Caching over the Entire User-to-Data Path in the Internet

Theo Härder

University of Kaiserslautern, Germany
haerder@informatik.uni-kl.de

Abstract. A Web client request traverses four types of Web caches, before the Web server as the origin of the requested document is reached. This client-to-server path is continued to the backend DB server if timely and transaction-consistent data is needed to generate the document. Web caching typically supports access to single Web objects kept ready somewhere in caches up to the server, whereas database caching, applied in the remaining path to the DB data, allows declarative query processing in the cache. Optimization issues in Web caches concern management of documents decomposed into templates and fragments to support dynamic Web documents with reduced network bandwidth usage and server interaction. When fragment-enabled caching of fine-grained objects can be performed in proxy caches close to the client, user-perceived delays may become minimal. On the other hand, database caching uses a full-fledged DBMS as cache manager to adaptively maintain sets of records from a remote database and to evaluate queries on them. Using so-called cache groups, we introduce the new concept of constraint-based database caching. These cache groups are constructed from parameterized cache constraints, and their use is based on the key concepts of value completeness and predicate completeness. We show how cache constraints affect the correctness of query evaluations in the cache and which optimizations they allow. Cache groups supporting practical applications must exhibit controllable load behavior for which we identify necessary conditions. Finally, we comment on future research problems.

1 Motivation

Internet-based information systems and e*-applications are growing with increasing pace and their users are placing tremendous workloads with critical response-time restrictions on the Internet and the Web servers. For these reasons, scalability, performance—in particular, minimization of user-perceived delays—, and availability are prime objectives for their system development. Most of all, various forms of caching in the Web have proven to be a valuable technique[1] towards these design goals. Three aspects make caching attractive in the Web environment, because it effectively reduces network bandwidth usage, user-perceived latency, and workload on the origin server.

To improve response time and scalability of the applications as well as to minimize communication delays in wide-area networks, a broad spectrum of techniques has emerged in recent years to keep static Web objects (like HTML pages, XML fragments, or images) in caches in the client-to-server path. These techniques, often summarized as *Web caching*, typically support access by object identifiers and aim at locating and possibly assembling user-requested Web objects in caches near the Web client to unburden the Web traffic and to achieve minimal response times. In particular for static Web objects, it can provide various kinds of performance improvements for e*-applications [26]—a reason which amplified the setup of Web caches and the optimization of their usage by tailored replacement strategies [23] in recent years. Nowadays, however, more and more dynamically generated content is needed and offered making Web applications even more attractive and enabling new forms of business: contents' personalization, goal-oriented advertisement, interactive e-commerce, one-to-one marketing, and so on. Obviously, the way caching is performed has to respond to these new requirements. To effectively serve this trend, caches have to be aware of the internal structure of documents (Web pages) to enable selective reuse of static fragments (objects) and exchange of dynamic parts in order to assemble them to the actual document to be delivered in the most cost-effective way. Fragment-enabled caching techniques have to be developed which can distinguish and manage templates and fragments of Web documents separately.

2 The Client-to-Server Path

Conceptually, a Web request is processed as follows: A Web client (client throughout the paper) sends a query containing a URL via HTTP and the Internet to a Web server (origin server or server, for short) identified by the URL. The server processes the request, generates the answer (typically an HTML or XML document), and sends it back to the client. To solve the performance and availability problems sketched above, we add, again conceptually, a Web proxy server somewhere in the client-to-server path. Such a proxy can be used in a number of ways, including

– caching documents and parts thereof
– converting data to HTML/XML format so it is readable by a client browser
– providing Internet access for companies using private networks
– selectively controlling access to the Internet based on the submitted URL
– permitting and restricting client access to the Internet based on the client IP address

In this contribution, we concentrate on the caching functionality of proxies and discuss the client-to-server path how it evolved during the recent past. Caches, in general, store

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

frequently accessed content and locally answer successive requests for the same content thereby eliminating repetitive transmission of identical content over network links. Thus, the complete caching solution comprises a networking component and a cache component which work together to localize traffic patterns: A user requests a Web page from a browser. The network analyzes the request and, based on certain parameters, transparently redirects it to a local cache in the network. If the cache does not contain the Web page, it will make its own request to the origin server, which then delivers the content to the cache, which, in turn, delivers the content to the client while saving the content in its local storage, that is, caching the content. Subsequent requests for the same Web page are analyzed by the network and, based on certain parameters, transparently redirected to the local cache.

This process may substantially reduce network traffic and latency time for the client. Based on their use and behavior, we can distinguish four types of caches:

- *Browser cache*: For all user requests, this cache dedicated to the browser is first searched. If the specific content is located in the cache, it is checked to make sure that it is "fresh". Such a private cache is particularly useful if a user scrolls back in his request history or clicks a link to a page previously looked at.
- *Proxy cache*
  While working on the same principle, but at a much larger scale, such a cache is shared, performs demand-driven *pull caching*, and serves hundreds or thousands of users in the same way. It can be set up on the firewall or as a stand-alone device. Unless other search paths are specified, a cache miss sends the request to the next proxy cache in the client-to-server path.
- *Reverse proxy cache*
  This kind of cache is an intermediary also known as "edge cache", "surrogate cache", or "gateway cache". While not demand-driven, such caches reverse their role as compared to proxy caches, because they are supplied by origin servers with their most recent offerings—a kind of *push caching*. Furthermore, they are not deployed by network administrators to save bandwidth and to reduce user-perceived delays which are characteristic for proxy caches, but they are typically deployed by Web masters themselves, to unburden the origin servers and to make their Web sites more scalable, reliable, and better performing.
- *Server cache*
  It keeps generated content and enables reuse without interaction of the origin server. Intermediate results and deliverable Web documents help to reduce the server load and improve server scalability.

