The VLDB Journal manuscript No.
(will be inserted by the editor)

Theo Härder · Andreas Bühmann

Value Complete, Column Complete, Predicate Complete

Magic Words Driving the Design of Cache Groups

the date of receipt and acceptance should be inserted later

Abstract Caching is a proven remedy to enhance 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 accessed and kept ready somewhere in caches in the user-to-server path, database caching uses full-fledged database management systems as caches, close to application servers at the edge of the Web, to adaptively maintain sets of records from a remote database and to evaluate queries on them.

We analyze a new class of approaches to database caching where the extensions of query predicates that are to be evaluated are constructed by constraints in the cache. Starting from the key concept of value completeness, we explore the application of cache constraints and their implications on query evaluation correctness and on controllable cache loading called cache safeness. Furthermore, we identify simple rules for the design of cache groups and their optimization before discussing the use of single cache groups and cache group federations. Finally, we argue that predicate completeness can be used to develop new variants of constraint-based database caching.

Keywords database caching · query processing · cache constraints · predicate completeness

1 Motivation

Transactional Web applications (TWAs) in a variety of domains, often called "e*-applications", grow dramatically in number and complexity. At the same time, each individual Web-based application must cope with increasing demands regarding data volumes and workloads to be processed. In general, caching is a proven concept in such situations to improve response time and scalability of the applications as well as to minimize communication delays in wide-area networks. For these reasons, a broad spectrum of techniques has emerged in recent years to keep static Web objects (HTML pages, XML fragments, images, etc.) in caches in the user-to-server path (client-side proxies, forward and reverse proxies, content-delivery nodes).

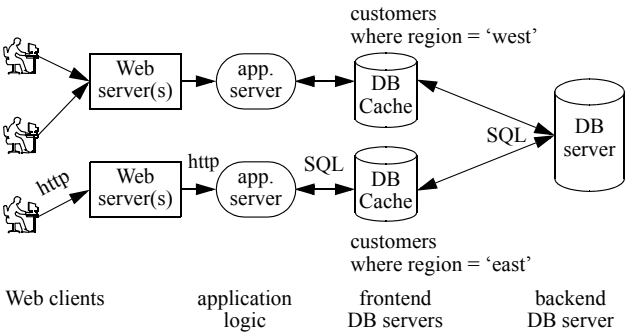


Fig. 1 Database caching for Web applications

As more and more dynamic content must be generated and as frequently updated information must be processed by the TWAs, this *Web caching* [19] 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, adapt to changing workloads. Attempts to reach these objectives are called *database caching*, for which a number of different solutions have been proposed in recent years [2; 3; 6]. Currently all leading database vendors are developing prototype systems or are just extending their current products in a suitable way [2; 5; 11; 15; 20].[1]

What is the technical challenge of all these approaches? 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, which, in turn, asks the *backend* database (DB) server for up-to-date information, thereby causing substantial latency.

Theo Härder · Andreas Bühmann
University of Kaiserslautern, P. O. Box 3049, 67653 Kaiserslautern, Germany
E-mail: {haerder, buehmann}@informatik.uni-kl.de

[1] "The three most important parts of any Internet application are caching, caching, and, of course, caching . . . "—Larry Ellison, Oracle Chairman & CEO

An obvious reaction to this performance problem is to migrate the application servers to data centers closer to the users in order to provide "nearby" services. As Fig. 1 illustrates, server-selection algorithms enable Web clients to determine one of the replicated servers close to them, which minimizes the response time of the Web service. This optimization is amplified if the invoked application servers can provide the expected data—frequently influenced by geographical contexts.

However, the displacement of application servers to the edge of the Web alone is not sufficient; in contrast, 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, prevalently used data should be kept close to the application servers in *database caches* (also called *frontend* DB servers). A flexible solution should not only support DB caching at mid-tier nodes of enterprise infrastructures [5; 20] but also at edge servers of content delivery networks [1] and at remote data centers.

The rest of the paper is organized as follows: In Sect. 2 we start with desired properties of DB caches and a sketch of the design space. Section 3 presents the idea of constraint-based DB caching and introduces parameterized cache constraints to specify *cache groups*. In Sect. 4, we discuss *safeness* conditions for cache groups, which prevent excessive loading operations in the cache. Using the results derived so far, we can subsequently give a set of rules for cache group design and optimization. Section 6 illustrates practical examples for cache groups, before we take a look at the situation where several distinct cache groups overlap in some tables thereby forming a *cache group federation*. Section 7 describes the related work on cache groups and outlines the novel aspects of this paper. Finally in Sect. 8, we summarize our results and give further conclusions for new classes of constraint-based DB caching.

2 Database caching

2.1 Requirements

The ultimate goal of DB caching is to process frequently requested DB operations close to the application. Therefore, the complexity of these operations and, in turn, of the underlying data model essentially determines the required mechanisms. The use of SQL implies a particular challenge because of its declarative and set-oriented nature: For the DB cache to be useful, the cache manager (a system component) must guarantee that queries can be processed there; that is, the sets of records (of various types) satisfying the corresponding predicates must be completely in the cache.

Using a *full-fledged DB server* as cache manager offers great advantages. A substantial portion of the query processing logic (parsing, optimization, and execution) must be made available anyway. By providing the full functionality, one can exploit additional DB objects such as triggers, constraints, stored procedures, or access paths in the cache: This simulates DB semantics locally and enhances application performance due to increased locality. Furthermore, transactional updates seem to be conceivable in the cache someday and, as a consequence, continued service for TWAs when backend DBs become unavailable.

A federated query facility as offered in [12; 14] allows cooperative predicate evaluation by multiple DB servers. For cache use, this property is very important, because the 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 use the term *predicate* to refer only to the portions of predicates that are to be evaluated in the cache.

Another important property of practical solutions is full *cache transparency* for applications; that is, we do not tolerate modifications of the application programming interface. This application transparency, which also is a prime aspect to distinguish caching from replication, is a key requirement of DB caching. At run time it gives the cache manager the choice to process a query locally or to send it to the backend DB, to comply with strict consistency requirements, for instance.

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. 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, whose queries typically contain up to three or four joins [2]. Often the number of cache tables, which feature a high degree of reference locality, is only in the order of ten or less, even if the backend DB consists of hundreds of tables.

2.2 Design space

The conceptually most simple approach—namely, full-table caching, which replicates entire contents of selected backend tables—attracted various DB cache products [18]. It seems infeasible, however, for large tables even under moderate update dynamics, because replication and maintenance costs may outweigh the potential savings on query processing.

Traditional approaches to caching at a finer granularity are settled at the object level and, hence, only support access to objects by identifiers. When receiving a declarative query, the cache generally has no means to decide whether a complete answer can be provided without querying the backend DB. Semantic descriptions of the cached data, however, enable the cache manager to determine the completeness of query results.

So far, most approaches to DB caching were primarily based on the use of single tables, sometimes called semantic caching [8; 13], or on materialized views and their variants [3; 6; 9; 15; 16]. A materialized view consists of a single table whose contents are the query result V of the view-defining query Q_V (with predicate P_V) and whose columns correspond to the set $O_V = \{O_1, \ldots, O_n\}$ of Q_V's output attributes. Materialized views can be loaded into the DB

cache in advance or can be made available on demand, for example, once a given query has been processed the nth time ($n \geq 1$). In this way, one can achieve some kind of built-in locality and adaptivity (together with a replacement scheme).

When materialized views V_i are used for DB caching, they represent results of queries Q_{V_i} executed in the backend DB and are typically cached as separate tables in the frontend DB. In general, query processing for an actual query Q_A is limited to such a single cache table. The result of Q_A is contained in V_i, if P_A is logically implied by P_{V_i} (subsumption) and if O_A is contained in O_{V_i} (i. e., the output of the new query is restricted to the attributes of a query result that is used). Only in special cases a union $V_1 \cup V_2 \cup \cdots \cup V_n$ of cached query results can be exploited.

The term active caching is used in [17] and indicates there that the proxy (cache) is actively functioning in a limited query processing role. The proxy stores the results of queries (views) and uses them to answer subsequent queries. Although user queries submitted via forms and containing joins may be handled by the proxy, this does not mean that it executes joins. Rather it treats all queries from a given form as selection queries on a single table view (with a keyword predicate).[2]

DBProxy [3] has proposed some optimizations: In order to reduce the number of cache tables, it tries to store query results V_i with strongly overlapping output attributes or schemas in common tables. Storing a superset of the attributes O_{V_i} in the cache may, on the one hand, enhance caching benefits of V_i, but, on the other hand, it may increase storage and maintenance costs. As a consequence, some flexibility is gained for single-table queries and some special multi-table queries. However, these provisions do not enable general multi-table queries.

The expressiveness of more powerful caching techniques is not confined to deriving query results (similar to project-select (PS) queries) from single cache tables via subsumption. Instead, project-select-join (PSJ) queries across multiple cache tables are desirable [2; 20]. For this purpose, an appropriate specification mechanism is needed to achieve predicate completeness in the cache for the indented class of queries. Loading directions for the cache take care that predicate completeness is dynamically guaranteed for queries anticipated frequently in the future to ensure for locality of reference in the cache. Because cache filling is selective and dynamic, an efficient probing mechanism is needed to determine whether the predicate extension is in the cache for a specific predicate to be evaluated. We proceed along these lines and propose a method based on *parameterized cache constraints* which can be dynamically adjusted to varying workload characteristics. Such steps are urgently required to comply with challenging demands like self-administration and adaptivity.[3]

3 Constraint-based database caching

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

A cache contains a collection of cache tables, which represent backend tables and which can either be isolated or related to each other in some way. All records (of various types) in the backend DB that 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 predicate extensions depend on the data model; that is, they always support only specific operations of a data model under consideration. Cache constraints enable cache loading in a constructive way and guarantee the presence of their respective predicate extensions in the cache.

This technique does not rely on static predicates: Parameterized constraints make the specification adaptive; it is completed when specific values instantiate the parameters: An "instantiated constraint" then corresponds to a predicate and, once the constraint is satisfied (i. e., all related records have been loaded), it delivers correct answers to eligible queries. Note, the set of all present predicate extensions flexibly allows combined evaluation of their predicates in the cache (e. g., $P_1 \cup P_2 \cup \cdots \cup P_n$ or $P_1 \cap P_2 \cap \cdots \cap P_n$ or subsets/combinations thereof).

Given suitable cache constraints, there are no or only simple difficulties in deciding whether certain predicates can be evaluated. At run time, only a simple existence query is required to determine whether suitable predicate extensions are available. Furthermore, because all columns of the corresponding backend tables are kept, all *project* operations possible in the backend DB can also be performed in the cache. Other operations like *selection* and *join* depend on specific cache constraints.

