoted as *induced domain completeness*). Hence, we can correctly evaluate the query predicate ($O.b = y$ and $O.c = P.d$) if we encounter value y in $O.b$ – in addition to ($C.a = x$ and $C.a = O.b$ and $O.c = P.d$) if value x is in $C.a$.

Note, additional RCCs ending in $O.b$ would not destroy the DC of $O.b$, though any additional RCC ending in a column different from $O.b$ would do[2]: Assume an additional RCC ending in $O.e$ induces a new value v, which implies the insertion of $\sigma_{e=v}O_{\mathrm{B}}$ into O – just a single record o. Now a new value w of $o.b$, so far not present in $O.b$, may appear, but all other records of $\sigma_{b=w}O_{\mathrm{B}}$ fail to do so. For this reason, a cache table filled by RCCs (or cache keys) on more than one column cannot have an induced DC column. This means that induced DC is *context dependent*; in contrast to explicit or implicit DC (i. e., DC of cache key, self-RCC, or U columns), it can be lost when a cache group configuration is modified. This leads us to the following definition:

Definition 8 (Induced domain completeness). *A cache table column is induced domain complete, if it is the only column of a cache table filled*[2] *via one or more RCCs or via a cache key definition.*

Let us summarize our findings concerning the population of cache tables and the domain completeness of their columns: A cache table T can be filled via cache key columns or RCCs ending in one or more of its columns. A column

[2] We must distinguish between RCCs that only reach a column and RCCs that fill it: RCCs that never cause any record to be loaded (e. g., a self-RCC on a U column) do not disturb induced DC. How to effectively classify an arbitrary RCC is unsettled.

of T can be domain complete due to specifications in the backend (implicitly: U columns), due to specifications in the cache (explicitly: cache keys, self-RCCs), or as a result of the interaction of specified items (induced).

Analogous to extra DC columns, one can discover *optimization RCCs* in a cache group, i. e., RCCs that have not been specified, but hold in the given context. For example, in OP the (optimization) RCC $O.b \rightarrow C.a$ allows an additional join direction.

3.2 Safeness of Cache Groups

It is unreasonable to accept all conceivable cache group configurations, because cache misses on cache key columns may provoke unforeseeable load operations. Although the cache can be populated asynchronously to the transaction observing the cache miss, avoiding a burden on its response time, uncontrolled loading is undesirable: Substantial extra work, which can hardly be estimated, may be required by the frontend and backend DB servers, which will influence the transaction throughput in heavy workload situations.

Specific cache group configurations may even exhibit a recursive loading behavior that jeopardizes their caching performance. Once cache filling is initiated, the enforcement of cache constraints may require multiple phases of record loading. Such behavior typically occurs, when two NU-DC columns a and b of a cache table X must be maintained. A set of values appears in a, for which X is loaded with the corresponding records of X_{B} to keep column a domain complete. These records populate b with a set of (new) values that have to be made value complete, which possibly introduces new values into a again. As a result, a and b may receive new values in a recursive way.

Cache groups are called *safe* if no recursive load behavior is possible. Upon a cache key miss, we want the initiated cache loading to stop after a *single pass* of filling operations through the tables of the cache group.

Obviously, recursive loading requires that there is a cyclic structure among the specified RCCs (remember, every cache key also contains an RCC). Simple examples show that there are not only unsafe RCC cycles, but also safe ones (consider a homogeneous cycle) [2,7]. We analyzed cycles in detail and derived safeness conditions for cache group configurations. These conditions are more sophisticated than a simple exclusion of pairs of NU-DC columns (as sketched above), because the mutual introduction of new values can span several tables and can also be neutralized by compensating effects. Nevertheless the safeness conditions can be stated as a single rule that requires the designer of a cache group to inspect all contained cycles for certain patterns of U and NU columns.

4 Loading Behavior of Cache Groups

So far, we have discussed the correctness and safeness conditions of cache groups. To analyze the loading behavior, we will derive a simple quantitative cost model. It is aimed at estimating the approximate cost (depending on column selectivities and RCCs specified) for the population of a cache group caused by a single cache key value.

