Query Processing in Constraint-Based Database Caches

Theo Härder Andreas Bühmann
Department of Computer Science, University of Kaiserslautern, Germany
{haerder, buehmann}@informatik.uni-kl.de

Abstract

Database caching uses full-fledged DBMSs as caches to adaptively maintain sets of records from a remote DB and to evaluate queries on them, whereas Web caching keeps single Web objects ready somewhere in caches in the user-to-server path. Using DB caching, we are able to perform declarative and set-oriented query processing near the application, while data storage and consistency maintenance is remote. We explore which query types can be supported by DBMS-controlled caches whose contents are constructed using parameterized cache constraints. Schemes on single cache tables or on cache groups correctly perform local evaluation of query predicates. In practical applications, only safe schemes guaranteeing recursion-free load operations are acceptable. Finally, we comment on future application scenarios and research problems including empirical performance evaluation of DB caching schemes.

1 Introduction

Database caching tries to accelerate query processing by using full-fledged DBMSs to cache data in wide-area networks close to the applications. The original data repository is a backend database (BE-DB), which maintains the transaction-consistent DB state, and up to n frontend databases (FE-DBs) may participate in this kind of "distributed" data processing. For example, server-selection algorithms enable (Web) clients to determine one of the replicated servers that is "close" to them, which minimizes the response time of the (Web) service. This optimization is amplified if the invoked server can provide the requested data—frequently indicating geographical contexts. If a query predicate can be answered by the cache contents, it is evaluated locally and the query result is returned to the user. When only the answer to a partial predicate is locally available, the remaining part of the predicate is sent to the BE-DB. Obviously, this kind of federated query processing, where the final query result is put together by the FE-DB, can be performed in parallel and can save substantial communication cost.

An FE-DB supports declarative and set-oriented query processing (e. g., specified by SQL) and, therefore, keeps sets of related records in its DB cache which must satisfy some kind of *completeness condition* w. r. t. the predicate evaluated to ensure that the query execution semantics is equivalent to the one provided by the BE-DB. Updates to the database are handled—in the simplest scenario—by the BE-DB which propagates them to the affected FE-DBs after the related transaction commit. We assume that an FE-DB modifies its state of the cache within a time interval δ after the update, but do not discuss all the intricacies of DB cache maintenance.

So far, all approaches to DB caching were primarily based on materialized views and their variants [2, 4]. A materialized view consists of a single table whose columns correspond to the set of output attributes $O_V = \{O_1, \ldots, O_n\}$ and whose contents are the query result V of the related view-defining query Q_V with predicate P.

Copyright 2004 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Materialized views can be loaded into the DB cache in advance or can be made available on demand, for example, when a given query is processed the nth time ($n \geq 1$), exhibiting some kind of built-in locality and adaptability mechanism. When they are used for DB caching, essentially independent tables, each representing a query result V_i of Q_{V_i}, are separately cached in the FE-DB. In general, query processing for an actual Q_A is limited to a single cache table. The result of Q_A is contained in V_i, if P_A is logically implied by P_i (subsumption) and if O_A is contained in O_{V_i}. Only in special cases, a union of cached query results, e. g., $V_1 \cup V_2 \cup \ldots \cup V_n$, can be exploited. DBProxy [2] has proposed some optimizations at the storage level. To reduce the number of cache tables, DBProxy tries to store query results V_i with strongly overlapping output attributes in common tables.

2 Concept of constraint-based database caching (CBDC)

CBDC promises a new quality for the placement of data close to their application. The key idea is to accomplish for some given types of query predicates P the so-called *predicate completeness* in the cache such that all queries eligible for P can be evaluated correctly. All records (of various types) in the BE-DB which are needed to evaluate predicate P are called the *predicate extension* of P. Because predicates form an intrinsic part of a data model, the various kinds of eligible predicate extensions are data-model dependent, that is, they always support only specific operations of a data model under consideration.