The primary task of this constraint-based caching approach is to support local processing of queries that typically contain simple projection and selection operations as well as equi-joins (PSJ). Since full DB functionality is available, the results of these queries can further be evaluated by aggregation functions such as *sum* or refined by selection predicates such as *like* or *is null*, as well as by processing options like *distinct*, *order by*, *group by* or *having* (restricted to predicates evaluable on the predicate extension).

[2] A number of rules listed in [17] describes how to answer more restrictive eligible queries from less restrictive ones by selection or intersection.

[3] Minimum interaction by the database administrator (DBA) is desirable when a large number of caches exists. For example, Akamai's network has nearly 15,000 edge caching servers [1].

3.1 Completeness

If we want 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.

Definition 1 (Predicate completeness) A collection of tables is said to be *predicate complete* with respect to a predicate Q if it contains all records from the backend DB needed to evaluate Q, that is, its predicate extension.

In our present model, we deal with whole records only and do not consider restrictions to certain sets of columns—however, the model could be easily extended, as DBProxy [3] does, for instance. Note that a predicate extension in the sense used here consists of all records from the backend tables needed to reconstruct the query result. For an aggregate query, the predicate extension would not be the aggregate (as the query result) but all records that are to be aggregated.

3.1.1 Equality predicates

Let us begin with single cache tables. For simplicity, let the names of tables and columns be the same 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.

For simple equality predicates like $S.c = v$, the completeness condition takes the shape of *value completeness*.

Definition 2 (Value completeness) A value v is said to be value complete (or *complete* for short) in a column $S.c$ if and only if all records of $\sigma_{c=v} S_\mathrm{B}$ are in S.[4]

If we know that a value v is value complete in a column $S.c$, we can correctly evaluate $S.c = v$, because all records from table S_B that carry this value are in the cache. But how do we know that v is value complete? We are going to answer this question in Sect. 3.2; let us assume for the moment that we are somehow able to identify complete values in the cache.

3.1.2 Range predicates

To answer range queries on a single column, we need a more general type of completeness condition that assures us that all values of a specified *interval* are value complete. Naturally, we restrict our considerations to domains with ordered values (e. g., integer or string). Closed intervals $[l,u]$ on such a domain are characterized by two parameters l and u ($-\infty \leq l \leq u \leq +\infty$), as are half-open intervals $(l,u]$ and $[l,u)$ or open intervals (l,u).

Definition 3 (Interval completeness) An interval r is said to be *interval complete* (or *complete* for short) in a cache column $S.c$ if and only if all records of $\sigma_{c\in r} S_\mathrm{B}$ are in S (making all individual values of r value complete in $S.c$).

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

In a query, a range predicate can take various forms using the relationships $\Theta \in \{<, \leq, =, \geq, >, \neq\}$. An actual range predicate r_A can easily be mapped to an interval (or two in case of $\neq$), for example, $x > l$ to $(l, +\infty)$. Hence, for simplicity, we identify a range predicate r with its corresponding interval.

3.1.3 Equi-join predicates

How do we obtain the predicate extensions of PSJ queries? The key idea is to use *referential cache constraints* (RCCs) to specify all records needed to satisfy specific equi-join predicates. An RCC is defined between two cache columns, which need not belong to separate tables.

Definition 4 (Referential cache constraint, RCC) A referential cache constraint $S.a \rightarrow T.b$ from a source column $S.a$ to a target column $T.b$ is satisfied if and only if all values v in $S.a$ are value complete in $T.b$.

Our definition of RCCs is equivalent to the one given by Altinel et al [2], which can be easily verified by replacing the use of value completeness with its definition.

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

Of course, the application of RCCs can be chained: A second RCC $T.d \rightarrow U.e$ could expand the predicate extension of $S.c = x \wedge S.a = T.b$ to the predicate extension of $S.c = x \wedge S.a = T.b \wedge T.d = U.e$.

A column can be non-unique (NU) or can be declared unique (U) via the SQL constraint *unique* in the backend DB schema (or via *primary key*, which is basically the same because we do not care about *null* "values" anyway). Depending on the types of source and target columns, we classify RCCs as $1:n$, $n:1$, and $n:m$ and denote them as follows:

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

Note, using RCCs we implicitly introduce a value-based table model intended to support queries. Despite similarities to the relational model, MCs and OCs are not identical to the PK/FK (primary key / foreign key) relationships contained in the backend schema. A PK/FK relationship can be processed

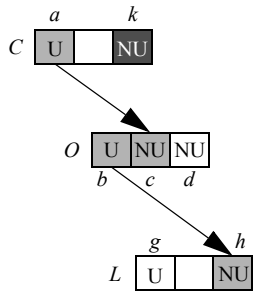


Fig. 2 Cache table collection *COL*

symmetrically, whereas our RCCs can be used for join processing only in the specified direction. There are other important differences: XCs have no counterparts in the backend DB, and a column may be the source of n and the target of m RCCs. In contrast, a column in the role of a primary key may be the starting point of k, but in the role of a foreign key the ending point of only one (meaningful) PK/FK relationship.

Because a very high fraction (probably more than 99 %) of all SQL join queries refer exclusively to PK/FK relationships (they represent real-world relationships explicitly captured by the DB design), we expect almost all RCCs specified between cache tables to be of type MC or OC. As a corollary, XCs and multiple RCCs ending on a specific NU column seem to be very infrequent.

Example 1 Assume cache tables C, O, and L connected by RCCs $C.a \rightarrow O.c$ and $O.b \rightarrow L.h$, where $C.a$, $O.b$, and $L.g$ are U columns and $C.k$, $O.c$, $O.d$, and $L.h$ are NU columns, as illustrated in Fig. 2. In a common real-world situation, C, O, and L could correspond to backend DB tables Customer, Order, and OrderLine. Hence, both RCCs would typically characterize PK/FK relationships that are to be used for join processing in the cache.

The specification of additional RCCs $O.b \rightarrow C.k$ or even $O.c \rightarrow L.h$ and the inverse $L.h \rightarrow O.c$ is conceivable (given join-compatible domains); but such RCCs have no counterparts in the backend-DB schema: When using them for a join of O and C or a cross join of O and L, the user is completely responsible for assigning a meaning to these joins.

3.2 Probing for entry points

RCCs allow us to draw conclusions about predicate extensions that are in the cache, but only if we can rely on some value (or an interval of values) being complete and serving as an entry point. Considering some column $S.c$, how do we know that a value v is value complete there? Obviously, our goal ought to be to provide simple and efficient means for deciding about the completeness of values in the cache.

Of course, each cache-resident value of a U column is value complete by definition. Hence, here we need to care about NU columns only.

A straightforward way to gain control over complete values is to choose a set $F_{S.c}$ of values out of $S.c$'s domain and to enforce completeness of these values whenever they appear in the cache (in $S.c$). In this case, all that is needed to decide whether v is complete in $S.c$ is to check if v exists in $S.c$ (given that v is in $F_{S.c}$ in the first place). This process of using a simple (existence) query on the cache to decide about completeness is called *probing*; the query used is called *probe query* accordingly.

In the simplest form of this idea, the entire domain of $S.c$ is used as $F_{S.c}$. This approach has been used by Altinel et al [2] as a part of their *cache keys*[5] and presumably lead to the notion of *domain completeness* of a column. The choice of this denotation is rather unfortunate because it wrongly suggests that all values of a column $S.c$'s domain (i. e., all values permitted to be in $S.c$) are value complete. Instead, you always have to think of the set $F_{S.c}$ of values that are forcibly complete *whenever* they they appear in the cache. Therefore we prefer the term *column completeness*.

Definition 5 (Column completeness) A cache column $S.c$ is said to be column complete (or *complete* for short) if and only if all values v in $S.c$ are value complete.

Given a complete column $S.c$, if a probe query confirms that value v is present in $S.c$ (a single record suffices), we can be sure that v is value complete and thus evaluate $S.c = v$ in the cache. Unique columns of a cache table are complete per se. In contrast, the completeness of non-unique columns must be enforced specially, as indicated above.

The declaration and enforcement of complete columns raises problems regarding the overall loading behavior of the cache because many sets of records are forced into the cache for the sake of easiest probing. (We will show some of these problems in Sect. 4.) Furthermore, low-selectivity columns or single values in columns with skewed value distributions may cause cache filling actions involving huge sets of records never used later.

Therefore, our first refinement was to enforce completeness only upon a subset of the domain (e. g., to exclude values with low selectivity); however, this approach brings with it basically the same problems as enforced column completeness, albeit to a lesser extent.

We propose a new probing approach [7], which does not require new constraints and thus does not load extra records into the cache. Furthermore, it allows more flexible use of the cache contents than probing in complete columns alone.

The fundamental insight is that RCCs already provide guarantees about complete values in the cache: The source column of an RCC (or more precisely, the values therein) controls which values are complete in its target column. This insight is reflected in our reformulation of the RCC definition (Def. 4) and justifies a special name for source columns in this controlling role.

Definition 6 (Control column) We say that $S.a$ is a *control column* of $T.b$ if there is an RCC $S.a \rightarrow T.b$.

[5] The other part is that cache keys fulfill the function of filling columns (see Sect. 3.3.1).

In general, any given column $T.b$ can have an arbitrary number of control columns (including zero). Whenever a column $S.c$ we would like to use as an entry point for a predicate $S.c = v$ has at least one control column, we have the option of probing in the control columns of $S.c$. If we find the value v in one of these columns, we know that it is value complete in $S.c$ and that we can correctly evaluate the predicate in the cache. This makes the cache usable in a more flexible way than has been possible before.

Example 2 Returning to Fig. 2, we find that columns $C.a$, $O.b$, and $L.g$ are complete (because they are U columns). If probing verifies the existence of single values for $C.a = 1$, $O.b = \alpha$, or $L.g = z$, respectively, we can evaluate, in addition to the predicate type *COL* is designed for, the three predicates

$$C.a = 1 \wedge C.a = O.c \wedge O.b = L.h\,,$$
$$O.b = \alpha \wedge O.b = L.h\,, \quad \text{and}$$
$$L.g = z\,.$$

Since $O.c$ has $C.a$ as a control column, we can furthermore evaluate the predicate

$$O.c = 3 \wedge O.b = L.h$$

if we probe successfully for the value $C.a = 3$.

3.2.1 Negative caching

Using control columns for probing has another benefit besides allowing the more flexible choice of entry points in the cache: It enables *negative caching*, that is the representation of knowledge in the cache that something does not exist, cannot or does not give an answer [4; 7].