4.1 Model Assumptions

For quantitative modeling, we generally assume uniform value distribution in columns and stochastic independence between columns (i. e., the standard assumptions of query processing). Each column of a cache table T inherits the cardinality of the corresponding column of backend table T_B. Hence, $T.j$ has cardinality $c_{T.j}$ (i. e., it has up to $c_{T.j}$ distinct values). We define the selectivity of column $T.j$ to be $s_{T.j} = 1/c_{T.j}$. Thus, the smaller the value $s_{T.j}$, i. e., the larger the value $c_{T.j}$, the higher is the column selectivity. For example, if T contains N_T records, an equality predicate on $T.j$ qualifies $N_T \cdot s_{T.j} = N_T/c_{T.j}$ records ($1 \leq c_{T.j} \leq N_T$) implying that NULL values are excluded.

When n_T records are filled into a table T, e. g., to satisfy a cache constraint on a given column $T.i$, how many distinct values d are entered into a stochastically independent column $T.j$? The result for the boundary values is obvious: If $T.j$ is unique, $d = n_T$, and if the cardinality of $T.j$ is 1, $d = 1$. In general, an accurate determination of d demands for a stochastic model which evaluates the expected number of distinct values of $T.j$ [12]. In abstract terms: Given natural numbers N, c, n ($1 \leq c \leq N, n \leq N$), what is the expected number d of colors when n balls are drawn without replacement from a bucket with N balls. These balls occur in c different colors and are uniformly distributed, i. e., there are N/c balls per color. The following model, which we have derived for the sketched situation, is used throughout the paper and referenced by $f(N, c, n)$:

$$d = f(N, c, n) = c \cdot \left(1 - \binom{N - N/c}{n} \Big/ \binom{N}{n}\right) .$$

In frequent situations, more than one record set is independently filled into table T. Instead of computing the sum of the various set sizes, we could improve our population estimation by modeling such situations of a combination of events. Then the expected size n_T of T's population induced by m independent cache constraints could be calculated with the following considerations. If $A_1, \ldots, A_m$ are m events, what is the probability of the event that at least one among the m A_j occurs. In symbols, this event is $A_{1\ldots m} = A_1 \cup A_2 \cup \cdots \cup A_m$. It is not sufficient to know the probabilities of the individual events A_j, but we must have complete information concerning all possible overlaps. Fortunately, due to the stochastic-independence assumption, we can easily compute for each pair (i, j), each triple (i, j, l), etc., the probability of events $A_i \cap A_j$ or $A_i \cap A_j \cap A_l$, etc. Furthermore, we can compose our formula iteratively, thereby computing the probabilities of the following events [5]: $(A_1 \cup A_2), (A_{12} \cup A_3), (A_{123} \cup A_4), \ldots$.

This abstract model for the combination of events can easily be applied to our problem of determining the number of distinct records when independent record sets are filled into a table. By multiplying the (filling) probabilities with table cardinality N_T, we immediately yield $n_{T1\ldots m}$ as the number of distinct records for m overlapping record sets. If n_{T1} and n_{T2} records are to be filled independently into table T, n_{T12} is the number of records actually loaded, etc.:

$$\begin{aligned} n_{T12} &= n_{T1} + n_{T2} - n_{T1} \cdot n_{T2}/N_T \ , \\ n_{T123} &= n_{T12} + n_{T3} - n_{T12} \cdot n_{T3}/N_T \ , \\ n_{T1234} &= n_{T123} + n_{T4} - n_{T123} \cdot n_{T4}/N_T \ , \ldots \end{aligned}$$

The rationale of our somewhat simplified estimation model, the calculation of average record populations in cache tables, is considered to be a great help for cache group design, e. g., when applied by a cache group advisor. A model refinement is only possible at the expense of substantially increased model complexity. The actual value distributions in columns and the size of record sets induced by RCCs (equivalently, (intermediate) join results) could be approached more accurately by introducing histograms, describing the frequency of individual values or of values belonging to value ranges, and join selectivities.[2] This would require additional and more accurate statistical data for cache tables and cache groups which is, due to its dynamic nature, hard to derive and to maintain.