With these definitions, we are able to explain how a Web request is processed in detail, as illustrated in Fig. 1. Note, we pursue a functional view and focus on the client-request paths of a single ISP (Internet service provider) to servers connected to the Internet via another ISP.

Each of the ISPs is able to connect a set of clients and servers to a *wide-area network*. C_i and S_j represent clients and origin servers, respectively[2]. BC_i refers to the browser cache of C_i, whereas P_{TL} (top-level), P_n, P_{nm}, ... identify proxy caches typically

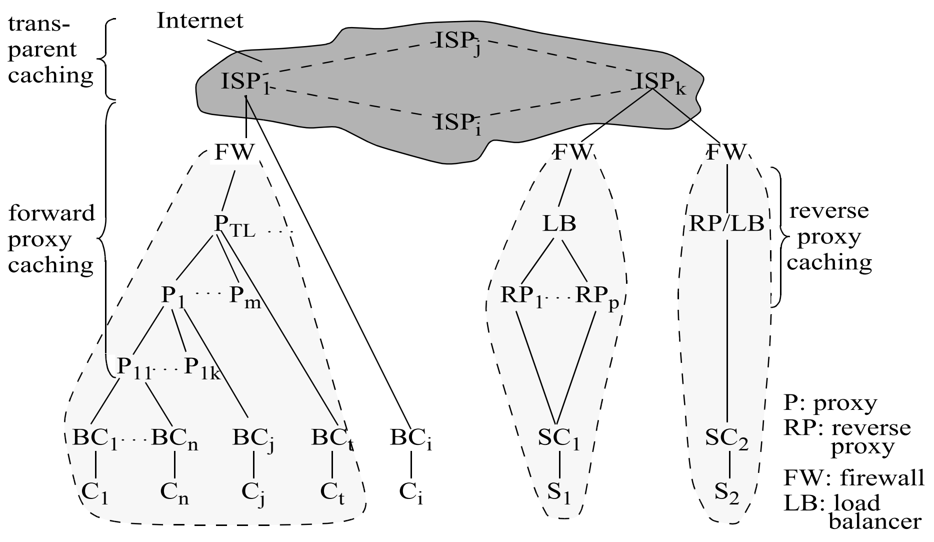


Fig. 1 The Client-to-Server Path through the Internet

allocated in *local-area networks* (LANs) and organized as a multi-level hierarchy in the domain of an enterprise (Intranet). Usually, a stand-alone firewall device (FW) separates the Intranet from its ISP and the Internet; for private clients, the firewall is often combined with the browser cache. An ISP plays two roles: it provides access to the Internet for its clients and usually manages the top-most proxy cache for them. On the other hand, it is a transit provider for routing requests through the Internet. If an ISP offers caching services for routing requests, it is sometimes denoted as transparent caching[3] which is much less effective, because routing may use multiple communication links to the origin server (from ISP_1 to ISP_k in Fig. 1) and select a different path on each re-request. Hence, client-side proxy caching is most effective in the invocation paths from BC up to the corresponding ISP (also denoted as *forward proxy caching*).

In contrast, the caching mechanisms at the server side are primarily directed towards server scalability and overload protection. Requests can be routed to reverse proxies by a number of methods; two of them are indicated in Fig. 1, where some form of load balancing (LB) is involved. This mechanism makes the reverse proxy caching look like the origin server to clients. The incoming requests are either distributed via LB to the reverse proxies (RP_i) or directly to an RP with built-in LB which, on an RP cache miss, forward them to the server cache (SC_j) or a specific server processor, when the server itself is embodied by a processor complex.

2. Using port 80, it is always possible to choose a direct communication link from C to S if the IP address of the origin server is known.

3. While usually the use of a proxy server must be explicitly disclosed to its clients, that of a transparent proxy must not. Hence, caching of such a proxy server remains transparent to the clients.

Another form of reverse proxy caches or edge caches is used in so-called content delivery networks (CDNs) where such caches are distributed throughout the Internet or a part of it. Dynamic content is supplied, if possible, in edge servers. Otherwise, when data is missing or when stale content does not comply with the clients' consistency requirements, the requests are forwarded to the Web server. These edge servers are considered as an extension of the trusted Web server environment, because they are either within the server's administrative domain or within a CDN contracting with the content provider. Enterprise software and CDN solutions like EdgeSuite (Akamai) and WebSphere Edge Server (IBM) are offloading to edge servers certain applications components (such as servlets, JSPs, Enterprise Beans, and page assembly) which usually run at the Web server. For example, Akamai's CDN currently contains up to 15.000 edge servers [1]. A CDN provider sells caching as a service to interested Web sites and guarantees availability of contents on all important Internet nodes for their customers. In this way, customers such as Amazon or MSNBC reach high availability even under extreme load situations without scaling their own Web servers.