Example 3 Imagine that in our *COL* example (Fig. 2) we have a customer $C.a = 1$ that has not placed any orders yet (i. e., there are no records in O_B where $O_\mathrm{B}.c = 1$). We will learn in Sect. 3.2.3 that $O.c$ is a complete column and can thus be used as an entry point for a predicate

$$P' = (O.c = 1 \wedge O.b = L.h)\,,$$

but only if probing verifies the existence of value 1 in $O.c$: So in this case the predicate would have to be evaluated at the backend.

However, if we probe in the control column $C.a$ of column $O.c$ instead, we find value 1 in the cache; we then know that it must be complete in $O.c$; and we can finally evaluate the predicate P' in the cache, which yields an empty but correct result. Hence, the cache contains the knowledge that there are not any order records for this customer in the backend.

3.2.2 Probing strategies

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

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

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

If column $S.c$ is not (always) complete, the best we can do is this:

1. Probe in each control column of $S.c$ in turn. If v is found, success ($S.c$ is an entry point).
2. Failure ($S.c$ cannot be used as entry point).

This strategy has the drawback that we might have to check a lot of control columns to find the right one (which contains v). But in the likely case of only one control column, a single probing operation suffices.

Complete columns can still be useful as an optimization: If column $S.c$ is always complete, we can use a more sophisticated probing strategy that in many cases stops after the first probing operation, even if there are multiple control columns:

1. Probe in $S.c$. If v is found, success.
2. If negative caching is impossible[6] or not cared about, failure.
3. Probe in each control column of $S.c$ in turn. If v is found, success (negative caching).
4. Failure.

Of course, we can skip probing in the complete column entirely and go directly to the control columns (i. e., fall back to the first probing strategy) if this promises to be cheaper in the average case (e. g., if there is only one control column and it has an index).

3.2.3 Complete columns

As we have just seen, cache-supported query evaluation becomes more flexible if we can correctly decide which cache columns are (always) complete and thus have more entry points and probing strategies to choose from.

Example 4 Let us refer again to *COL*. Because $C.a \rightarrow O.c$ is the only RCC that induces loading of records in O, we know that only value-complete values enter the column $O.c$

[6] If referential integrity constraints valid in the backend DB are known by the cache manager, it can immediately exclude negative caching in some cases (e. g., when an owner constraint in the cache corresponds to a foreign key constraint in the backend DB).

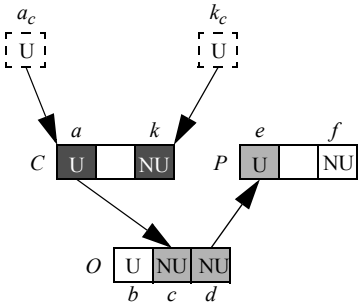


Fig. 3 Cache table collection *COP*: Artificial control columns a_c and k_c of filling columns a and k

and, hence, that all values in $O.c$ are value-complete. So we know that $O.c$ is complete (also called *induced* complete).

Note, additional RCCs ending in $O.c$ would not abolish the completeness of $O.c$, though any additional RCC ending in a different column would: Assume an additional RCC ending in $O.b$ induces a new value v, which implies the insertion of $\sigma_{b=v} O_\mathrm{B}$ into O—just a single record o. Now a new value w of $O.c$ may appear (so far not present), but all other records of $\sigma_{c=w} O_\mathrm{B}$ fail to do so.

As this example shows us, a cache table loaded by RCCs on more than one column cannot have an induced complete column. Therefore, induced completeness is *context dependent*: It may vanish when, for example, new RCCs are added to the cache configuration. But in the absence of other influences (e. g., by future cache constraints or cache keys) we can make the following observation:

Observation 1 A cache column $S.c$ is complete if it is the only column of S that is loaded via one or more RCCs.

There are two types of columns whose completeness is *not* context dependent: We have already encountered U columns as the simplest case. In addition, columns $S.c$ with a *self*-referencing cache constraint (self-RCC) $S.c \to S.c$ are always complete, which is easily deducible from the definition of RCCs (Def. 4).

3.3 Loading predicate extensions

To be able to evaluate a predicate Q in the cache, the cache manager must guarantee predicate completeness for Q by loading all required records into the cache tables.

Example 5 Let us take a look at *COP* (Customer, Order, Product) in Fig. 3, which is a variant of our *COL* example and includes an owner constraint (OC) $O.d \to P.e$.

Assume the predicate of a PSJ query to be evaluated on *COP* is

$$Q_1 = (C.k = x \wedge C.a = O.c \wedge O.d = P.e)\,.$$

An example of Q_1's predicate extension is sketched in Fig. 4, where records are represented by dots and value-based relationships by lines. To establish value completeness for the

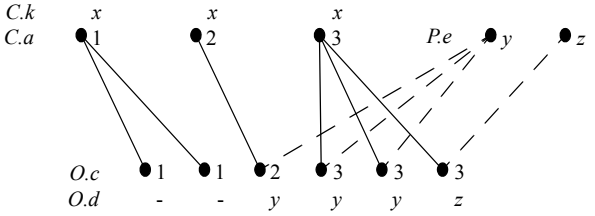


Fig. 4 Construction of a predicate extension for *COP*

value x of column $C.k$, the cache manager loads all records of $\sigma_{k=x} C_\mathrm{B}$ in a first step. For each of these records loaded, the RCC $C.a \to O.c$ must be fulfilled (PK/FK relationships, solid lines); that is, all values of source column $C.a$ (1, 2, 3 in the example) must be made value complete in the target column $O.c$. Finally, for all values present in $O.d$ (y, z), the RCC $O.d \to P.e$ makes their counterparts complete in $P.e$ (FK/PK relationships, dashed lines).

Following the RCCs, the cache manager can construct predicate extensions using only simple loading steps based on equality of values. Accordingly, it can correctly evaluate the corresponding queries locally.

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

3.3.1 Candidate values in filling columns

Besides RCCs, a second type of cache constraint is needed in order to establish a parameterized loading mechanism: Attached to selected *filling columns*[7] are sets of *candidate values*, which alone initiate the loading of predicate extensions when they are referenced by user queries.

Definition 7 (Candidate value) A candidate value v for a filling column $S.f$ belongs to the domain of $S_\mathrm{B}.f$. Whenever the predicate $S.f = v$ is referenced by a query, v is made value complete in $S.f$.

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

How do records get into a cache table? As a consequence of a cache miss attributed to the candidate value v, the cache manager satisfies the value completeness for v by fetching

[7] For the course of this document, we assume single filling columns. A multi-column mechanism different from multiple single columns is conceivable; in this case, values would be composed of the simple values belonging to the participating columns.

all required records from the backend and loading them into the respective cache table. Hence, the cache is ready to answer the corresponding equality query locally from then on.

Apparently, a reference to a candidate value v serves as a kind of indicator that, in the immediate future, locality of reference is expected on the predicate extension determined by v. Candidate values therefore carry information about the future workload and sensitively influence caching performance. Hence, candidate values must be selected carefully. In an advanced scheme, the cache manager itself takes care that only those values with high re-reference probability become and stay candidate values. By monitoring the query load, the cache manager can dynamically optimize this set of values that trigger the loading of predicate extensions into the cache. In a straightforward case, the database administrator (DBA) specifies this set of candidate values.

A set $C_{S.f}$ of candidate values can be specified as an exhaustive set (the domain itself), an enumeration, a range, or as other predicates; candidate values can be expressed positively (recommendations) or negatively (stop-words).

3.3.2 Control columns revisited

Let us repeat: The subset of candidate values of a filling column $S.f$ that actually are in the cache controls which values are complete in $S.f$. Does that not sound familiar?

It is beneficial to introduce an artificial control column f_c for each filling column f: This allows uniform treatment of all cache columns with regard to probing and filling.

This artificial control column is a U column of a separate, anonymous table with an RCC $f_c \rightarrow f$ pointing to the filling column f. In Fig. 3 we have illustrated this situation for the filling columns $C.a$ and $C.k$ of our *COP* example (dark-gray columns).

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

Example 6 In order to illustrate the interplay of filling columns, artificial control columns, and candidate values, let us start from an empty cache in Fig. 3: Tables C, O and P are empty, as is control column k_c. Let $C_k = \{x, z\}$ be the set of candidate values of filling column k.

Imagine two predicates $C.k = v$ with different values v arriving at the cache and producing cache misses.

- Value $v = x$ is a candidate value. Because of the cache miss it is inserted into k_c; this makes x complete in $C.k$ and subsequently loads all dependent records into the cache (i. e., , the predicate extension known from Fig. 4).
- Value $v = y$ is *not* a candidate value. Despite the cache miss it is neither inserted into k_c nor made complete in $C.k$.

Now imagine that a new record with $C.k = z$ is loaded into the cache table C (due to a cache miss on the other filling column $C.a$). Value z is a candidate value of filling column $C.k$, but because the cache miss did not occur on $C.k$, nothing happens: Value z does not need to be inserted into k_c; it does not matter whether it is complete in filling column $C.k$ or not.

Our artifical control columns have been inspired by the *control tables* in the MTCache project [15; 21], which are used in quite a similar way: In MTCache, a set of stacked materialized views is used to describe the cache contents, each dependent on the contents of another view (which resembles RCCs) or ultimately on the contents of a control table.

In following figures and examples we will assume that you are aware of an artificial control column behind each filling column and we will no longer depict or draw attention to those unless it is crucial to the discussion. Just keep in mind that the only special thing about filling columns is their sensitivity to references of values in equality predicates, which leads to new values in their artificial control columns. With respect to probing, query evaluation, and even filling via RCCs they behave exactly like any other column.

3.3.3 Range predicates

When loading extensions of range predicates on a filling column, the cache manager must take into account intervals r_i that are in the cache already. In the simplest case, the cache could be populated with all records belonging to the interval r_A when a range query with predicate $S.f \in r_A$ is evaluated and leads to a (partial) cache miss (some sub-intervals of r_A may already be in the cache). Cache loading makes r_A interval complete in cache column $S.f$ such that subsequent queries with range predicates contained in r_A can correctly be answered from the cache.

To be aware of the intervals r_i present in the cache, the cache manager holds information on them in an ordered list (this ordered list could be regarded as an implementation of the artificial control column). As soon as two adjacent r_i overlap, they are merged into a single interval. Hence, queries with range predicates r_A contained in an r_i that is present ($r_A \subseteq r_i$) can be evaluated locally. If r_A only overlaps r_i, the range predicate can only partially be evaluated in the cache. If the remaining portions of r_A are not specified in the set of candidate values, no further cache population takes place. Otherwise, the missing intervals could be made complete.