4.2 Effective Cache Keys and Applicable RCCs

To keep the population model simple, but at the same time as accurate as possible, we need the "right" concepts. As argued in the following, two essential concepts for the population estimation of a cache group G are *applicable RCC* and *effective cache key.*

Any filling action in a cache group is path-dependent and depends on the type of RCC traversed. For example, an optimization RCC does not change G's filling and need therefore not be considered for G's population estimation. (In Fig. 3, $Q.e \rightarrow O.a$ and $V.g \rightarrow O.a$ would be such optimization RCCs.) Otherwise, we would have to deal with MC $\rightarrow$ OC cycles (the reverse owner constraint for an already traversed member constraint) adding unnecessary complexity. In Fig. 3, all four RCCs shown are applicable for the population estimation.

Conversely, if a reverse *member* constraint is specified explicitly, this RCC is considered a design error. The resulting homogeneous cycle would only lead to excessive load situations without benefiting the transaction's queries.

When more than one cache key is specified for a root table, we can always reduce such a set of cache keys to a single effective cache key (ck_{eff}) as far as cost estimations for the filling process are concerned.[3] ck_{eff} is the only non-unique cache key, if any; otherwise, it is any unique cache key. When $O.a$ and $O.k$ are defined as cache keys in Fig. 3, then $O.k$ is the effective one.

[2] However, this would only improve certain situations. Since values in a column, distributed according to a histogram, are used in RCCs which enforce the filling of these values in columns of other tables, the model complexity seems to be very hard to control.

[3] Consider two cache keys, $T.u$ unique and $T.n$ non-unique. If a value of u causes a cache miss, the single qualified record is loaded into the cache. The new value of $T.n$ has to be made value complete which determines the set of records to be loaded (except in the case of a NULL value in $T.n$, which we exclude from our estimations).

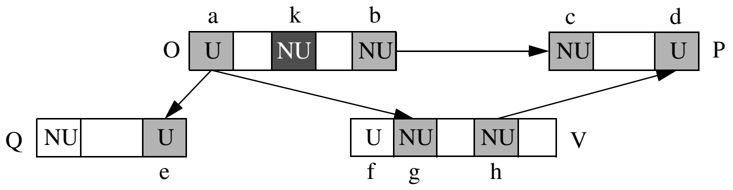

Fig. 3. Cache group G

The column ck_{eff} determines the number n_S of records to be loaded into the root table S upon any cache miss, which is caused by probing an equality predicate on a cache key column of S. Furthermore, it guarantees that domain completeness for all cache key columns of S is satisfied after loading these n_S records. In fact, for the computation of n_S, only the cardinality of the ck_{eff} column matters (together with N_S, the cardinality of S_{B}). Therefore, we always deal with ck_{eff} in the following.

A computation step considers a source table S – for the initial step, the root table –, an applicable RCC R, and a target table T. Given N_S and N_T, the expected filling size n_T of T enforced by R can be derived from n_S and from the cardinalities of the columns connected by R. In a subsequent step, table T may become the new source table S', and an outgoing RCC together with its target table T' is selected as the component under consideration.

4.3 Population Estimation – A Simple Example

To derive a general scheme of cache group filling, the basic step and its quantitative description can be studied using the situation illustrated in Fig. 3. It covers the essential column and RCC types: $O.k$ labels a NU (effective) cache key, whereas $O.a$ and $O.b$ denote U and NU (non-domain-complete) columns.

In the following, a new value v of cache key $O.k$ is assumed to be filled into cache table O, enforcing the load of $n_{O.k}$ records of type O to guarantee DC of $O.k$. Because $O.a$ is unique, the number of new $O.a$ values is $n_{O.k}$. But what is the number $n_{O.b}$ of $O.b$ values that appear in the cache as a consequence of v? Furthermore, how many records of types Q, V, and P have to be loaded to satisfy the RCCs $O.a \rightarrow Q.e$, $O.a \rightarrow V.g$, and $O.b \rightarrow P.c$?