Suitable cache constraints for these predicates have to be specified for the cache. They enable cache loading in a constructive way and guarantee, when satisfied, the presence of their predicate extensions in the cache. The technique does not rely on the specification of static predicates: The constraints are parameterized making this specification adaptive; it is completed when the parameters are instantiated by specific values. An “instantiated constraint” then corresponds to a predicate and, when the constraint is satisfied—i. e., all related records have been loaded—it delivers correct answers to eligible queries. Note, the union of all existing predicate extensions flexibly allows the evaluation of their predicates, i. e., $P_1 \cup P_2 \cup \ldots \cup P_n$ or $P_1 \cap P_2 \cap \ldots \cap P_n$ or subsets/combinations thereof, in the cache.

There are no or only simple decidability problems whether predicates can be evaluated. Only a simple probe query is required at run time to determine the availability of eligible predicate extensions. Furthermore, because all columns of the corresponding BE tables are kept, all *project operations* possible in the BE-DB can also be performed in the cache. Other operations like *selection and join* depend on specific *cache constraints*. Since full DB functionality is available, the results of these queries can further be *refined* by selection predicates such as Like, Null, etc. as well as processing options like Group-by, Having (potentially restricted), or Order-by.

A cache contains a collection of cache tables which can be isolated or related to each other in some way. For simplicity, the names of tables and columns are identical in the cache and in the BE-DB. Considering a cache table S, we denote by S_{B} its corresponding BE table, by $S.c$ a column c of S. Note, a cache usually contains only subsets of records pertaining to a small fraction of BE tables. Its primary task is to support local processing of queries that typically contain simple projection (P) and selection (S) operations or such having up to 3 or 4 joins (J) [1]. 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 BE-DB consists of hundreds of tables.

3 Supporting PS queries

Let us begin with single cache 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 table S_B carrying 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 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) columns of a cache table (defined by SQL constraints "unique" or "primary key" (PK) in the BE-DB schema) are DC per se (*implicit* domain completeness). Non-unique (NU) columns in contrast need extra enforcement of DC.

3.1 Equality predicates

If a DC column is referenced by a predicate $S.c = v$ and v is found in $S.c$, then table S is eligible for a PS query containing this equality predicate. To explicitly specify a column to be domain complete, we introduce a first cache constraint called *cache key column*, cache key for short[1], which can always be used as an entry point for the evaluation of equality predicates. But in addition, a cache key serves as a *filling point* for a table S. In contrast to [1], we define cache keys in a more subtle way. Because low-selectivity values even in a high-selectivity column defined as cache key column can cause filling actions involving huge sets of records never used later, filling control by a recommendation list R or stop-word list L is mandatory. Whenever a query references a particular cache key value v that is not in the cache, query evaluation of this predicate has to be performed by the BE-DB. However, as a consequence of this cache miss attributed to a cache key reference, the cache manager satisfies value completeness for v—if it is contained in R (or not contained in L)—by fetching all required records from the backend and loading them into the table S (thus keeping the cache key column domain complete[2]).

Definition 3 (Cache key column): A cache key column $S.k$ is always kept domain complete. Only values in $R \subseteq \pi_k S_B$ initiate cache loading when they are referenced by user queries.

Hence, a reference to a cache key value x—or, more general, to a cache constraint—in a query serves as something like an indicator that, in the immediate future, locality of reference is expected on the predicate extension determined by x. Cache key values therefore carry information about the future workload and sensitively influence caching performance. Therefore, usage statistics and history information collected for such references should be employed to select cache keys and to build recommendation lists.

3.2 Range predicates

To answer range queries on a single column, we need a new type of cache constraint called cache value range which makes all values of a specified range value complete. Although possible for any domain with ordered values, here we restrict our considerations to domains of type integer or string. Hence, a value range carries two parameters l and u ($-\infty \leq l \leq u \leq +\infty$).