Note that the selectivity and potential locality of intervals must strictly be controlled to prevent "performance surprises"due to excessive cache loading. This is even more important than in the case of single values and is especially true for intervals where l or u is infinite: Again, cache filling should be refined with a set of candidate values.

3.4 Cache groups

In general, our caching mechanism supports PSJ queries that are characterized by predicate types of the form

$$(RP_1 \vee \ldots \vee RP_n) \wedge EJ_1 \wedge \ldots \wedge EJ_m \qquad (1)$$

where RP_i, $1 \leq i \leq n$, is a range predicate (an equality predicate in the simplest case) on a specific cache table called *root table* and the EJ_j, $1 \leq j \leq m$, correspond to RCCs that (transitively) connect the root table with the remaining cache tables involved.

We can use cache tables, filling columns and RCCs to specify *cache groups*, which is our unit of design to support a specific predicate type in the cache.

Definition 8 (Cache group) A cache group is a collection of cache tables linked by a set of RCCs. A distinguished cache table is called the *root table R* of the cache group and holds one or more filling columns. The remaining cache tables are called *member tables* and must be reachable from *R* via RCCs.

Example 7 Hence, our *COP* example constitutes a simple cache group with root table *C*, filling columns $C.a$ and $C.k$, and two RCCs to the member tables *O* and *P*. It is designed for the following predicate type ($n = 2$ and $m = 2$ in Eq. (1)):

$$(C.a = v_1 \vee C.k = v_2) \wedge C.a = O.c \wedge O.d = P.e\,.$$

Let us summarize our findings about how cache groups are populated and where their entry points can be found (with complete columns as a means for optimization).

A cache table T can be loaded via one or more RCCs ending in one or more of its columns. (A filling column is a special case in that an RCC coming from its artificial control column ends there.)

A column $T.c$ is a potential entry point if

- it has control columns (i. e., it has incoming RCCs) or
- it is a complete column.

A column $T.c$ is complete (at all times) if

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

3.5 Units of cache unloading

Flexible adjustment of the (dynamic) set of candidate values that are present in the cache is key to cache adaptivity.[8] Because a probe query always precedes the actual query evaluation, completeness for a value v in a filling column $S.f$ can be abolished at any time by removing v from the control column.

[8] This is orthogonal to displacement of cache pages to a local disk. If we allow this kind of flexibility in the cache, probing and query processing has to consider both main memory and local disk contents.

In the simplest case, there may be no removal at all, and thus a value, once made complete, is left in the cache forever, or there may be a periodical purge (*complete unloading*) by continuing with an empty cache. Alternatively, complex algorithms could be applied to support *selective unloading* (i. e., the predicate extension for $S.f = v$ is removed from the cache, if the re-reference probability for candidate value v sinks). Note, besides the costs for memory and storage space, there is always a trade-off between the savings for query evaluation and the penalties for keeping the records of a predicate extension consistent with their state in the backend.

How difficult is it to cope with a unit of loading and unloading? Such a unit is dependent on a candidate value and consists of a set of *cache instances*, each of which is a minimal collection of records satisfying all RCCs for a single root record with that candidate value. For example, Fig. 4 shows a single unit of loading / unloading (for candidate value x) and three cache instances having for $C.a$ the values $1, 2$, and 3 in their root records. Depending on their complexity, load units and, in turn, their cache instances may exhibit good, bad, or even ugly maintenance properties.

The good load units are disjoint from each other and the RCC relationships between the contained records form trees for their cache instances. A simple example is provided by cache group *COL* in Fig. 2 on page 5. Here a newly referenced candidate value of $C.k$ (NU) causes a forest of such trees to be loaded, which, in case of unloading, can be removed without interference with other cache instances.

For the bad load units, the cache instances form directed acyclic graphs and weakly overlap with each other. Cache group *COP* in Figures 3 and 4 on page 7 is an example where several cache instances may share records of cache table *P*. Because they may also overlap with cache instances of other load units, cache loading must beware of duplicates. Accordingly in case of unloading a cache unit, shared records must be removed only together with their last sharing cache instance.

To maintain cache groups with cross constraints or RCC cycles can be characterized as ugly, because load units and cache instances may strongly overlap so that duplicate recognition and management of shared records may dominate the work of the cache manager. The problems arising with cycles during loading and unloading will be our subject of discussion in Sect. 4.4.

4 Safeness of cache groups

So far, we know how to configure a cache group by specifying the participating cache tables, the RCCs connecting them, and the filling columns, which initiate the population of the cache group. Using complete values as entry points for query processing in the cache, we can produce correct results for eligible query predicates. However, it is unreasonable to accept all conceivable cache group configurations. As cache misses on filling columns may provoke unforeseeable load-

ing operations, we ought to estimate the effects and costs of those operations as accurately as possible. In particular, we would like to prevent excessive, uncontrollable cache table loading.

Although the cache can be populated asynchronously to the transaction that observes the cache miss and therefore a burden on the response time of this transaction can be avoided, uncontrolled loading is highly undesirable: It will influence the transaction throughput in heavy-workload situations, because substantial extra work, which can hardly be estimated and preplanned, will be required from the cache and backend DB servers.

Specific cache group configurations may even exhibit a *recursive* loading behavior, which jeopardizes their caching performance. Once cache filling is initiated, the enforcement of cache constraints may require multiple phases of record loading until all specified constraints are re-satisfied.

Cache groups are called *safe* if such a recursive loading behavior cannot occur under any circumstances. Therefore, we elaborate safeness conditions for cache group configurations: Upon a miss on a filling column, we want the initiated cache loading to stop after a *single pass* of loading operations through the cache tables—starting from the root table to all member tables reachable via RCCs.

Apparently, RCC cycles play *the* critical role for recursive loading situations: They can make a cache group unsafe. For this reason, we explore cycles in detail, looking at safeness: Our goal is to find a set of design rules that will preclude unsafe cache groups. In Sect. 4.3 we will shortly revisit query evaluation correctness in the context of RCC cycles.

4.1 Properties of RCC cycles

An *RCC cycle* (or *cycle* for short) is a closed path of tables connected by RCCs. In an *atomic* cycle each table is visited only once (apart from start and end table being the same); such a cycle cannot be decomposed into smaller subcycles. An atomic cycle can furthermore be *isolated*, which means it does not share tables with any other cycle. (Non-atomic cycles cannot be isolated: They share tables with their subcycles.) If only U columns participate in a given cycle, we call it a *U-cycle*.

An RCC cycle is said to be *homogeneous*, if it involves only a single column per table, for example, $T.c \to V.d$, $V.d \to W.e$, $W.e \to T.c$ (Fig. 5). In contrast, a cycle is said to be *heterogeneous*, if it involves more than one column in some participating table [2].

An *initiating* column of a cycle is a column of some cache table X (e. g., T in Fig. 5) whose values initiate the loading of cycle-induced records. This means that table X must be reached by an RCC not belonging to the cycle. In general, more than one column can be initiating in a given cycle.

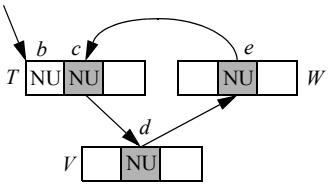


Fig. 5 A homogeneous RCC cycle

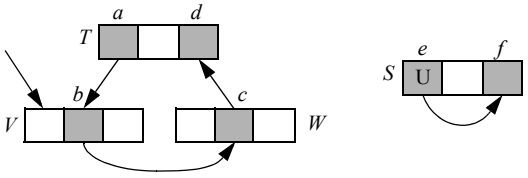


Fig. 6 Heterogeneous RCC cycles

4.1.1 Homogeneous cycles

Example 8 In Fig. 5, we have sketched an isolated homogeneous RCC cycle comprising only NU columns:

$$T.c \to V.d \to W.e \to T.c\,.$$

Let $T.c$ be its only initiating column (the RCC responsible for that is indicated). If cache table T is populated with some records that have the values $VS = \{v_1, \ldots, v_m\}$ in $T.c$, these values become complete in the succeeding column $V.d$, because RCC $T.c \to V.d$ forces all records of $\sigma_{d \in VS} V_\mathrm{B}$ into cache table V.

By the same argument, in each column of the cycle a subset of VS is made complete: If some value v_j of VS does not occur in some (backend) table belonging to the cycle (e. g., if $\sigma_{d=v_j} V_\mathrm{B}$ is empty), it cannot show up in the cache, which is irrelevant for its completeness in $V.d$. But v_j will not be loaded into the following columns. This means only a subset $R \subseteq VS$ becomes value complete in the initiating column $T.c$, before the loading stops.

The loading actions caused by an isolated homogeneous cycle stop at the latest when the initiating table is reached during the maintenance of RCCs: Because no new values ever populate the initiating column, this process is not recursive.

4.1.2 Heterogeneous cycles

We take a look at isolated *heterogeneous* cycles now: What are the properties of NU columns participating in such a cycle?

Example 9 Assume in Fig. 6 that $V.b$ is an initiating column and receives a set VS of values. The RCCs $V.b \to W.c$ and $W.c \to T.d$ populate columns $W.c$ and $T.d$ then. The set VS may shrink along this population process to $R \subseteq VS$. The records $\sigma_{d \in R} T_\mathrm{B}$ inserted into T may carry a different set of values in $T.a$, which keeps the loading of this cycle “running”.

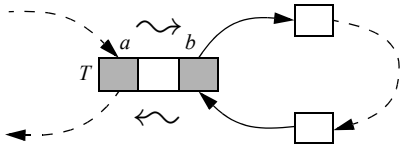


Fig. 7 Pair of smuggler relationships (path p)

We denote this effect that "smuggles" a set NV of new values into the RCC cycle by $T.d \rightsquigarrow T.a$. Obviously, NV may force the cycle-induced loading to begin a second round and so on.

Definition 9 (Smuggler relationship) Let $T.a$ and $T.b$ be two different columns of the same cache table T. We use the term *smuggler relationship between $T.a$ and $T.b$*, in symbols $T.a \rightsquigarrow T.b$, to describe the influence of $T.a$'s values on $T.b$'s values due to the records of T_B inserted into T.

As we have seen in the example, isolated heterogeneous RCC cycles definitively cause extensive recursion [10]. This is even true in very simple cache group configurations.

Example 10 Assume table S in Fig. 6 with an RCC $S.e \rightarrow S.f$, which may represent the employee hierarchy of some company. As soon as a single record is inserted into table S (representing a manager with $S.e = v$ as primary key), the entire (sub-)hierarchy of S under this person is recursively loaded into S. Even in the degenerate case of a U/U RCC $S.e \rightarrow S.f$, recursive loading operations may happen in S. Such an RCC can be used to model linear list structures. A single reference to some item (i. e., a value of $S.e$) forces all following list items into S.