Of course, when a column is unique, its cardinality is equal to the number of records in the corresponding backend table, e. g., $c_{O.a} = N_O$. Using the uniform-value-distribution assumption, we can immediately compute $n_O = n_{O.k} = N_O/c_{O.k}$, which is the number of records of type O to be loaded. The computation of $n_{O.b}$ is much more difficult and requires additional thoughts which led to our model $f(N, c, n)$. Hence, by substituting $n_{O.b}$ for d and N_O, $c_{O.b}$, and n_O in $f(N, c, n)$, we can compute $n_{O.b} = f(N_O, c_{O.b}, n_O)$.[4]

[4] The formula for $f(N_O, c_{O.b}, n_O)$ is only one possible option. If available, any other (possibly more efficiently computable) approximation could act as a substitute.

A fundamental difficulty prohibits the more accurate approximation of the case where both $O.k$ and $O.b$ are non-unique. All the $n_{O.b}$ values computed by our formula become effective for the first $O.k$ value only. In a subsequent filling initiated by a new $O.k$ value v', some $O.b$ values qualified by v' may already reside in the root table, and, therefore, only $n_{\text{eff}} \leq n_{O.b}$ values may trigger further fillings via RCCs. A more accurate approximation would require to consider the filling history of the cache group which seems to be impractical and is definitely beyond the rationale of our estimation model. Moreover, this case does not seem so important to justify additional model complexity. Therefore, we always put up an inequality relation in formulas where $n_{O.b}$ is involved: $n_{\text{eff}} \leq n_{O.b} = f(N_O, c_{O.b}, n_O)$.

In case of a unique source column $O.a$ of an RCC (i. e., a membership constraint), always all $n_{O.a} = n_O$ values are new and lead to the loading of the corresponding values in the target column of the participating table (let's say Q or V). Hence, always n_Q (n_V) records of type Q (V) are to be filled into the cache table Q (V): $n_Q = n_O \cdot N_Q/c_{Q.e}$ ($n_V = n_O \cdot N_V/c_{V.g}$). In case of a non-unique source column $O.b$ of an RCC (i. e., an OC or XC), all $n_{O.b} \leq n_O$ values are assuredly new only when O is empty. In general, some of these values may already have been brought into the cache table by a previously referenced $O.k$ value. Therefore, the resulting cache load for the target table P can only be estimated by $n_{P1} \leq n_{O.b} \cdot N_P/c_{P.c} = f(N_O, c_{O.b}, n_O) \cdot N_P/c_{P.c}$.

As indicated in Fig. 3, P is reached by an additional RCC loading path $O.a \to V.g, V.h \to P.d$, the contribution of which has to be approximated. According to our assumptions, n_V records cause $d = f(N_V, c_{V.h}, n_V)$ different values to be loaded into $V.h$, which, in turn, need d different owner records in P. Because there are already records in P loaded via $O.b \to P.c$, we may encounter some of these owner records there. Hence, we can expect $n_P = n_{P1} + n_{P2} - n_{P1} \cdot n_{P2}/N_P$ records to be loaded into P where $n_{P2} = f(N_V, c_{V.h}, n_V) \cdot N_P/c_{P.d} = f(N_V, c_{V.h}, n_V)$.

4.4 General Scheme for a Single Evaluation Step

After having discussed, by referring to the example in Fig. 3, the various parameters influencing the filling of a cache group caused by a single cache key value, we can generalize our notation and summarize our findings. In the following, we use S and T for source and target table, or for a table in general. Compiled in Tab. 1, we have derived a general schema for determining the cache table filling of a single evaluation step i. There we assess the effects of a single RCC R of a given type U/NU $\to$ U/NU.[5] Initiated by n_S records filled into S, the listed value n_T is the expected size of the record set that is to be filled into T to satisfy RCC R. For the initial step ($i = 1$), n_S is derived from the effective cache key of the root table. Note again, a target table becomes the source table of the subsequent evaluation step $i + 1$: $T_i \mapsto S_{i+1}$ and $n_{T_i} \mapsto n_{S_{i+1}}$.

[5] We assume a lossless join along an RCC. If, for example, an RCC connects a unique column of S with a unique column of T, then $N_S = N_T$, i. e., NULL values do not occur in these columns.

