Database Caching: Analysis of Constraint-based Approaches Exemplified by Cache Groups

Andreas Bühmann, Theo Härder

University of Kaiserslautern, Postfach 3049, 67653 Kaiserslautern, Germany
{buehmann, haerder}@informatik.uni-kl.de

Abstract

Caching is a proven means to improve scalability and availability of software systems as well as to reduce latency of user requests. In contrast to Web caching where single Web objects are kept ready somewhere in caches in the user-to-server path, database caching uses a full-fledged DBMS as a cache to adaptively maintain sets of records from a remote DB and to evaluate queries on them. We give an introduction to the new class of constraint-based DB caching, by the example of cache groups. 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. Finally, once unsafe cache configurations, whose performance is uncontrollable, are identified, the costs of safe ones can be analyzed quantitatively.

1 Motivation

Transactional Web applications (TWAs) dramatically grow in number and complexity. At the same time, each application is expected to process increasing data volumes. 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. Many techniques have therefore emerged in recent years to keep static Web objects (like XML fragments or images) in caches in the user-to-server path.

As the TWAs must deliver more and more dynamic and frequently updated content, *Web caching* [6] should be complemented by techniques that are aware of the consistency and completeness requirements of cached data (whose source is updated in a backend DB) 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 [1, 2, 3]. Currently many DB vendors are developing prototype systems or are just extending their current products [e.g., 5, 7].

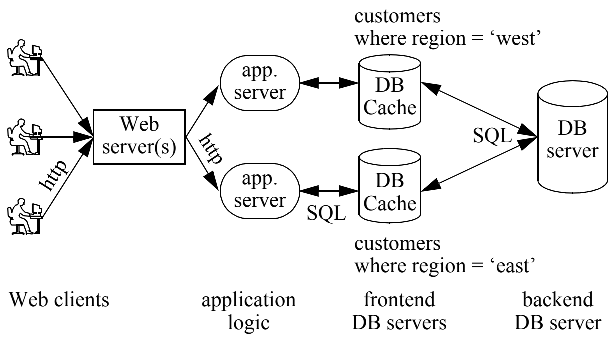

Fig. 1: Database caching for Web applications

What is the technical challenge of this approach? When responses to user requests are assembled from static and dynamic contents somewhere in a Web cache, the dynamic portion is generated by a remote application server (AS), 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 ASs to data centers closer to the users: But this displacement of ASs 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 AS in *DB caches* (Fig. 1). A flexible solution should not only support DB caching at mid-tier nodes of central enterprise infrastructures [7], but also at edge servers of content delivery networks or remote data centers.

A practical solution should also feature *cache transparency*, i.e., the application programming interface must not be modified. This 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., in order to comply with strict consistency requirements).

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

A federated query facility [1, 5] allows cooperative predicate evaluation by multiple DB servers. This is 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. 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 in the DBCache project [1] and explore the underlying ideas; this work has lead us to a class of techniques which we term *constraint-based database caching* [4].

Cache groups are collections of related cache tables; cache constraints defined on and between them determine which records of the corresponding backend tables to keep 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 *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 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 or not a complete predicate evaluation can be performed: Only a simple probe query is required in the cache to determine the availability of predicate extensions.

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.

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 condition takes the shape of *value completeness:*

Value completeness (VC). *A value v is said to be value complete in a column $S.c$ if and only if all records of $\sigma_{c = v}S_B$ 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 columns.

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 DC 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) cache table columns (defined by SQL constraints in the backend DB schema) are DC per se (*implicit* DC); non-unique (NU) columns in contrast need extra enforcement of DC.

So far, we can evaluate only equality predicates in the cache. To enhance such queries with equi-join predicates, we introduce *referential cache constraints*.