4.2 Dangerous cache groups

Having come so far, we are capable of evaluating two classes of cycles in terms of safeness: Isolated homogeneous cycles are safe, isolated heterogeneous cycles are not. In order to generalize our findings to arbitrary cycles, let us study in detail the properties of the specific effect that makes (isolated) heterogeneous cycles unsafe.

Consider two columns $T.a$ and $T.b$ of a cache table T and the smuggler relationship $T.a \rightsquigarrow T.b$ between them: Whenever a set R_a of values reaches $T.a$, a different set N_b of (new) values appears in (or is smuggled into) $T.b$. Of course, the same is true for the other direction $T.b \rightsquigarrow T.a$ with different sets R_b and N_a. If we imagine an RCC cycle in which both relationships occur, is there any dependency between these two?

Assume that the path

$$p = T.a \rightsquigarrow T.b \rightarrow \cdots \rightarrow T.b \rightsquigarrow T.a$$

is the interesting part of such an RCC cycle (Fig. 7). We start in $T.a$ with a set R_a of values, walk along the path p observing the induced sets of values and finally reach $T.a$ again with a set N_a of values. If $N_a \setminus R_a$ is not empty, then new values have been smuggled in; by analogy to the isolated heterogeneous case, any cycle containing p would be unsafe. Let us assume further that the subcycle $T.b \rightarrow \cdots \rightarrow T.b$ of p is homogeneous, so that it does not introduce new values, which would disturb our considerations (i. e., $R_b \subseteq N_b$).

We distinguish two cases, $T.b$ being U or NU. If $T.b$ is unique, there is in T only a single record per value of this column. Any value v in $T.a$, which corresponds to some set of records $V \subseteq \sigma_{a=v} T_\mathrm{B}$, will thus be mapped by $T.a \rightsquigarrow T.b$ to some values W unique in $T.b$. Therefore, when W is mapped back by $T.b \rightsquigarrow T.a$, we get v again: $M = \pi_a \sigma_{b \in W} T_\mathrm{B}$ consists only of this single value. Together with $R_b \subseteq N_b$ this yields $N_a \subseteq R_a$ for the U case: No new values appear; all value changes have been compensated. Because of this behavior we arrive at the following recursive definition:

Definition 10 (Compensating smuggler relationship) We call two smuggler relationships $a \rightsquigarrow b$ and $b \rightsquigarrow a$ (in this order) *compensating* with respect to a path p, if b is a U column and there are either no other smuggler relationships between them on the path or only compensating pairs.

We write $x \leftrightsquigarrow y$ if we want to refer to the pair $x \rightsquigarrow y$ and $y \rightsquigarrow x$, whether it is compensating or not.

If $T.b$ is non-unique, the main conclusion of the U case does not apply: M may contain new values besides v, and N_a may thus receive new values (compared with R_a).

In summary, every path that contains only pairs of compensating smuggler relationships (if any) does not introduce new values (not counting the different sets of values in columns between the pair; they are shielded from the outside). Any other path, however, may introduce new values, which leads to recursive loading in case of a cycle. Therefore, a cache group is safe if and only if it does not contain heterogeneous RCC cycles of such kind.

4.3 Correctness in cycles

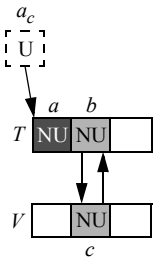
RCC cycles do not offer any new challenges regarding query evaluation correctness: For our probing approaches it does not matter whether there are cycles or not, because they examine a table and its neighbors locally. The use of control columns, the determination of complete columns for optimized probing, and the subsequent use of RCCs for join processing in the cache only rely on the validity of RCCs, which is guaranteed in every case.[9]

But does a homogeneous cycle not resemble a somewhat larger self-RCC, so that column completeness is guaranteed for cycle columns? This would allow us to use our advanced probing strategy.

Example 11 Let us return to the isolated homogeneous cycle in Fig. 5:

$$T.c \rightarrow V.d \rightarrow W.e \rightarrow T.c\,.$$

[9] The completeness of U columns may also be used; but this is invariant and independent of cycles, too.



T	a	b
t_1	x	1
t_2	x	2
t_3	y	1
t_4	y	2

V	e
v_1	1
v_2	1
v_3	2

Fig. 8 Unloading in the presence of a cycle (*TV*)

If cache table T is populated with some records that have the values $VS = \{v_1, \ldots, v_m\}$ in the initiating column $T.c$, these values are not (necessarily) value complete in $T.c$ because T does not contain all records of $\sigma_{c \in VS} T_B$, which are needed. Nevertheless, the succeeding column $V.d$ is (induced) complete, because RCC $T.c \rightarrow V.d$ forces all records of $\sigma_{d \in VS} V_B$ into cache table V.

By the same argument, all columns in the cycle become complete, for example, $W.e$ via $V.d \rightarrow W.e$, *except* for the initiating column $T.c$: If some value v_j of VS does not occur in some table belonging to the cycle (e. g., $\sigma_{d=v_j} V_B$ is empty), it cannot show up in the cache, which is irrelevant for the completeness of all cycle columns but the initiating column. Assume, some values v_j were lost in this way while the cycle was populated. Then the set of remaining values R in the last but one table, from where an RCC is closing the cycle (here W with $W.e \rightarrow T.c$), is a proper subset of VS ($R \subset VS$). In this case, the cycle-closing RCC does not load *all records for all values of VS* into the "initiating" table. Hence, such an initiating column may carry some values v_j that are not complete in the cache and, in turn, the column completeness is violated.

Because it is not guaranteed that all cycle-induced loading operations reestablish (through $R = VS$) value completeness in initiating columns, we cannot use our advanced probing strategies that rely on the columns' completeness. This result refutes a proof given by Altinel et al [2]: They claim that all the columns involved in a homogeneous cycle were complete; in their proof they neglect the fact that (in our words) a value v_j can be complete in a column even if it does not exist in this column. This means that not all values are necessarily propagated around the whole cycle.

As we have seen in the preceding example, we can guarantee (induced) completeness only for cycle columns that are the only columns in their tables for which loading is induced by an RCC; this matches Observation 1.

4.4 Unloading in cyclic cache groups

Although some types of cycles are acceptable in terms of safeness, they are very hard to manage such that unloading individual candidate values from cache tables containing cycles does not always pay off.

Example 12 Assume cache group *TV* in Fig. 8 is initially empty[10], its filling column's set $C_{T.a}$ of candidate values is $\{x, y\}$ and a query predicate $T.a = x$ arrives. As a consequence, x is loaded into control column a_c in the example and is is made complete in $T.a$ by loading t_1 and t_2 into T. RCC $T.b \rightarrow V.c$ forces v_1, v_2, and v_3 into table V, making values 1 and 2 value complete there. In turn, $V.c \rightarrow T.b$ implies loading of t_3 and t_4 into T thereby making 1 and 2 value complete (let t_3 and t_4 be the only records having y-values in $T_B.a$). Because y was not referenced by a query predicate, it is not made complete (despite being a candidate value). Assume in this situation, a query predicate $T.a = y$ arrives, which explicitly makes y complete ($a_c : x, y$), although no further record is loaded: Value y has already been complete before, but this fact was not derivable from the cache contents.

When either x or y is not referenced for a long time, its predicate extension should be unloaded. Unloading y drops it from a_c and attempts to remove all cache instances with root records carrying value y in $T.a$. Removing t_3 and t_4 (in either sequence or at once) would violate $V.c \rightarrow T.b$; these records must therefore remain in the cache. In the same way, unloading x (besides dropping it from a_c) has to check whether t_1 and t_2 can be removed from T. However, unloading of any of them would violate $V.c \rightarrow T.b$ for value 1 or value 2.

In this example we loaded and tried to unload the predicate extensions of two candidate values x and y, which were involved in a simple cycle. Because the RCC cycle forced all cache instances to stay in the cache, removing each cache instance or even each unload unit separately was unsuccessful. The example further shows that testing which cache instances can be or cannot be removed is very expensive in the presence of cycles and is not always successful when approached step by step. In cyclic cache groups, only an entire collection of intertwined units of unloading—corresponding to a set of candidate values (in our example x and y)—could simply be removed at once, which, however, often would affect high-locality candidate values, too. Furthermore, to detect an such an intertwined collection is very hard and, in the worst case, the collection could comprise the entire cache content. Therefore, depending on a cost model, we propose to purge the cache from time to time and to continue with an empty cache in such *ugly* cases.

5 Cache group design

When designing a cache group, we proceed top-down and start with a predicate type—of the form given in Eq. (1)—that we would like our cache group to primarily support. We can then easily define a root cache table with a filling column for each range predicate RP_i in our predicate type. For

[10] In DBCache [2], $T.a$ would have been loaded via a cache key, which would create an unsafe cache group TV, because two complete NU columns appear in a cache table.

each equi-join predicate EJ_j, in turn, we add a corresponding RCC to our cache group design, creating new (member) cache tables as necessary.

On the one hand, a cache group so designed should enable an as flexible use as possible for predicate evaluation. On the other hand, "dangerous" loading behavior—that is, excessive population of cache groups—must be prevented. Hence, only safe cache groups are acceptable.

5.1 Restrictions for cache group design

In the following, we capture our observations in some design rules for cache groups, which guarantee our design objective of safeness. We proceed with increasing complexity and coverage of cache group configurations.

As already indicated in Sect. 4, we do not need to restrict the selection of filling columns in root tables of cache groups. Therefore, we need to deal with RCC cycles only.

Rule 1 Isolated heterogeneous RCC cycles are not allowed.

Isolated heterogeneous RCC cycles are dangerous, no matter, whether or not both cycle columns in some table are U/U, U/NU or NU/NU. In contrast, isolated homogeneous RCC cycles are acceptable, because cache table loading stops after a single round of cycle-induced loading operations.

Even multiple homogeneous U-cycles that share some cache tables are acceptable: A new value in the column of one RCC cycle induces a single new value in the column of another RCC cycle, if both columns are in the same table. This value induces a single round of loading in the second cycle stopping at the initiating column at the latest. (The U-cycles are subcycles of a heterogeneous cycle with only compensating pairs of smuggler relationships.)

Rule 2 Heterogeneous RCC cycles with non-compensating smuggler relationships are not allowed.

In Sect. 4.2 we have discovered which heterogeneous cycles are unsafe; those ones are prohibited here. This rule includes rule 1 as a special case: In isolated cycles there are no pairs of smuggler relationships, which could be non-compensating.

A proof that these two design rules identify exactly all unsafe cache group configurations is given in Appendix A.