A value range defined with closed boundaries serves as another type of cache constraint. In a query, a range predicate can take various forms using the relationships $\Theta \in \{<, =, >, \leq, \neq, \geq\}$. An actual range predicate r_A can be easily mapped to a cache value range, e. g., $x > l$ to $l \leq x < +\infty$. When loading range predicate extensions, value ranges r_i already existing in the cache have to be taken into account. In the simplest case, the cache could be populated by all records belonging to the (partial) range which leads to a cache miss when a range query with $S.c = r = (l \leq x \leq u)$ is evaluated. Cache loading makes table S range complete for $S.c = r$ such that subsequent queries with range predicates contained in r can be correctly answered in the cache.

[1] We assume single-column cache keys. An extension of our statements to multi-column cache keys, however, is straightforward.

[2] VC for each cache key value keeps probing simple; cache update operations may require a revised definition and complex probing.

Definition 4 (Value range completeness, RC): A value range $r = (l \leq x \leq u)$ is called range complete in a column $S.c$ if and only if all records of $\sigma_{c \geq l \wedge c \leq u} S_{\mathrm{B}}$ are in S (making all individual values in $S.c$ value complete).

To be aware of the value ranges r_i present in the cache, the cache manager keeps an ordered list of the r_i. As soon as two adjacent r_i merge, they are melted to a single range. Hence, queries with range predicates r_A contained in an r_i present ($r_A \subseteq r_i$) can be locally evaluated, whereas overlapping r_A cause the cache to be populated by records making the missing values RC. Note, the selectivity and potential locality of key ranges have to be strictly controlled to prevent "performance surprises". This is especially true for open ranges (l or u is ∞) or $\Theta = \neq$. Again, cache filling should be refined by a recommendation list R (or stop-word list L).

Definition 5 (Cache value range, CVR): A column $S.k$ is domain complete and each value range r_i is always kept range complete. Only values of r_i in $R \subseteq \pi_k S_{\mathrm{B}}$ initiate cache loading when they are referenced by queries.

3.3 Other types of predicates

SQL allows some more types of predicates on single tables. However, although possible, it is not reasonable to strive for keeping their predicate extensions in the cache. For predicates which need large portions of or even the entire table for their evaluation, it is more reasonable to process the related query in the BE-DB and to provide a materialized view in the cache. This is usually true for all aggregate queries (MAX, MIN, SUM, user-defined aggregate functions, etc.) or queries containing *like* predicates. Of course, depending on the specific parameters and references, predicates such as Exists, All, etc. or those requiring subqueries are "bad" for the use of CBDC, because their predicate extensions may be too large and may not exhibit enough reference locality.

However, if queries contain a constraint-determining (part of a) predicate such as an equality or range predicate P, these "bad" types of predicates, when limited to predicate extensions of P, can be applied in the cache thereby allowing perfectly local query processing.

4 Support of PSJ queries

A powerful extension of cache use is to enable equi-joins to be processed in the cache and to combine them with PS predicates, thereby achieving PSJ queries. For this purpose, we introduce *referential cache constraints* (RCCs), which guarantee the correctness of equi-joins between cache tables. Such RCCs are specified between two columns of cache tables S and T, which need not be different, not even the columns themselves.

Definition 6 (Referential cache constraint, RCC): An 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 $T.b$.

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: The 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. In a similar way, CVRs can be combined with RCCs to construct predicate extensions (for range and join predicates) to be locally evaluated. For the implementation of this idea, we refer to a particular approach called *cache groups*.

Definition 7 (Cache group): It is a collection of cache tables linked by a number of RCCs. A distinguished cache table is called the root table R of the cache group and holds at least one cache key or cache value range. The remaining cache tables are called member tables M_i and must be reachable from R via the (paths of) RCCs.

Figure 1: Cache groups COL and COP for order processing

Cache constraints—cache keys as filling points defined on R and RCCs specified between R and M_i—determine the records of the corresponding BE tables that have to be kept in the cache. Depending on the types of the source and target columns on which an RCC is defined, we classify the RCCs as U $\rightarrow$ U or U $\rightarrow$ NU (member constraint, MC), NU $\rightarrow$ U (owner constraint, OC), and NU $\rightarrow$ NU (cross constraint, XC).