Referential cache constraint (RCC). *RCC $S.a \rightarrow T.b$ between a source column $S.a$ and 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 $S.a \rightarrow T.b$ 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: ($S.c = x$

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 a *self-RCC* $S.c \rightarrow S.c$; 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 constituting a predicate extension get into the cache? And how are these predicate extensions actually chosen? For these tasks, we introduce a second kind of cache constraint:

Cache key. *A cache key column* $S.k$ *is always kept domain complete. Only values in* $\pi_k S_B$ *initiate cache loading when they are referenced by user queries.*

You can imagine that a cache key includes a self-RCC; it can always be used as an entry point. (The columns get *explicitly* DC in both cases.) But in addition, a cache key serves as a *filling point* for a *root table* R and—via the RCCs between R and related cache tables—for the *member tables* of the cache group: Whenever a query references a particular cache key value v that is not in the cache, the query has to be evaluated by the backend DB; but the cache manager satisfies the value completeness for the missing value v by fetching all required records from the backend and loading them into the cache table R. To satisfy the RCCs, the member tables of the cache group are loaded in a similar way (for details see [1]). Hence, a reference to a cache key value v serves as something like an indicator that, in the immediate future, locality of reference is expected on the predicate extension determined by v.

Assume a cache group G with cache tables C, O, and P (customer, order, product), formed by $C.a \rightarrow O.b$ and $O.c \rightarrow P.d$, where $C.a$, and $P.d$ are U columns and $O.b$ and $O.c$ are NU columns (Fig. 2). 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 $T.c = x$. Any enhancement of this predicate with equi-join predicates is allowed if these predicates correspond to RCCs reachable from cache table T.

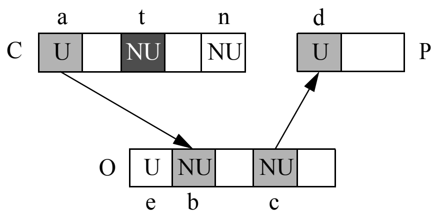

Fig. 2: Cache group G

Assume, we find 'gold' in $C.t$ (of cache group G), 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 $C.n$ like 'Smi%' and $O.e > 42$ 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 initiating 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, RCC cycles are easily constructed, which can lead to excessive population of cache groups. Such "dangerous" load behavior must clearly be prevented.

3.1 Entry points for query evaluation

We have argued that a cache table column can be tested and used correctly by an equality predicate 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; unique columns are implicitly domain complete. Cache-supported query evaluation gains much more flexibility and power, if we can correctly decide that other cache table columns are domain complete as well.

Let us refer again to G. Because $C.a \rightarrow O.b$ is the only RCC that induces filling of O, we know that $O.b$ is domain complete (*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 $O.b$. 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[1]: Assume an additional RCC ending in $O.e$ induces a new value v, which implies the insertion of $\sigma_{e = v}O_B$ 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 $\sigma_{b = w}O_B$ 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 it can be lost when a cache group configuration is modified.

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 a given context. For example, in G the 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), this extra work will influence the transaction throughput in heavy workload situations.

Specific cache group configurations may even exhibit a recursive loading behavior. 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 a domain complete. These records, in turn, populate b with a set of (new) values; all records having one of these values in b must then be loaded into X, possibly introducing new values into a. As a result, a and b may receive new values in a recursive way.

Cache groups are called *safe* if such recursive loading behavior cannot occur: Upon a cache key miss, the initiated cache loading always stops after a *single filling pass* through the cache group. Obviously, recursive loading requires 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 involving only one column per table). 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 Evaluation of Quantitative Aspects

Having identified unsafe cache configurations, whose performance is unpredictable, we must compare the safe ones in terms of cost and benefit. The resulting knowledge could lead to a design tool that proposes promising cache configurations.

A first step towards a cost model for cache groups is to answer, how many records n_T of which types T are loaded after a reference to a cache key value. Even if one makes the standard assumptions of query optimization (i.e., uniform value distribution in each column, stochastically independent of other columns), difficulties arise: The sets of records dependent on different cache key values can intersect (e.g. many customers in Fig. 2 may have ordered the same products); therefore, with an increasing number of cache key values in the cache, the number of records to be loaded for a new one decreases in such situations. Accordingly, we will gain only an upper bound for n_T if we assume an empty cache.