5.2 Optimization of cache group use

If we observe the two rules derived above when designing a cache group, we come up with a set of cache tables that guarantee the evaluation correctness of query predicates in the cache and that do not hide performance surprises. We may exploit all values known to be complete to check and evaluate equality or range predicates; the RCCs serve equi-join operations where the arrow of an RCC indicates the join direction.

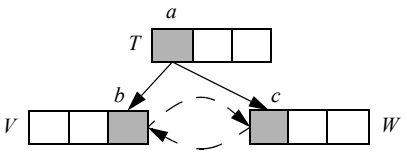


Fig. 9 Synchronous RCCs and resulting optimization RCCs (G_1)

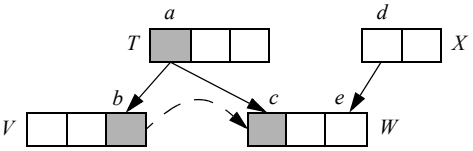


Fig. 10 Optimization RCCs depend on the context (G_2)

5.2.1 Optimization RCCs

In special cases, we are able to extend the possibilities of how joins can be performed without modifying the cache contents. If we have an RCC $r = S.a \rightarrow T.b$ between tables S and T where $S.a$ is a complete column, and if $T.b$ is a complete column because T is targeted only by r, then we can infer that the inverse RCC $T.b \rightarrow S.a$ always holds. Hence, we can enhance our cache group design by an RCC without any additional cost. Such RCCs are called *optimization RCCs*; they are context dependent like the induced completeness of a column. Note that optimization RCCs do not interfere with induced completeness of any column of their target table.

Example 13 Let us take a look at more advanced examples of optimization RCCs in cache group G_1 (Fig. 9): Let us assume that columns $V.b$ and $W.c$ are loaded only by a pair of RCCs originating at the same column $T.a$; we might call such RCCs *synchronous*, because they make the same set of values in their target columns value complete: Each value v of $T.a$ is complete in both $V.b$ and $W.c$, even if it does not exist there. Since no other values reach those columns, we can add the optimization RCCs $V.b \rightarrow W.c$ and $W.c \rightarrow V.b$: If a value exists in one column, it is complete in the other one. Of course, every value in one of these columns is value complete in the same column as well; in other words, $V.b$ and $W.c$ are induced complete.

Adding another table X and an RCC $X.d \rightarrow W.e$ inhibits induced completeness of $W.c$, because new values are smuggled into $W.c$ and do not get complete there (G_2; Fig. 10). Because these new values do not occur in $V.b$ (even if they exist in V_B), the optimization RCC $W.c \rightarrow V.b$ no longer holds. In contrast, RCC $V.b \rightarrow W.c$ remains valid: Surplus values in the target column of an RCC never matter; value completeness of relevant values is never overridden.

All that does matter for the decision whether or not an optimization RCC $x \rightarrow y$ is valid is the set of values for which we *know* they are value complete in y. Let us denote this set by $K(y)$ for any column y. In general, $K(y)$ is different from the set $E(y)$ of values that *exist* in y. If we can deduce from the constraints specified for a cache group that $K(y) \supseteq E(x)$ holds for an arbitrary pair of columns x and y,

we are allowed to add $x \rightarrow y$. This is a direct consequence of the definition of RCCs.

Example 14 Let us apply this thought to G_2 and let us collect what we know about the sets $E(x)$ and $K(x)$ for columns x: Every value existing in $T.a$ is complete in both $V.b$ and $W.c$, that is, $E(T.a) \subseteq K(V.b)$ and $E(T.a) \subseteq K(W.c)$. Column $V.b$ is loaded only via RCC $T.a \rightarrow V.b$, so we have $E(V.b) \rightarrow E(T.a)$. Column $W.c$ is loaded via $T.a \rightarrow W.c$, but also via $W.e \rightsquigarrow W.c$; we describe the values smuggled into $W.c$ by means of a function f, which yields

$$E(W.c) \subseteq E(T.a) \cup f_{W.e \rightsquigarrow W.c}(E(W.e)) .$$

From this set of subset relationships we can now deduce that $E(V.b) \subseteq E(T.a) \subseteq K(W.c)$, which confirms optimization RCC $V.b \rightarrow W.c$.

Such value-set considerations could lead to an algorithm that finds all valid optimization RCCs and induced complete columns of a given cache group. Before striving for this goal, however, we still have to clarify the influence of smuggler relationships (hidden above behind f) and incorporate their compensating behavior in cycles into the reasoning.

5.2.2 More exotic cases

We revisit the differences between cache groups G_1 and G_2 in Figures 9 and 10 and assume that $T.a$ is complete in both. Given a value x in $T.a$ we can then evaluate the following predicate P in G_1:

$$P = (T.a = x \wedge T.a = W.c \wedge W.c = V.b) .$$

In G_2, however, we had to remove the optimization RCC $W.c \rightarrow V.b$ because of the additional values smuggled into column $W.c$ via $W.e \rightsquigarrow W.c$. Therefore P does not seem to be eligible for evaluation in G_2 at first sight. But looking more carefully, at $W.c$ we are restricted to values that also exist in $T.a$. We thus stay within the set of values that are complete in both $W.c$ and $V.b$ due to synchronous RCCs. As we know from G_1, the join $W.c = V.b$ is correct for these values. Therefore the predicate P can be also evaluated in G_2.

We have just discovered something that behaves like a path-dependent RCC: Whenever we reach cache table W via $T.a \rightarrow W.c$ during the construction of evaluable predicates, we are allowed to use $W.c \rightarrow V.b$; whenever we get there via a different path, we are not.

6 Use of cache groups

6.1 Single cache groups

Let us illustrate all of our cache group concepts in a realistic setting. In Fig. 11, we have visualized cache group G_3 consisting of four cache tables Customer (C), Order (O), OrderLine (L), and Product (P), which are connected by three RCCs $C.cid \rightarrow O.cid$, $O.oid \rightarrow L.oid$, and $L.pid \rightarrow P.pid$.

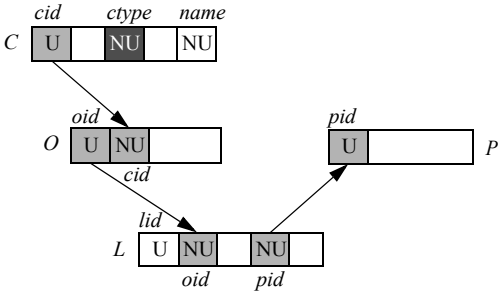


Fig. 11 Cache group G_3

Let the column $C.ctype$ be specified as the only filling column with a set $C_{C.ctype}$ of candidate values. Obviously there are a number of columns that can be used as entry points of queries (for complete values):

- by probing in control columns
  - filling columns: $C.ctype$
  - other columns having control columns: $O.cid$, $L.oid$, $P.pid$
- by probing directly in the column
  - U columns: $C.cid$, $O.oid$, $L.lid$, $P.pid$
  - induced complete NU columns: $O.cid$, $L.oid$

In G_3, we can add two optimization RCCs $O.cid \rightarrow C.cid$ and $L.oid \rightarrow O.oid$ to identify all applicable equi-joins together with their directions. Because they do not influence the loading of G_3, induced completeness of $O.cid$ and $L.oid$ remains intact.

As we know, if a probing operation for some entry column $T.c$ verifies a value x to be complete, we can use $T.c$ to start the evaluation of $T.c = x$. 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 the control column of $C.ctype$, then the predicate

$$C.ctype = \text{'gold'} \wedge C.cid = O.cid \wedge O.oid = L.oid \wedge L.pid = P.pid$$

can be processed in the cache correctly. Because the predicate extension (with all columns of all cache tables) is completely accessible, we may specify any column for output. Of course, we can refine a correct predicate by "and-ing" additional selection terms (referring to cache columns) to it, for example,

$$C.ctype = \text{'gold'} \wedge C.name \text{ like 'Smi\%'} \wedge O.oid > 99 \wedge \ldots$$

To illustrate the subtle interplay of RCCs and induced complete columns, we specify an additional RCC $P.pid \rightarrow L.pid$ in cache group G_4 shown in Fig. 12—assuming the presence of the former optimization RCCs. Since now $L.oid$ is not an induced complete column anymore, RCC $L.oid \rightarrow O.oid$ is no longer redundant; in contrast, it contributes to the loading of additional records into cache table O. In turn, $O.cid$ looses its induced completeness. The RCC $O.cid \rightarrow$

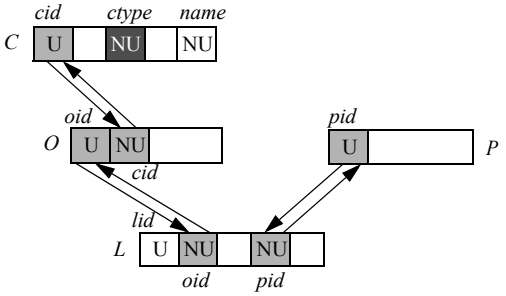


Fig. 12 Cache group G_4

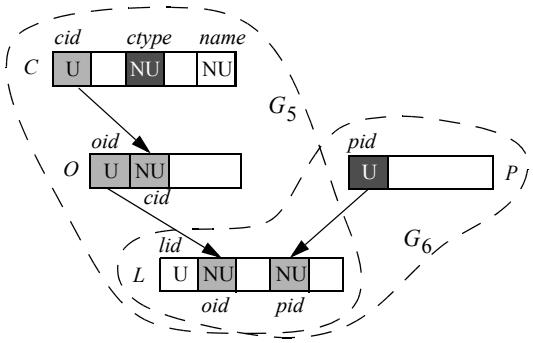


Fig. 13 A cache group federation F

$C.cid$ now enforces the loading of additional records into cache table C; these records have a disastrous effect on the population of G_4. They imply the loading of additional records into O via $C.cid \rightarrow O.cid$, and, in turn, these records force additional records into L via $O.oid \rightarrow L.oid$, which again populate P with additional records via $L.pid \rightarrow P.pid$ and so on. As you may have noticed, we have created a heterogeneous cycle.

The NU filling column $C.ctype$ may receive new values $V = \{v_1, v_2, \dots\}$ when additional records are loaded into table C through $O.oid \rightarrow C.cid$. But even if these values are candidate values of $C.ctype$ (i. e., they are in $C_{C.ctype}$), they need not be made complete in our approach.[11]

We can describe the dependencies in the overall cycle by

$$\begin{aligned} C.cid \rightarrow O.cid \rightsquigarrow O.oid \rightarrow L.oid \rightsquigarrow L.pid \rightarrow P.pid \\ \rightarrow L.pid \rightsquigarrow L.oid \rightarrow O.oid \rightsquigarrow O.cid \rightarrow C.cid\,. \end{aligned}$$

It can be perceived as a combination of three homogeneous subcycles, which are coupled by pairs of smuggler relationships:

$$C.cid \rightleftarrows O.cid \leftrightsquigarrow O.oid \rightleftarrows L.oid \leftrightsquigarrow L.pid \rightleftarrows P.pid\,.$$

While each of these cycles is non-recursive in its isolated form, their combination is no longer harmless, because the individual cycles exchange values. The second pair of smuggler relationships $L.oid \leftrightsquigarrow L.pid$ (NU/NU) is not compensating and makes this cache group unsafe.