3 Web Caching

An important practical consideration concerns the validity of Web objects, particularly in client-side caches. For this reason, the so-called time-to-live algorithm (TTL) is used to determine whether or not a Web object present in a cache can be used to satisfy a client request. Hence, only valid objects can be delivered to the client and are, therefore, kept in the cache. If it runs out of space anyway, some algorithm has to be used to make room for the objects of the current client requests.

3.1 Replacement Strategies

As compared to DB buffer management, which typically provides fixed-length frames and applies LRU- or LRD-based replacement algorithms, Web caching is much more complex. Web objects need variable-length frames and are characterized by more factors that critically influence the replacement decision. To indicate their spectrum, we include the list of important factors from [18]:

- time of the last reference to the object (recency)
- number of requests to an object while in the cache (frequency)
- size of the Web object (size)
- cost to fetch an object from its origin server (cost)
- time of last modification, time when an objects gets stale (expiration time).

The influence or interdependencies of these factors cannot be discussed in detail. We can only summarize the resulting cache replacement strategies which typically exploit the first four factors above. A suitable classification of them was given in [16] and surveyed in [23]:

- *Recency-based strategies* incorporate recency (and size and/or cost) into the replacement process.
- *Frequency-based strategies* exploit frequency information (and size and/or cost) in the replacement decision.
- *Recency/frequency-based strategies* consider both recency and frequency under fixed or variable cost/size assumptions.

3.2 Validity of Cached Objects

A Web cache must be able to locally determine the validity or freshness of its objects. For this reason, the cache is equipped with some checking rules and each object carries a number of parameters in its HTTP header. Some simple ground rules together with object-related parameters allow rapid checking of the object's validity.

The *Expires* HTTP header is the basic means of controlling caches. It tells the cache how long the object is fresh for; after that time, the cache will always check back with the origin server to see if a document is changed. Most Web servers provide a number of ways to set *Expires* response headers. Commonly, they will allow setting an absolute time to expire (e.g., Expires: Mon, 4 April 2005 13:49:31 GMT), a time based on the last time that the client saw the object (e.g., Last-Accessed: Fri, 8 April 2005 23:07:18 GMT), or a time based on the last time the document changed on your server (e.g., Last-Modified: 8 April 2005 21:27:28 GMT). If no *Expires* value as the definite time limit is set (for so-called ZeroTTL objects), the cache may estimate the freshness via *Last-Accessed* or *Last-Modified*. If these values are also undefined, caching of this object is usually not possible.

Although the *Expire* mechanism is useful, it is still somewhat limited. In quite a number of cases, content is cacheable, but the protocol lacks methods to tell the caches how to handle such objects. Some experimental studies have shown that a considerable portion of uncacheable HTTP content is actually cacheable [28]. To improve this situation, HTTP 1.1 introduces enhanced and more flexible object control via the *Cache-Control* response headers which allow Web masters to define how pages should be handled by caches. They include directives to specify what is cacheable, what may be stored, how to modify the expiration mechanism, as well as how to revalidate or reload objects. Useful *Cache-Control* response headers include:

- *max-age=[seconds]*—specifies the maximum amount of time that an object will be considered fresh. Similar to Expires, this directive allows more flexibility.
- *public*—marks the response as cacheable, even if it would normally be uncacheable, e.g., if the object is authenticated, the public directive makes it cacheable.
- *no-cache*—forces caches (both proxy and browser) every time to submit the request to the origin server for validation before releasing a cached copy. This is useful to assure that authentication is respected (together with public), or to maintain rigid object freshness, without sacrificing all of the benefits of caching.

- *must-revalidate*—tells the cache that it must obey any freshness information for the object. This header forces the cache to strictly follow the given rules.

In addition, the checking times of caches, that is, when they control their objects' validities, can be configured, e.g., once per session or time unit. Hence, simple cache-related rules together with object-related parameters determine the freshness semantics of cached objects and guarantee rapid local checking of an object's validity.

3.3 Dynamic Content

So far, our discussion primarily considered static Web objects, typically Web pages containing static HTML/XML data, whose source code is stored at the Web server. In such cases, a page miss in all proxy caches causes the delivery of a fresh page from the Web server.

In the current Internet, however, interactive pages of online shops, member logins of community pages, etc. play a performance-critical role. All of them contain static fragments which have long validity periods and may be shared by many clients accessing essentially the same page in some personalized appearance. On the other hand, some of their fragments are highly dynamic, can only be shared by a few clients or not at all, and must be re-generated almost upon each reference. There are some reasons not discussed here, why proxy caches are of limited use today when dynamic content is frequently needed. However, there are already useful caching mechanisms available at the server side which help to satisfy the requirements and optimize the run-time behavior of the "client-server" loop. In principle, these mechanisms can be refined such that they are also applicable at the client side in the near future.

In cases where a few and small content fragments exhibit high update frequencies, the concept of edge-side includes (ESI) is particularly helpful. Dynamic Web pages are not handled as units anymore, instead fragment-enabled caching allows the management of such pages at a finer level of granularity. Dynamic pages (documents) are decomposed into a template and several fragments for which separate TTL values and URLs can be specified. The template describes the layout of the Web page and specifies the location of each content fragment belonging to it.