4.1 TimesTen cache groups

The TimesTen (TT) team originally proposed the notion of a cache group [5] that consists of a root table and a number of member tables connected via member constraints (corresponding to PK/FK relationships in the BE-DB). A TT cache instance (CI) is a collection of related records that are uniquely identifiable via a CI key. The root table carries a single identifier column (U) whose values represent CI keys. Because all records of CIs must be uniquely identifiable, they form non-overlapping tree structures (or simple disjoint DAGs) where the records embody the nodes and the edges are represented by PK/FK value-based relationships.

Note, there is no equivalence to our notion of cache keys (possibly NU), because cache loading is not based on reference to specific values (parameterized loading). In contrast, it is controlled by the application (which gives something like prefetching directives or hints) where various options are supported for the loading of CIs ("all at once", "by id", "by where clause"). There is no notion of domain completeness, of cache-controlled correctness, or of the completeness of predicate extensions. Figure 1 illustrates two cache groups with tables C,O,L and P where $C.a$, $O.d$, and $P.e$ are U columns and $O.b$, $O.c$, and $L.e$ are NU columns. In a common real-world situation, C,O,L and P could correspond to BE-DB tables Customer, Order, OrderLine and Product. COL is a TT cache group and is formed by $C.a \rightarrow O.b$ and $O.d \rightarrow P.e$ where both RCCs would typically characterize PK/FK relationships used for join processing. For example, if Customer $(C.a = 4711)$ as CI key is in the cache, its CI represents a tree structure used to locate all of his Orderline records as leaves.

4.2 Cache groups

Cache groups fully adhere to our definitions of cache keys and RCCs. They are introduced by the DBCache project [1] and extend the TT cache groups by enabling the use of RCCs of types OC and XC—in addition to MC—and by explicit specification of cache keys which make them parameterizable and (weakly) adaptive. Despite similarities, MCs and OCs are not identical to the PK/FK relationships in the BE tables: Those can be used for join processing symmetrically, RCCs only in the specified direction. XCs have no counterparts at all in the BE-DB. 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.

Query processing power and flexibility of cache groups are enhanced by the fact that specific columns are made implicitly domain complete via our RCC mechanism (for details, see [3]).

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

If a probing operation on some explicitly specified or implicitly enforced 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 table T.

Cache group COP in Figure 1 is formed by $C.a \rightarrow O.b$ and $O.c \rightarrow P.e$, and carries $C.t$ as a cache key. In COP, CIs form DAGs, and a single cache key value may populate COP with a set of such CIs. If we find 'gold' in $C.t$, then the predicate ($C.t =$ 'gold' and $C.a = O.b$ and $O.c = P.e$) 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. Additional RCCs, for example, $C.t \rightarrow O.b$ or $O.c \rightarrow C.n$ are conceivable (but only useful, if their values are join compatible); such RCCs, however, have no counterparts in the BE-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. In both cache groups, $O.b$ has IDC and $O.d$ is domain complete by definition. Hence, they can be used as entry points for equality predicates. If, for example, $O.d = x$ is found in the cache, predicate ($O.d = x$ and $O.c = P.e$) can be correctly evaluated. Of course, a correct predicate can be refined by "and-ing" additional selection terms (referring to cache tables) to it; e. g., ($C.t =$ 'gold' and $C.n$ like 'Smi%' and $O.e > 42$ and $\ldots$).

4.3 Enhancement for outer joins

Join processing in cache groups can be enhanced to support outer join operations. An outer join along an RCC R can be evaluated correctly, if anchored at the starting column of R (via some predicate P)—just like ordinary equi-joins. Assume a left outer join to be processed in cache group COP: (C Left Join O Where $C.t$ = 'gold'). RCC $C.a \rightarrow O.b$ guarantees that all (equi-)join partners of C tuples (in the cache) are in O; for the left outer join, those C tuples that do not have join partners receive artificial ones in the cache just like in the BE-DB. In a similar way, full or right outer joins can be achieved if they are supported by appropriate cache constraints.