Of course, if we remove the former optimization RCC $L.oid \rightarrow O.oid$ from G_4, the middle cycle disappears. All remaining cycles provoke no harm. However, as compared to the situation in G_3, the additional RCC $P.pid \rightarrow L.pid$ abolishes the induced completeness of $L.oid$: It completes a homogeneous cycle $L.pid \rightarrow P.pid \rightarrow L.pid$ with $L.pid$ as its initiating column, which smuggles new values into $L.oid$.

[11] If $C.ctype$ were a cache key [2], the values V would have to be made complete, which would insert sets of records $\sigma_{ctype \in V} C_B$ into C, which as a second filling source—independently of the records that already are in the heterogeneous cycle and keep it "running"—would pump this additional "feed" into the cycle.

6.2 Federation of cache groups

Often multiple applications should be supported by a DB cache. Hence, for each application we must design a cache group by observing the restrictions given in Sect. 5.1. Some of these individually designed cache groups may have some cache tables in common. The transparency requirements for cache tables, however, demand that each table (logically) appears only once in the cache.

For this situation, there is no perfect solution[12]. If we manage disjoint cache groups overlapping in some tables, we may necessarily create copies in the cache. Then we have to cope with keeping record copies consistent. Furthermore, query optimization becomes more complex, because possible cache table collections in separate cache groups have to be considered. When a query is evaluated, the cache manager may have to probe in several tables to identify a matching predicate extension.

For these reasons, cache groups overlapping in some tables should be merged into *cache group federations*, which opens a number of research questions. As a first requirement, we must apply our design restrictions to the cache group federation as a whole. When unsafe situations occur (e. g., a heterogeneous cycle evolves upon federating a cache group) we must modify and refine our design, which leads to modifications of the individual cache groups. For example, unsafe cycles would have to be broken up by removing RCCs from some of the participating cache groups. Although performance surprises in the form of recursive loading operations will be prevented if we enforce our design restrictions, the individual cache groups in the federation may strongly influence the loading behavior of each other.

Example 15 Figure 13 shows two cache groups G_5 and G_6, which have been derived from G_3 by slight modifications. $P.pid$ is the only filling column of G_6 and table L is shared by both cache groups. Apparently, the induced completeness of $L.oid$ in G_5 and of $L.pid$ in G_6 (when viewed separately) has disappeared due to the shared representation of table L.

In this case, the federation F may provide some (small) gains as compared to the individual cache groups G_5 and G_6.

[12] If a problem has no solution, it may not be a problem but a fact, not to be solved but to be coped with over time. (Shimon Perez)

If the record sets of G_5 and G_6 overlap on table L, some storage space is saved due to shared records. Because the RCCs of both cache groups do not interfere, G_5 and G_6 remain separated to a large degree.

This is no longer true if we replace G_5 by G_3. In this case, we obtain a homogeneous RCC cycle $L.pid \rightarrow P.pid \rightarrow L.pid$, which is "activated" by insertions in both G_3 and G_6. Although evaluation correctness is not challenged and cycle-induced loading always stops in the first round, the RCC cycle seems harmful in many situations. Technically the cycle has two initiating columns that pump a lot of records—unwanted in both G_3 and G_6—into the cache group federation.

Hence, in the general case, the penalty each group in a federation must pay as "membership fees" is often much larger than the gain due to record sharing. In practical situations, cost models have to be provided to decide whether the management of multiple cache groups is more beneficial in the form of cache group federations or separate caches.

7 Related work

7.1 TimesTen

In our terms, TimesTen cache groups [20] consist of a root table and a number of member tables connected via member constraints (corresponding to PK/FK relationships in the backend DB). A TimesTen cache instance is a collection of related records (also denoted as a complex object) that is uniquely identifiable via a cache instance key. For this purpose, the root table carries a single identifier column (U) whose values represent cache instance keys.

Because all records of cache instances must be uniquely identifiable, they form non-overlapping tree structures (or simple disjoint directed acyclic graphs) where the records embody the nodes and the edges are represented by PK/FK value-based relationships. Individual cache instances represent the units of loading, aging, and replacement.

Note, there is no equivalence to our notion of filling columns (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 cache instances ("all at once", "by id", "by where clause": specified independently from a cache group). There is no automatic control of value completeness or, as a consequence, of completeness of predicate extensions. When declarative, set-oriented query processing is performed in the cache, the user has to guarantee for them. Prevalent operations seem to be based on navigation, where the application controls loading and unloading of cache instances. Both operations are (at least conceptually) very simple, because they deal with disjoint collections of records. Furthermore, we adhere to a value-based table model—separate from the underlying relational model—, which does not require the matching of RCCs and backend schema constructs (e. g., PK/FK pairs). In summary, the TimesTen solution can be considered a simple special case of our approach[13].

7.2 DBCache

Cache groups in the DBCache project have their origin in the TimesTen approach but are substantially enhanced towards cache-supported declarative query processing. The approach taken by DBCache rests on the concepts of *domain completeness*, *cache keys*, *RCCs*, and *cache groups* [2] (some of which were co-defined by one of the authors while he was participating in the DBCache project).

Domain completeness and cache keys are not orthogonal concepts and are responsible for the limitation and some undesired behavior of cache groups. A cache key column is domain complete per se, which means that there is no way to prevent cache loading for any value (even smuggled)in such a column, although only loading of values that promise "locality of reference" is the ultimate goal of caching. Hence, this approach does not enable controlled cache loading. On the other hand, for cache safeness one needs to observe the cache key rule that at most one cache key column may be of type NU.

In our approach, the concepts *candidate value* and *control column* (borrowed from the MTCache project) together with the pivotal term *value completeness* enable orthogonal specification of what is to be cached and what is guaranteed to be value complete. The concept *cache key* and the cache key rule are not needed anymore. Domain completeness (our term: column completeness) is reduced to an option for a simple specification of candidate values and for rapid probing. The introduction of control columns for target columns of RCCs and for filling columns extended the use of RCCs and enabled a more powerful and flexible probing mechanism; a direct consequence is the option of utilizing negative caching.

The cache groups of DBCache are implicitly constructed using cache keys and RCCs (bottom-up). In contrast, we specify cache groups declaratively starting with the predicate type that is to be supported (top-down). We start with the concept *value completeness* and can constructively generalize our approach to predicate completeness for classes of PSJ predicates. By rewriting the definition of RCCs in terms of value completeness, we discovered the mistake in the proof "domain completeness in homogeneous cycles" [2, Sect. 3.4.2]. In contrast to DBCache, we have defined all issues related to cycles more precisely and have explored their properties in detail (centered around the concept *smuggler relationship*). Furthermore, this refined consideration revealed the existence of *optimization RCCs* and also *path-dependent RCCs*.

[13] Note, we do not assess its practical usefulness nor its inherent cache performance.

For convenience, we still use the two terms cache group and RCC, although essential caching operations (controlled loading, selective unloading, probing via an RCC's source column, use of negative caching) are different from the DBCache approach. As a major advantage of our approach, we regard cache groups as a unit of design and can explicitly control their loading behavior (in our case, a cache group does not grow if another set of RCCs (from another cache group) makes more cache tables reachable from the root table). Unexpected growing of such reachability graphs is a problem of its own when constructing federations from cache groups.

8 Conclusions and outlook

We have explored cache group design in full depth and have derived two rules for safeness as well as simple conditions for identifying complete columns. Flexible probing strategies enable us to recognize single complete values in otherwise non-complete columns. With these rules and strategies, cache groups guarantee correct evaluation of specific PSJ queries and safe loading of the related predicate extensions whenever new instances are demanded. Furthermore, we have shown that optimization RCCs can be exploited to improve the flexibility and usability of cache groups at no additional cost.

Our first attempts to design cache groups were based on something like a bottom-up approach: We started with tables and RCCs and then added filling columns. In doing so, we experienced many surprises when populating the cache and essentially did not know what was influenced by what. The decisive step that facilitated the understanding of cache groups was to apply a top-down approach. Using this insight, we arrived at the concept of cache group federations and could easily explain the effects of different predicate types on cache groups and their unwanted mutual augmentation of cache table loading. Considering the cache group approach in this way, it becomes clear that mutual RCC dependencies quickly lead to large cache table populations. Hence, if a number of different overlapping predicate types should be supported by a cache, it is often better to provide the complete table contents in the cache (full-table caching).

We are only at the beginning of a promising research area concerning constraint-based and adaptive DB caching. Hence, a number of important issues remains to be solved or explored.

8.1 Updates and adaptation

Other interesting research problems occur if we apply different update models to DB caching. Instead of processing all (transactional) updates in the backend DB first, one could perform them in the cache (under ACID protection) or even jointly in cache and backend DB under a 2PC protocol. Such update models may lead to futuristic considerations where the conventional hierarchic arrangement of frontend cache and backend 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.

Improving the adaptability is another important problem. How can constraint-based approaches evolve with changing locality patterns of the workload? To support frequently requested join operations by additional RCCs or to remove RCCs not exploited anymore needs adaptive cache group specifications!

Hence, for each variation of constraint-based caching, quantitative analyses must help to understand which cache configurations are worth the effort. For this purpose, a cache group advisor can be designed to support the DBA in the specification of a cache group when the characteristics of the workload are known. Here, the expected costs for cache maintenance and the savings gained by predicate evaluation in the cache can be determined thereby identifying the tradeoff 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/or tables until the optimum point of operation is reached. On the other hand, such a tool may 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.

8.2 Other predicate types

So far, predicate completeness can be achieved for equality and range predicates combined with equi-join predicates. Varying the fundamental idea of cache groups, we can explore the probing and completeness conditions of other types of predicates, for example, aggregate predicates or, for specialized use, recursion predicates. Of course, all these ideas of constraint-based caching are not restricted to the relational model. They may be applied equally well to other data models and the caching needs of their applications, for example, to XML documents and XQuery operations.

Acknowledgements We would like to thank M. Altinel, Ch. Bornhövd, and C. Mohan for many fruitful discussions while Theo Härder spent his sabbatical at the Almaden Research Center of IBM.