For this purpose, the ESI concept—proposed by W3C in summer 2001 and currently the de facto standard for templates in fragment-based documents [27]—offers XML-based language constructs which enable the composition of dynamic Web pages and a fragmentation of their content. Actually, the markup format specifies content fragments for inclusion and assembly in a base template. ESI also includes a complete framework for conditional fetching of fragments, cookie support, and error control. As a consequence, separation of content generation and content provision becomes possible which greatly increases Web server scalability.

As illustrated in Fig. 2, such a fragmentation assigns separate TTL values to the fragments of template-based page, all of them identified by separate URIs. Hence, these fragments can be selectively exchanged (if expired) or flexibly composed to new pages (shared use). If a cache is equipped with the assembly and exchange logic of such tem-

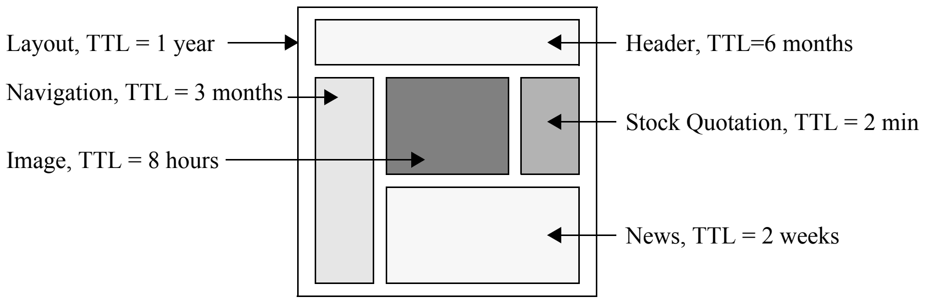


Fig. 2 Fragmentation of a Dynamic Web Document using ESI

plate- and fragment-based pages, much of the extra burden of dynamic page management can be assigned to the caches in the client-to-server path.

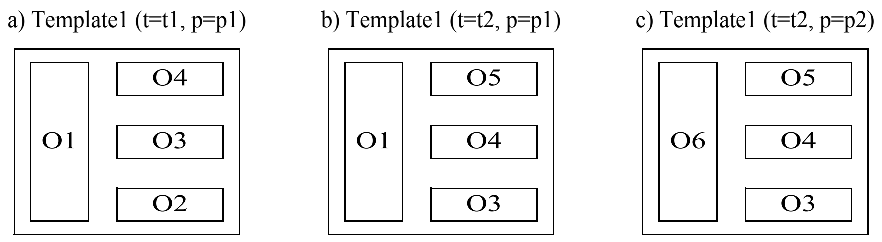
Today, ESI is primarily used at the server side, in particular by the edge caches of CDNs. In addition to Web server scalability, its use will gain more and more benefits also at the client side when it conquers the proxy caches or even the browser caches thereby reducing user-perceived delays and network bandwidth usage.

3.4 Future Fragment-Enabled Caching

As sketched above, ESI concepts help to distinguish and manage templates and fragments of Web documents separately. The resulting fragment-enabled caching techniques have to be refined to maximize flexibility of dynamic document assembly and minimize its underlying overhead. Therefore, fine-grained fragment identification is desirable independently of their location on the template, which makes various kinds of fragment behavior possible, for example, fragment movement in a newly assembled document.

Fragment-based documents need efficient and fine-granular update methods, even if the fragment locations change or if some fragments serve personalization purposes. Fragment movements frequently occur, as illustrated in Fig. 3a and b, when in a Web document containing news, actual stories added on the top are pushing the older ones down in the document. A similar situation occurs if personalized pages use the same template, but differ in a few fragments which carry, for example, the salutatory address or some items preferred by the client. Technically, each esi:include tag references a specific URI with a TTL, which is stored in the template file. Hence, all the information related to the template and its fragments is actually present in the template itself. Fragment movement in the current ESI concept can be performed as follows: One option is to update all fragments where the object moved to where it moved from, which seems expensive. Alternatively, the template can be updated such that it contains the URLs of the new locations pointing to the correct fragments [25].

There is no effective and efficient way to solve these problems in the current ESI infrastructure. Neither invalidation of many objects that are actually valid nor invalida-


a) Template1 (t=t1, p=p1)
O1
O4
O3
O2
b) Template1 (t=t2, p=p1)
O1
O5
O4
O3
c) Template1 (t=t2, p=p2)
O6
O5
O4
O3

```
template {
<esi:include src=o1>
<esi:include src=o4/>
<esi:include src=o3/>
<esi:include src=o2/>
}
```

```
template {
<esi:include src=o1>
<esi:include src=o5/>
<esi:include src=o4/>
<esi:include src=o3/>
}
```

```
template {
<esi:include src=o6>
<esi:include src=o5/>
<esi:include src=o4/>
<esi:include src=o3/>
}
```

Fig. 3 Personalization and Movement of Fragments

tion of the template which typically needs very little data modification seem appropriate. A new solution is proposed by the DyCA (dynamic content adaptor) approach described in [7]. The essential idea is to extract the objects from the original content thereby achieving the needed separation between template, objects, and object location. DyCA uses a *mapping table* which introduces some kind of reference indirection and, in turn, does not require the template and the objects to be invalidated for spatial changes. Even personalization seems possible with dedicated mapping tables by reusing the original template and all common fragments (Fig. 3c).