We denote by $c_{S.a}$ the cardinality of a column $S_B.a$ (the number of different values in a column) and by N_S the number of records in table S_B. Let us now calculate n_T for all tables T in our cache group example G; we assume all cache tables to be empty and insert a single value v into $C.t$. Value com-

[1] We must distinguish between RCCs that only *reach* a column and RCCs that also *fill* it: There are RCCs that never cause the loading of any record (e.g., a self-RCC on a U column) and thus cannot disturb induced DC. How to effectively classify an arbitrary RCC is still an open issue.

pleteness of $C.t$ requires $n_C = N_C / c_{C.t}$ records; the same number of values appears in $C.a$. Each of these values is made value complete in $O.b$, which forces $n_O = n_C(N_O / c_{O.b})$ records into O. The number $d_{O.c}$ of different values expected in $O.c$ is not as easily calculated as $d_{C.a}$, because $O.c$ is NU; the derivation requires stochastic considerations not shown here. The next step (RCC $O.c \rightarrow P.d$) is simple again: Column $P.d$ is U, so that $n_P = d_{O.c}$ records are expected in P.

Cache group G has a linear structure, which is reflected in the calculation: Each n_T is determined by at most one other n_S, and there is one that is directly known (here n_C); this also applies to all trees among possible configurations. In acyclic graphs, we can proceed in a similar way, following a topological order; to calculate n_T, we need only a way to merge the influences of the immediate predecessors of T. New ways of approximating the number of records loaded must be found in the case of an RCC cycle, where mutual influences occur.

The maintenance costs of cache groups deliver an important building block for a model determining the setup costs of a cache group which, in turn, is essential for estimating the savings of evaluating a predicate in the cache. To achieve the required precision of the loading costs, we need to develop a DB model characterizing the cardinalities of the backend tables, the selectivities of their columns, and the distribution of their values. On the other hand, a workload model essentially governs the actual cache group design, because type and frequency of given queries identify the cache keys (controlled by stop-word lists) and RCCs. Hence, sufficiently accurate models for workload, cache group, and DB are vital for a quantitative justification of a cache group in a TWA environment. To validate these results, various kinds of measurements are needed in a real DB cache setting.

5 Conclusions

We have introduced constraint-based database caching using as an example the specific kind of cache groups proposed in the DBCache project. 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. In the future we want to explore, how the idea of constraint-based caching can be extended to other types of predicates (e.g., range or aggregation predicates). Perhaps it is also possible to let evolve cache group specifications themselves (e.g., by adding or dropping RCCs, when changed join patterns are observed in the workload), thus reaching a higher level 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.

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 predicates extensions from the cache, or about different strategies how updates can be applied to cache and backend.

References

[1] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald: Cache Tables: Paving the Way for an Adaptive Database Cache. VLDB Conference 2003: 718–729

[2] K. Amiri, S. Park, R. Tewari, S. Padmanabhan: DBProxy: A Dynamic Data Cache for Web Applications. ICDE Conference 2003: 821–831

[3] R.G. Bello, K. Dias, A. Downing, J.J. Feenan Jr., J.L. Finnerty, W.D. Norcott, H. Sun, A. Witkowski, M. Ziauddin: Materialized Views in Oracle. VLDB Conference 1998: 659–664

[4] T. Härder, A. Bühmann: Datenbank-Caching: Eine systematische Analyse möglicher Verfahren, Informatik – Forschung und Entwicklung, Springer (2004)

[5] P.-Å. Larson, J. Goldstein, J. Zhou: MTCache: Mid-Tier Database Caching in SQL Server. ICDE Conference 2004

[6] S. Podlipinig, L. Böszörmenyi: A Survey of Web Cache Replacement Strategies. ACM Computing Surveys 35:4, 374–398 (2003)

[7] The TimesTen Team: Mid-tier Caching: The TimesTen Approach. SIGMOD Conference 2002: 588–593