A Proof

Let us show that our design rules prevent dangerous cache group federations (and dangerous cache groups as a special case, of course). Moreover, we are going to show that our rules are precise (i. e., that there is not any safe cache group federation that our rules consider unsafe).

Theorem 1 *A cache group federation F is safe if and only if Rules 1 and 2 are followed.*

We proceed by dividing the space of all possible cache group federations into disjoint classes (discussed in five cases below) for each of which we will determine whether our rules match and analyze whether the cache groups in this class are safe. If this analysis and our rules agree in the question of safeness in every case, our proof is complete.

During the proof we use sketches of cache group configurations for illustrations: Circle and arcs drawn with a straight line indicate homogeneous RCC cycles or parts thereof. The familiar zigzag arrows represent smuggler relationships. The columns of cache tables that are involved are not shown explicitly: Of course, at each end of a smuggler relationship there *is* a column (both in the same table); a homogeneous cycle may traverse any number of columns. Some columns may be highlighted by a black dot in order to draw attention to them.

Let us begin with the simple cases. Each of the subsequent cases is more complex and implicitly excludes the cache group federations discussed so far.

Case 1 There are no cycles in F.

Cache loading follows RCCs (only): Since every path of loading steps through F visits each RCC at most once (due to the absence of cycles), no recursive loading is possible and F is safe. Our rules consider F safe as well, because they only deal with cycles.

For the following cases, we can assume that there is at least one cycle in F.

Case 2 Every cycle in F is homogeneous.

In this case, any two of the homogeneous cycles in F cannot be coupled by a pair of smuggler relationships: Suppose there were such cycles C_1 and C_2,

then there would also be a heterogeneous cycle C_3, which contradicts our main assumption of this Case 2. So each homogeneous cycle is either isolated or only connected to other homogenous cycles directly on the same column.

If a new value v appears in a column c of such a cycle, the loading proceeds along the cycle (and probably other connected homogeneous cycles) with only this value v. As soon as the cycle has been traversed once and the loading process has reached c again—given that v appears in every (backend) column of the cycle—, no further loading steps are necessary, since v has already been made complete in the successor of c. If v does not appear in every column, the loading even stops earlier. Therefore no recursive loading is possible and F is safe.

Since there are no heterogenous cycles, our rules do not apply and thus consider F safe, too.

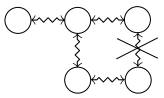
Here we are left with cache groups that have at least one heterogenous cycle.

Case 3 There is a cycle C in F with a smuggler relationship $R = T.a \rightsquigarrow T.b$ that does not pair up.

"R does not pair up" means that the inverse smuggler relationship $R' = T.b \rightsquigarrow T.a$ is not part of C. In this case, there must be an atomic heterogeneous subcycle C' of C that contains R. This cycle C' is dangerous by the same arguments given in Sect. 4.1.2: R smuggles new values into the cycle that have no chance of being compensated because R' is missing. Therefore, C enables recursive loading in the cache group federation, which makes F unsafe.

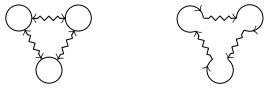
Our Rule 1 applies only if C is isolated, in which case F would be considered unsafe. Rule 2 does apply in any case, because R is a non-compensating smuggler relationship. Federation F is thus deemed unsafe, which matches the analysis above.

In the remaining cache group federations F, all smuggler relationships appear in pairs in every cycle. This means that F basically consists of homogeneous cycles coupled by pairs of smuggler relationships. (Of course, there can be non-cyclic parts leading away from this arrangement of cycles; but they cannot make the federation unsafe in analogy to Case 1.) Here is an example:

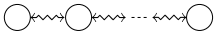


Isolated homogeneous cycles are safe. To explore the interaction between them, we regard the coupled homogeneous cycles as nodes in a more abstract graph representation: The undirected graph M (for *meta*-graph) comprises all homogeneous cycles as nodes and all pairs of smuggler relationships (in cycles) as edges. Paths and cycles in this graph are called *meta-paths* and *meta-cycles* accordingly.

The federations F in this Case 3 have another important property: There are no meta-cycles in F (as indicated by the crossed-out edge above): The meta-graph M is acyclic, which means it must be tree-structured. Otherwise F would contain a heterogeneous cycle with smuggler relationships that do not pair up—a contradiction, as illustrated by the following meta-cycle with three pairs of smuggler relationships:

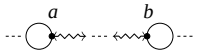


Let us now consider linear sequences of coupled homogeneous cycles in F (i. e., paths in M or *meta-paths* in F):



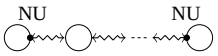
Obviously, if one such meta-path enables recursive loading, F is unsafe. This also holds the other way round: If F is unsafe, there must be at least one such meta-path responsible for this. If all meta-paths are safe, F is, too. This means that all we need to analyze is the different kinds of meta-paths.

The relevant columns in a meta-path are the ones participating in the smuggler relationships (all other ones are only part of homogeneous paths). We say that two such columns a and b *face* each other if they are arranged on opposite outer ends of smuggler relationships, as shown here:



Finally, we can distinguish two cases of facing columns.

Case 4 There is a meta-path in F that contains two NU columns facing each other.



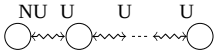
Each NU column $T.c$ shows the following behavior in combination with its homogenous cycle: If a new value v appears in $T.c$, the homogeneous cycle may (given suitable backend contents) lead to value completeness of v in the NU column $T.c$. This means that new records are inserted into T, which may smuggle new values into the remaining columns of T. These new values force loading steps in the meta-path up to the facing NU column, where the behavior just explained starts anew.

Therefore, these two remotely coupled homogenous cycles with facing NU columns may activate each other recursively, and the federation F is unsafe.

It is easily checked that at least one of the two outermost pairs of smuggler relationships is not compensating—no matter which point of view in the overall cycle is chosen: For a pair $a \rightsquigarrow b$ to be compensating, it is necessary that b is a U column. Here at least one of the two NU columns appears in the role of b. This means that Rule 2 considers F unsafe, too.

Case 5 In every meta-path in F, there are no NU columns facing each other.

This means that either there are not any NU columns among the facing columns of the meta-path or there is a configuration like the following (as a part of the meta-path viewed from an arbitrary NU column):



Here all pairs of smuggler relationships are compensating (when viewed from the left): The rightmost one is compensating because it has no smuggler relationships inbetween and the farther column is U. Every other pair is compensating because the farther column is U and all pairs to the right are compensating. This means that none of our rules applies and the federation F is considered safe. The same arguments hold in the above-mentioned case that there are not any NU columns.

All that is left to show is that this kind of meta-path does not enable recursive loading: Imagine a new value v inserted into the NU column at the left-hand side. At first, this meta-path shows the same behavior as sketched in Case 4: Value v may get value complete due to the homogeneous cycle; new records may be inserted because of the NU column, which may smuggle new values into the other columns, which again may force loading steps along the meta-path. But when this process reaches the right end, it stops because for each value inserted into the facing U column there is only a single record that has already been inserted (otherwise the value would not appear). Therefore the rightmost homogeneous cycle has no further effect on the meta-path; only in the homogeneous cycle itself the last loading steps will occur. Therefore, this kind of meta-path and the federation F are safe.

As we have seen, our rules agree in every case with the result of the safeness analysis. That concludes our proof. □

References

1. Akamai Technologies Inc (2004) Akamai EdgeSuite. URL http://www.akamai.com/en/html/services/edgesuite.html
2. Altinel M, Bornhövd C, Krishnamurthy S, Mohan C, Pirahesh H, Reinwald B (2003) Cache tables: Paving the way for an adaptive database cache. In: VLDB Conference, pp 718–729
3. Amiri K, Park S, Tewari R, Padmanabhan S (2003) DBProxy: A dynamic data cache for web applications. In: ICDE Conference, pp 821–831
4. Andrews M (1998) Negative caching of DNS queries (DNS NCACHE). Request for Comments (RFC) 2308, URL ftp://ftp.rfc-editor.org/in-notes/rfc2308.txt
5. Anton J, Jacobs L, Liu X, Parker J, Zeng Z, Zhong T (2002) Web caching for database applications with Oracle Web Cache. In: SIGMOD Conference, pp 594–599
6. Bello RG, Dias K, Downing A, Feenan JJ Jr, Finnerty JL, Norcott WD, Sun H, Witkowski A, Ziauddin M (1998) Materialized views in Oracle. In: VLDB Conference, pp 659–664
7. Bühmann A (2005) Einen Schritt zurück zum negativen Datenbank-Caching (A step back towards negative database caching). In: BTW Conference, Karlsruhe, pp 107–124
8. Dar S, Franklin MJ, Jónsson B, Srivastava D, Tan M (1996) Semantic data caching and replacement. In: VLDB Conference, Morgan Kaufmann, pp 330–341
9. Goldstein J, Larson P (2001) Using materialized views: A practical, scalable solution. In: SIGMOD Conference, pp 331–342
10. Härder T, Bühmann A (2004) Query processing in constraint-based database caches. Data Engineering Bulletin 27(2):3–10
11. IBM (2004) IBM Cloudscape. URL http://www.ibm.com/software/data/cloudscape/
12. IBM (2004) IBM DB2 Universal Database (V 8.1). URL http://www.ibm.com/software/data/db2/
13. Keller A, Basu J (1996) A predicate-based caching scheme for client-server database architectures. VLDB Journal 5(1):35–47
14. Larson P, Goldstein J, Guo H, Zhou J (2004) MTCache: Mid-tier database caching for SQL server. Data Engineering Bulletin 27(2):35–40
15. Larson P, Goldstein J, Zhou J (2004) MTCache: Transparent mid-tier database caching in SQL server. In: ICDE Conference, IEEE Computer Society, pp 177–189
16. Levy AY, Mendelzon AO, Sagiv Y, Srivastava D (1995) Answering queries using views. In: PODS Conference, pp 95–104
17. Luo Q, Naughton JF (2001) Form-based proxy caching for database-backed web sites. In: VLDB Conference, pp 191–200
18. Oracle Corporation (2004) Internet application server documentation library. URL http://otn.oracle.com/documentation/appserver10g.html
19. Podlipinig S, Böszörmenyi L (2003) A survey of web cache replacement strategies. ACM Computing Surveys 35(4):374–398
20. The TimesTen Team (2002) Mid-tier caching: The TimesTen approach. In: SIGMOD Conference, pp 588–593
21. Zhou J, Larson P, Goldstein J (2005) Partially materialized views. Tech. Rep. MSR-TR-2005-77, Microsoft Research, URL ftp://ftp.research.microsoft.com/pub/tr/TR-2005-77.pdf