The cache holds the mapping table together with the template. When a dynamic document is assembled in the cache, the identifiers of its fragments are looked up in the mapping table. If a fragment is located in the cache and satisfies the TTL or other validity constraints, it is directly integrated into the document; otherwise, a fresh version is fetched from the appropriate URL. Hence, object movement only requires an update of the small-sized mapping table and personalization of documents can be handled by using dedicated mapping tables.

In a recent empirical exploration [7], the update costs of dynamic documents were evaluated in detail. Tab. 1 copies some indicative numbers gained from this experiment referring to large existing Web sites. The update problem caused by object movement (e.g., by a news ticker) was quantified by simulating four different approaches:

– no fragment caching
– static template, fragment updates due to data and spatial changes
– template updates, static objects (fragment updates only to due data changes)
– use of a mapping table.

The evaluation provides interesting empirical data for the “total data transfer between server and cache” and “user-perceived latency”. In general, both “template updates, static objects” and “mapping table” clearly outperform the other approaches.

Tab. 1 Comparison of Template and Object Sizes

	NY Times	**India Times**	**Slashdot**
Template Size	17 KB	15 KB	1.7 KB
Avg. Object Size	3.6 KB	4.8 KB	0.6 KB
Mapping Table Size	1.0 KB	0.8 KB	2.2 KB

So far, we have discussed fragment-enabled Web caching which allows spatial movement of objects in and personalization of dynamic documents thereby supporting all caching objectives sketched in the introductory section. If the content is delivered as textual sources by information providers, for example, via news tickers, it is edited and formatted as XML or HTML fragments by the origin server and distributed to the clients (either by pull or push mechanisms). If, however, the dynamic content has to be queried and fetched by the Web server from continuously changing data in a possibly remote database, transactional programs (application logic) have to be provided to the server to evaluate the DB queries and to deliver transaction-consistent query results. This kind of content provision and its consistency requirements may introduce another bottleneck into the now prolongated client-to-server path which leads to the point where DB caching comes into play.

4 The User-to-Data Path

As transactional Web applications (TWAs) must deliver more and more dynamic content and often updated information, Web caching should be complemented by techniques that are aware of the consistency and completeness requirements of cached data (whose source is dynamically changed in backend databases) and that, at the same time, adaptively respond to changing workloads. Because the provision of transaction-consistent and timely data is now a major concern, optimization of Web applications has to consider the entire user-to-data path. Because the essential caching issues in the path up to the Web server are already addressed in sufficient detail, we target at specific problems on the remaining path towards DB-managed data.

Several different solutions, summarized as *database caching,* have been proposed in recent years [2, 3, 5]. Fig. 4 focuses on the realm of DB caching and complements the Big Picture of Web caching shown in Fig. 1. For this relatively new problem, currently many DB vendors are developing prototype systems or are just extending their current products [e.g., 14, 15, 19] to respond to the recently uncovered bottleneck for Web information systems or e*-applications.

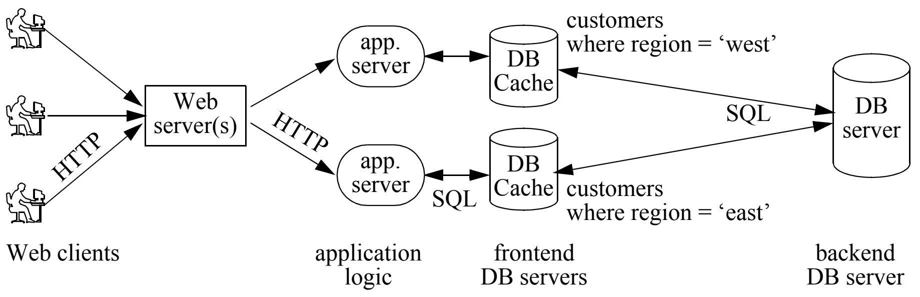


Fig. 4 DB Caching for Web Applications

4.1 Challenges of the Data Bottleneck

What is the technical challenge of all these approaches? When user requests require responses to be assembled from static and dynamic contents somewhere in a Web cache, the dynamic portion is often generated by a remote application server, which in turn asks the backend DB server (backend DB) for up-to-date information, thus causing substantial latency. An obvious reaction to this performance problem is the migration of application servers to data centers closer to the users: Fig. 4 illustrates that clients select one of the replicated Web servers "close" to them in order to minimize communication time. This optimization is amplified if the associated application servers can instantly provide the expected data—frequently indicated by geographical contexts. But the displacement of application servers to the edge of the Web alone is not sufficient; conversely it would dramatically degrade the efficiency of DB support because of the frequent round trips to the then remote backend DB, e.g., by open/next/close loops of cursor-based processing via SQL application programming interfaces (APIs). As a consequence, frequently used data should be kept close to the application servers in so-called DB caches. Note, the backend DB cannot be moved to the edge of the Web as well, because it has to serve several application servers distributed in wide-area networks. On the other hand, replication of the entire database at each application server is too expensive, because DB updates can be performed via each of them. A flexible solution should not only support DB caching at mid-tier nodes of central enterprise infrastructures [6], but also at edge servers of content delivery networks or remote data centers.

Another important aspect of practical solutions is to achieve full *cache transparency* for applications, that is, modifications of the API are not tolerated. This application transparency, which also is a prime aspect to distinguish caching from replication, is a key requirement of DB caching. It gives the cache manager the choice at run time to process a query locally or to send it to the backend DB to comply with strict consistency requirements, for instance. Cache transparency 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 DB.