To summarize our discussion so far, it has revealed that RCCs alone do not allow us to correctly perform joins in the cache. But using an equality or value range predicate on a DC column $S.c$ as an "anchor", we can restrict joins to records that exist in the cache: Hence, an 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 for which predicate completeness can be assured.

Definition 9 (Predicate completeness, PC): A collection of tables is said to be predicate complete with respect to predicate P if it contains all records needed to evaluate P, that is, its predicate extension.

5 Other types of queries

So far, we have outlined the most important cache constraints and their resulting predicate extensions to enable correct query evaluation in the cache. To indicate that CBDC can be streched even further, we briefly remark that queries exhibiting other predicate types—less important from a practical view—can be locally processed, too.

5.1 Processing of set operations

If record sets S and T are union compatible—having the same number of attributes mapped pairwise to the same domains—and can be derived from predicate extensions in the cache, then the usual relational set operations can be applied to them, that is, Union ($S \cup T$), Intersection ($S \cap T$), and Difference ($S \setminus T$). Typically, S and T themselves are the results of PS or PSJ queries supported by cache constraints introduced so far.

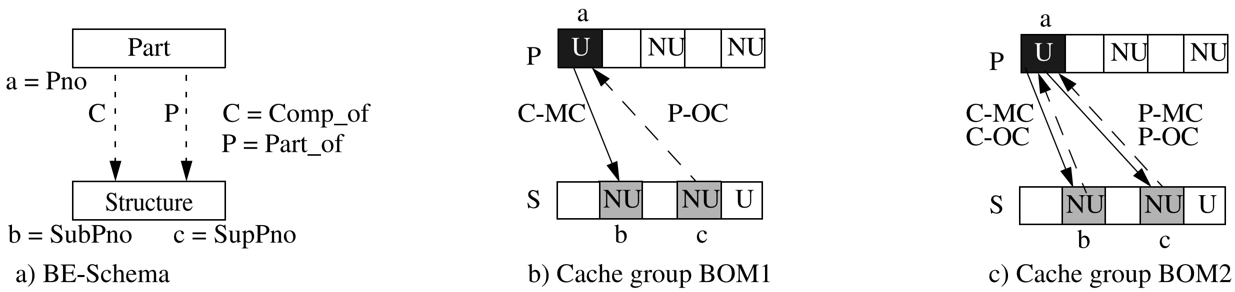

Figure 2: Cache groups for bill-of-material processing (BOM)

5.2 Evaluation of recursive queries

Cache constraints allow us to specify that the cache be recursively populated. Hence, the corresponding predicate extensions embody recursively applied relationships. Figure 2 illustrates some simple examples. Figure 2a represents the relational BE schema of a bill-of-material database. Pno is the primary key (PK) of table Part and SubPno and SupPno are foreign keys (FK) of table Structure where both PK/FK pairs form the relationships Comp_of and Part_of. These relationships are value based and symmetric. In contrast, the corresponding RCCs, although also value based, are directed. Figure 2b–c define two cache groups exhibiting simple and double recursion when the cache groups are populated (via reference of a cache key value of *P.a*). As an instructive exercise, the reader may provide a BOM example for the BE schema in Figure 2a and show the CI of BOM1 for top-level part $P.a = 1$. Then, he may provoke loading of BOM2 for any, even elementary, part $P.a = x$. To be brief, as soon as the predicate extension for $P.a = 1$ is loaded, BOM1 may be used to correctly deliver all component and elementary parts of product $P.a = 1$ (using an SQL query 'With Recursive ... Union All ...').

6 Safeness issues and cache modification

In practical applications, it is mandatory to prevent uncontrollable and excessive cache population as a consequence of recursive dependencies. Although loading can be performed asynchronously to the transaction observing the cache miss and, therefore, a burden on its own response time can be avoided, uncontrolled loading is undesirable for the following reason: It may influence the transaction throughput in heavy workload situations, because substantial extra work to be hardly estimated may be required by the FE-DB and BE-DB servers.