Table 1. General scheme for the table population induced by an RCC

target table filling n_T		target column $T.j$ U	target column $T.j$ NU
	U	$n_S \cdot 1$	$n_S \cdot N_T/c_{T.j}$
source column $S.i$	NU, DC	$n_S \cdot c_{S.i}/N_S \cdot 1$	$n_S \cdot c_{S.i}/N_S \cdot N_T/c_{T.j}$
	NU, non-DC	$f(N_S, c_{S.i}, n_S) \cdot 1$	$f(N_S, c_{S.i}, n_S) \cdot N_T/c_{T.j}$

Whether a non-unique column $S.i$ becomes domain complete, is context dependent (see Sect. 3.1). If it is domain complete, all its values cause new record sets of size $N_S/c_{S.i}$ to be loaded into S.

5 Evaluation of Single Cache Groups

So far, we have considered the effect of a single RCC on cache group filling. Note, since n_S and n_T are context dependent, it is not sufficient to sum the individual RCC filling results. Starting from an empty cache group G, our next goal is the estimation of G's population induced by a single ck_{eff} value. This estimation is an upper bound (in the average-case sense) for subsequent fillings due to a further ck_{eff} value, because, in case of a NU, non-DC source column $S.i$ of an RCC, some of the values estimated by $f(N, c, n)$ may already reside in column S. Hence, only some of these values may lead to a filling activity in the target table. The effective set of values is usually smaller than estimated by our formula – a rare situation that, however, has to be taken into account due to limited model accuracy.

We propose a population-estimation algorithm PE that refers to G's reachability graph γ built from its applicable RCCs only; we assume that γ is cycle free. PE starts at the given filling point and computes – once and for all – the number of records n_S to be filled into the root table. Each member table T has $m \geq 1$ incoming RCCs originating from source tables $S_1, \dots, S_m$. In order to avoid multiple evaluation of T's outgoing RCCs (for each incoming RCC separately), we need the expected size $n_{T\text{act}}$ of T's population based on all incoming RCCs, before we compute the populations of tables directly reachable from T.[6] Furthermore, in order to compute T's table population at once, we must know $n_{S_i\text{act}}$ of all source tables S_i. Since this rule applies to all S_i in their role as a target table as well, we have to compute the individual table populations in cache group G in such an order that, for the estimation of each T, the estimated populations of all S_i are already known. In other words, we have to perform a topological sort TS of G's reachability graph[7] to determine the order in which

[6] Although G is assumed to be cycle-free, this is the reason why a single traversal (e. g. left-most depth-first) of G's reachability graph is not sufficient for the population estimation.

[7] Since a topological sort detects cycles, it is a consistency check whether the PE algorithm is applicable to G's reachability graph.

```
1. Using its reachability graph, list the topological sort order of $G$'s tables in TSO.
2. Visit the first (root) table $S$ in TSO and compute the expected population $n_S$
   using the cardinality of $ck_{\text{eff}}$.
3. While there are tables in TSO not visited, visit the next table $T$ and obtain the
   expected table population $n_T$:
   (a) For each incoming RCC $R_i = S_i.a \rightarrow T.b_i$, compute the population $n_{Ti}$
       using the already computed $n_{S_i\text{act}}$ of its source table $S_i$, the type of $R_i$,
       and the cardinality of its target column $c_{T.b_i}$.
   (b) $n_{T\text{act}}$ is obtained by applying the combination-of-events model using all
       determined $n_{Ti}$.
```

Fig. 4. Algorithm PE, estimating the population of a cache group G caused by a reference to a single ck_{eff} value.

the table populations can be computed. Since only the root table of G lacks incoming arcs, it is the starting point of TS.

Knowing $n_{S_i\text{act}}$ for all source tables $S.i$ connected to T via incoming RCCs, we can apply the appropriate formula of Tab. 1 and compute the population sizes n_{Ti} $(i = 1, \ldots, m)$ expected from each of the m RCCs. Since the corresponding record sets are considered stochastically independent, we can eliminate the expected duplicates from our population estimation by computing $n_{T\text{act}} = n_{T1\ldots m}$ as sketched in Sect. 4.1. Figure 4 summarizes the steps of PE.