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

4.2 Technical Solutions for the Cache Manager

A *full-fledged DB server* used as cache manager offers great advantages. A substantial portion of the query processing logic (parsing, optimization, and execution) has to be made available anyway. By providing the full functionality, additional DB objects such as triggers, constraints, stored procedures, or access paths can be exploited in the cache thereby simulating DB semantics locally and enhancing application performance due to increased locality. Furthermore, transactional updates seem to be conceivable in the cache (some time in the future) and, as a consequence, continued service for TWAs when backend databases become unavailable.

Note, a cache usually contains only subsets of records pertaining to a small fraction of backend tables. Its primary task is to support query processing for TWAs, which typically contain up to 3 or 4 joins [2]. Often the number of cache tables—featuring a high degree of reference locality—is in the order of 10 or less, even if the backend DB consists of hundreds of tables.

A federated query facility as offered in [14, 20] allows cooperative predicate evaluation by multiple DB servers. This property is very important for cache use, because local evaluation of some (partial) predicate can be complemented by the work of the backend DB on other (partial) predicates whose extensions are not in the cache. Hence, in the following we refer to predicates meaning their portions to be evaluated in the cache.

4.3 Database Caching—Conventional Solutions

Static approaches to DB caching where the cache contents have to be prespecified and possibly loaded in advance are of little interest in Internet applications. Such approaches are sometimes called *declarative caching* and do not comply with challenging demands such as self-administration and adaptivity. Hence, what are the characteristics of a promising solution when the backend DB is (frequently) updated and cache contents must be adjusted dynamically?

The conceptually simplest approach—full-table caching, which replicates the entire content of selected backend tables—attracted various DB cache products [22]. 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.

So far, most approaches to DB caching were primarily based on materialized views and their variants [4, 5, 8, 9, 17, 21]. A materialized view consists of a single table whose columns correspond to the set $O_V = \{O_1, \ldots, O_n\}$ of output attributes and whose contents are the query result V of the related view-defining query Q_V with predicate P. Materialized views can be loaded into the DB cache in advance or can be made available on demand, for example, when a given query is processed the nth time ($n \geq 1$). In this way, some kind of built-in locality and adaptivity (together with a replacement scheme) can be achieved. When materialized views are used for DB caching, essentially independent tables, each representing a query result V_i of Q_{V_i}, are separately cached in the frontend DB. In general, query processing for an actual query Q_A is limited to a single cache table. The result of Q_A is contained in V_i, if P_A is logically implied by P_i (subsumption) and if O_A is contained in O_{V_i} (i.e., the output of the new query is restricted by the attributes contained in a query result that is used). Only in special cases a union of cached query results, e.g., $V_1 \cup V_2 \cup \ldots \cup V_n$, can be exploited. DBProxy [3] has proposed some optimizations at the storage level. To reduce the number of cache tables, a common-schema storage-table policy is used, which tries to store query results V_i with strongly overlapping output attributes in common tables. On the one hand, a superset of the attributes Q_{V_i} may potentially enhance caching benefits of V_i, but, on the other hand, it may increase storage and maintenance costs.

A new class of caching techniques [2, 24] follows the idea that the desired cache contents are specified by so-called *parameterized cache constraints*. As soon as a reference to a parameter causes a cache miss, all records satisfying the specified cache constraint for this parameter value are loaded into the cache. As a consequence, the completeness condition is accomplished for query predicates that match the satisfied cache constraints or are subsumed by them. Hence, cache maintenance guarantees that the corresponding predicate extensions can correctly be exploited for future queries.

5 Constraint-Based Database Caching

C*onstraint-based DB 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 so-called predicate completeness in the cache such that all queries eligible for P can be evaluated correctly [11]. 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 eligible predicate extensions are data-model dependent, that is, they always support only specific operations of a data model under consideration. Cache constraints enable cache

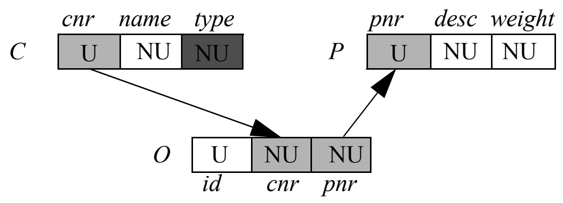


Fig. 5 Cache Table Collection *COP*

loading in a constructive way and guarantee the presence of their predicate extensions in the cache.

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

A cache contains a collection of cache tables that can be isolated or related to each other in some way. For simplicity, let the names of tables and columns be identical in the cache and in the backend DB: Considering a cache table *S*, we denote by S_B its corresponding backend table, by *S.c* a column *c* of *S*.

Assume cache tables *C, O,* and *P* where *C.cnr*, *O.cnr,* and *P.pnr* are unique (U) columns and the remaining columns are non-unique (NU), as illustrated in Fig. 5. In a common real-world situation, *C*, *O*, and *P* could correspond to backend DB tables Customer, Order, and Product. Hence, both arrows would typically characterize PK/FK relationships that can be used for join processing in the cache.