For this reason, only *safe* cache groups exhibiting recursive-free loading may be acceptable. Therefore, some restrictions should be applied to cache group design [3]. The most important one is related to NU-DC columns. When two or more NU-DC columns of a cache table must be maintained, then these columns may receive new values in a recursive way. Hence, in the same cache table, two or more NU columns that are DC are not allowed. Another restriction applies to heterogeneous RCC cycles in cache groups where in some table two columns are involved in the cycle.

In this document, we have excluded all aspects of cache maintenance. How difficult is it to cope with the units of loading and unloading? Let us call such a unit cache instance (CI). Depending on their complexity, CIs may exhibit good, bad, or even ugly maintenance properties. The good CIs are disjoint from each other and the RCC relationships between the contained records form trees (Figure 1 left). The bad CIs form DAGs and weakly overlap with each other (Figure 1 right). Hence when loading a new CI, one must beware of duplicates. Accordingly, shared records must be removed only together with their last sharing CI. To maintain cache groups with strongly overlapping CIs can be characterized as ugly, because duplicate recognition and management of shared records may dominate the work of the cache manager. Again, unsafe cache groups need not be considered at all. In general, they may feature unmanageable maintenance due to their recursive population behavior.

Other interesting research problems occur if we apply different update models to DB caching. Instead of processing all (transactional) updates in the BE-DB first, they could be performed in the FE-DB (under ACID protection) or even jointly in FE- and BE-DB under a 2PC protocol. Such update models may lead to futuristic considerations where the conventional hierarchic arrangement of FE- and BE-DB is dissolved: If each of them can play both roles and if together they can provide consistency for DB data, more effective DB support may be gained for new applications such as grid or P2P computing.

Furthermore, all caching schemes discussed need careful exploration of their performance behavior. Simulation with analytical models and data using column selectivities and artificial workloads only can provide a rough quantitative overview on costs and benefits. Hence, although very laborious, it must be complemented by empirical performance measurements, at least in a selective way. Interesting questions are related to update frequencies and types, that is, to the performance window in which CBDC is superior to replication or full-table caching. Moreover, how does a mix of different and overlapping predicate extensions perform? When each a single predicate type is mapped to a separate cache table or group, extension overlap would cause replicated data and, in turn, worse modification problems. In addition, for the same BE table, several cache tables would have to be maintained—a design decision which jeopardizes the highly desirable cache transparency for the applications. If at most one cache table is allocated for each BE table, overlapping cache groups, so-called cache group federations [3], necessarily interfere with each other, that is, a cache key may drive cache table filling via RCCs of other cache groups. Of course, some benefit—primarily storage saving—is gained when the same records appear in more than one cache group. However, the penalty may quickly outweigh typically small gains! Many overlapping cache constraints and, in turn, predicate extensions may question the entire approach and call for simple, but communication- and storage-intensive solutions such as full-table caching.

7 Conclusions

We have surveyed the use of CBDC for a variety of query types and primarily have structured this research area top-down. We believe that our definitions are fundamental for describing the related research problems as well as approaching viable solutions. In this respect, the terms value completeness, value range completeness, domain completeness, and predicate completeness are our *magic words*. Because we have explored the underlying concepts of CBDC at the type level, value range completeness and domain completeness to easily control it were most important. However, by controlling value completeness dynamically, CBDC can be improved even further. On the other hand, a combined solution of tasks performed by database caching and such of Web caching may offer additional optimization potential. Moreover, improvement of DB cache adaptivity seems to be a very important issue to make caches (nearly) free of administrative interventions. This aspect of *autonomic computing* is underlined by the fact that today Akamai's content distribution network already has nearly 15 000 edge caching servers [1]. Hence, we are only at the beginning of a promising research area concerning constraint-based and adaptive DB caching where a number of important issues remains to be solved or explored.

References

[1] M. Altinel, Ch. 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] T. Härder, A. Bühmann: Value Complete, Domain Complete, Predicate Complete—Magic Words Driving the Design of Cache Groups, submitted (2004)

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

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