6 Conclusion and Future Work

We have introduced constraint-based database caching using as an example a specific kind of cache groups tailored to PSJ queries, which frequently occur in TWAs. Cache groups provide predicate completeness for predicates built constructively from simple base predicates, which are specified as parameterized constraints on cache tables. This use of parameters gives cache groups a simple kind of adaptability.

The analysis of the basic type of cache groups has shown that one must be aware of the consequences of a set of specified cache constraints: On the one hand, performance problems due to uncontrolled cache loading must be prevented; on the other hand, one must know which kinds of predicates can be evaluated correctly in the cache and must have efficient probe operations to check the availability of predicate extensions. Furthermore, for each variation of constraint-based caching, quantitative analyses must help to understand which cache configurations are worth the effort. Therefore, we have developed the basic principles to quantitatively estimate the loading costs of a given cache group configuration.

Our framework can be used for the design of a cache group advisor supporting the DBA in the specification of a cache group, when the characteristics of the workload are known. Then, the expected costs for cache maintenance and the savings gained by predicate evaluation in the cache can be determined thereby identifying the trade-off point of cache operation. For example, starting with the cache tables and join paths exhibiting the highest degrees of reference locality, the cache group design can be expanded by additional RCCs and tables until the optimum point of operation is reached. Such a tool may also be useful during cache operation by observing the workload patterns and by proposing or automatically invoking changes in the cache group specification. This kind of self-administration or self-tuning opens a new and complex area of research often referred to as autonomic computing.

There are many other issues that wait to be resolved: For example, we have not said anything about the invalidation of predicates, about the removal of overlapping predicate extensions from the cache, or about different strategies how updates can be applied to cache and backend DB. We also want to explore, how the idea of constraint-based caching can be extended to other types of predicates (e. g., range or aggregation predicates).

Acknowledgments. We want to thank M. Altinel, Ch. Bornhövd, and C. Mohan for many fruitful discussions during the first author's sabbatical at ARC of IBM in San Jose. This cooperation led to deep insights into the problems of constraint-based database caching.

References

1. Akamai Technologies Inc.: Akamai EdgeSuite. http://www.akamai.com/
2. Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Berthold Reinwald: Cache Tables: Paving the Way for an Adaptive Database Cache. VLDB Conference 2003: 718–729
3. Khalil Amiri, Sanghyun Park, Renu Tewari, Sriram Padmanabhan: DBProxy: A Dynamic Data Cache for Web Applications. ICDE Conference 2003: 821–831
4. Jesse Anton, Lawrence Jacobs, Xiang Liu, Jordan Parker, Zheng Zeng, Tie Zhong: Web Caching for Database Applications with Oracle Web Cache. SIGMOD Conference 2002: 594–599
5. William Feller. *An Introduction to Probability Theory and Its Application.* 3rd edition, John Wiley & Sons, 1968
6. Jonathan Goldstein, Per-Åke Larson: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. SIGMOD Conference 2001: 331–342
7. Theo Härder, Andreas Bühmann: Datenbank-Caching – Eine systematische Analyse möglicher Verfahren, Informatik – Forschung und Entwicklung, Springer (2004)
8. Per-Åke Larson, Jonathan Goldstein, Jingren Zhou: MTCache: Mid-Tier Database Caching in SQL Server. ICDE Conference 2004
9. Stefan Podlipinig, Laszlo Böszörmenyi: A Survey of Web Cache Replacement Strategies. ACM Computing Surveys 35:4, 374–398 (2003)
10. The TimesTen Team: Mid-tier Caching: The TimesTen Approach. SIGMOD Conference 2002: 588–593
11. Gerhard Weikum: Web Caching. In: *Web & Datenbanken – Konzepte, Architekturen, Anwendungen.* Erhard Rahm/Gottfried Vossen (Hrsg.), dpunkt.verlag, 191–216 (2002)
12. S. Bing Yao: An Attribute Based Model for Database Access Cost Analysis. ACM Trans. Database Syst. 2(1): 45–67 (1977)