Because all columns of the corresponding backend tables are kept in the cache, all *project* operations possible in the backend DB can also be performed. Other operations like *selection* and *join* depend on specific completeness conditions enforced by cache constraints. Given suitable cache constraints, there are no or only simple decidability problems whether predicates can be evaluated. Only a simple probe query is required at run time to determine the availability of eligible predicate extensions. An important goal for cache processing is to support local evaluation of queries that typically contain simple projection (P) and selection (S) operations and equi-joins (J).

Assume for the moment, the cache enables PSJ queries, for example, with predicate Q_1 = (*C.type* = *'gold'* and *C.cnr* = *O.cnr* and *O.pnr* = *P.pnr*) on *COP*. Then, all evaluable predicates can be refined by "and-ing" additional selection terms (referring to cache table columns) to it; e.g., (and *C.name* like 'Smi%' and *O.pnr* > 17 and ...). Because full DB functionality is available, the results of these queries can further be refined by selection predicates such as Exists, Null, etc. as well as processing options like Distinct, Group-by, Having (restricted to predicates evaluable on the predicate extension), or Order-by.

5.1 Equality Predicates

Let us begin with single cache tables. If we want to be able to evaluate a given predicate in the cache, we must keep a collection of records in the cache tables such that the completeness condition for the predicate is satisfied. For simple equality predicates like *S.c* = *v*, this completeness condition takes the shape of *value completeness:* 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 $\sigma_{c=v}S_B$ are in *S*. If we know that a value *v* is complete in a column *S.c*, we can correctly evaluate *S.c* = *v*, because all rows from table S_B carrying that value are in the cache. But how do we know that *v* is complete? A straightforward way is to provide the cache manager with a list of *candidate values* of those columns we want to use in equality predicate queries. Possible candidate values for a column S.c belong to the domain of S_B.c. A list of candidate values can be specified as a complete list (all domain values), an enumeration, a range, or other predicates; candidate values can be expressed positively (recommendations) or negatively (stop-words).

Whenever a candidate value *x* occurs in an equality predicate of a query, the cache manager probes the respective cache table 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 *x*, the cache manager satisfies the value completeness for *x* by fetching 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 *x* serves as a kind of indicator that, in the immediate future, locality of reference is expected on the predicate extension determined by *x*. Candidate values therefore carry information about the future workload and sensitively influence caching performance. As a consequence, they must carefully be selected. In an advanced scheme, the cache manager takes care that only those candidate values with high re-reference probability are in the cache. By monitoring the query load, the cache manager itself can dynamically optimize the list of candidate values, for which completeness is guaranteed whenever they appear in the cache. In a straightforward case, the database administrator (DBA) specifies this list of values.

Flexible adjustment of the (dynamic) list of candidate values that are present in the cache is key to cache adaptivity. Because a probe query always precedes the actual query evaluation, completeness for a value *v* can be abolished at any time by removing all records with value *v* from the cache table. Again, in the simplest case, there may be no removal at all, and thus a value, once made complete, is left in the cache forever. Alternatively, complex replacement algorithms could be applied to unload all records carrying a complete value if its re-reference probability sinks. Note, besides the factors memory and storage space, there is always a cost trade-off between the savings for query evaluation and the penalties for keeping the records consistent with their state in the backend.

5.2 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 table columns: a source column *S.a* and a target column *T.b* where the tables *S* and *T* need not be different. RCC $S.a \rightarrow T.b$ is satisfied if and only if all values v in S.a are value complete in T.b. It ensures that, whenever we find a record *s* in *S*, all join partners of *s* with respect to $S.a = T.b$ are in *T*. Note, the RCC alone does not allow us to correctly perform this join in the cache: Many rows of S_B that have join partners in T_B may be missing from *S*. But using an equality predicate on a complete value of column *S.c* as an “anchor”, we can restrict this join to 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 ($S.c = x$ and $S.a = T.b$). In this way, a complete value can serve as an entry point for a query.

Depending on the types of the source and target columns (unique: U, non-unique: NU) on which an RCC is defined, we classify RCCs as (1:1), (1:n), and (n:m), and denote them as follows:

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

Note, using RCCs we implicitly introduce something like 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 tables. A PK/FK relationship can be processed symmetrically, whereas our RCCs can be used for join processing only in the specified direction. Other important differences are that XCs have no counterparts in the backend DB and that a column may be the source of *n* and the target of *m* RCCs. In contrast, a column in the role of PK may be the starting point of *k*, but in the role of FK the ending point of only one (meaningful) PK/FK relationship. Because a very high fraction (probably > 99%) of all SQL join queries refers exclusively to PK/FK relationships (they represent real-world relationships explicitly captured by DB design), almost all RCCs specified between cache tables are expected 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.

Assume in our *COP* example of Fig. 5 that $C.cnr \rightarrow O.cnr$ and $O.pnr \rightarrow P.pnr$ are RCCs which, as usual, characterize PK/FK relationships that guarantee regular join semantics when processed in the cache. The specification of additional RCCs $O.id \rightarrow C.type$ or even $O.cnr \rightarrow P.weight$ and $P.weight \rightarrow O.cnr$ is conceivable (assume join-compatible domains); such RCCs, however, have no counterparts in the backend DB schema and, when used for a join of *O* and *C* or a cross join of *O* and *P* or *P* and *O*, it completely remains the user's responsibility to assign a meaning.

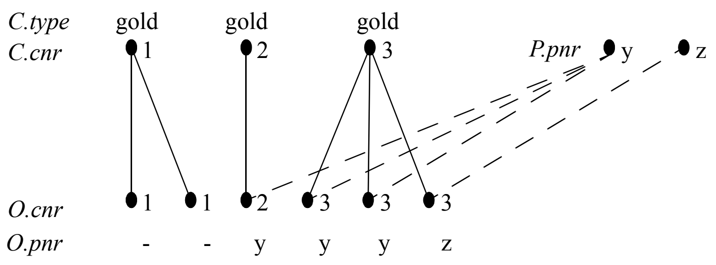


Fig. 6 Construction of a Predicate Extension for *COP*

5.3 Loading of Predicate Extensions

To evaluate predicate *Q* in the cache, the cache manager has to guarantee for *Q* predicate completeness. A collection of tables is said to be predicate complete with respect to *Q* if it contains all records needed to evaluate *Q*, i.e., its predicate extension.

An example of Q_1's predicate extension is illustrated in Fig. 6, where records are represented by bullets and value-based relationships by lines. To establish completeness for value *gold* of column *C.type*, the cache manager loads all records of $\sigma_{type\,=\,gold}S_C$ in a first step. For each of these records, RCC $C.cnr \rightarrow O.cnr$ has to be fulfilled (PK/FK relationships, solid lines), that is, all values of source column *C.cnr* (1, 2, 3 in the example) have to be made complete in the target column *O.cnr*. Finally, for all values present in *O.pnr* (y, z), RCC $O.pnr \rightarrow P.pnr$ makes their counterparts complete in *P.pnr* (FK/PK relationships, dashed lines).

In this way, the cache manager can construct predicate extensions using only simple load steps based on equality of values. Accordingly, it can correctly evaluate the corresponding queries locally. To generalize this example, we make the important observation that for the local processing of each PSJ predicate we need an entry point satisfying an equality predicate. Then we can proceed with the processing of equi-joins via reachable RCCs. Hence, each complete value is eligible for deriving a predicate to be evaluated locally.

Note, each cache-resident value of a U column is complete by definition. Furthermore, if only complete values enter a column, all values of this column are complete. This is true for *O.c* in our example. We can generalize this case to *domain completeness* greatly simplifying cache probing: A column *S.c* is said to be domain complete (DC) if and only if all values v in *S.c* are value complete.

Given a domain-complete column *S.c*, if a probe query confirms that value *v* is in *S.c* (a single record suffices), we can be sure that *v* is complete and thus evaluate $S.c = v$ locally. Unique columns of a cache table (defined by SQL constraints "unique" or "primary key" in the backend DB schema) are DC per se (implicit domain completeness). Non-unique columns in contrast need extra enforcement of DC.

5.4 Cache Groups

So far, we have introduced the general idea of supporting cache-based query evaluation using the *COP* example for a single complete value. Now we will generalize our approach and specify the predicate types to be processed in the cache together with the kind of constraints to load their predicate extensions. Our mechanism supports PSJ queries that are characterized by (valid SQL) predicate types of the form

$$((EP_1 \text{ or } ... \text{ or } EP_n) \text{ and } EJ_1 \text{ and } ... \text{ and } EJ_m)$$

where EP_i, $1 \leq i \leq n$, is an equality predicate 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 collection of the remaining cache tables involved.

For equi-join predicates, we have already introduced their specification mechanism: RCC. To establish a parameterized loading mechanism together with an entry option for cache tables, a second type of cache constraint specified on a root table and called filling column is needed: A column *S.k* with an associated list of candidate values is called a filling column. Whenever a candidate value appears in *S.k*, it is kept complete; only candidate values initiate caching when they are referenced by user queries.

Typically, filling columns are assumed simple. A multi-column mechanism different from multiple simple columns is conceivable; then, values are to be composed of simple values belonging to the participating columns. The cache manager guarantees that a candidate value present in the cache is complete. Therefore, these values—provided either manually by the DBA or automatically upon monitoring the cache traffic by the cache manager—can always be used as entry points for predicate evaluation.

Note, candidate values of filling columns play a dual role: They enforce cache loading upon reference and—once in the cache—they represent entry points for querying, because they are complete. The resulting collection of cache tables, filling columns, and RCCs is called *cache group:* the particpating cache tables are linked by a set of RCCs. A distinguished cache table is called the root table R of the cache group and holds i filling columns ($i \geq 1$). The remaining cache tables are called member tables and must be reachable from R via the (paths of) RCCs. For example, our *COP* example constitutes a simple cache group having *C* as its root table, two RCCs (m = 2), *O* and *P* as member tables, and a single equality predicate on *C.type* (n = 1) as its filling column.

Domain-complete filling columns offer a simple way of specification because lists of candidate values are not required, but they do not seem to be generally applicable[4].

Safeness of cache groups. It is unreasonable to accept all conceivable cache group configurations, because cache misses on filling columns may provoke unforeseeable load operations. Although the cache-populating procedure can be performed asynchronously to the transaction observing the cache miss, so that a burden on its own response time

4. In the DBCache project [2], so-called cache keys are used as filling columns defined to be domain complete. 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. It is therefore necessary to control the cache loading in a more refined way.

can be avoided, uncontrolled loading is undesirable: Substantial extra work, which can hardly be estimated, may be required by the frontend and backend DB servers, which will influence the transaction throughput in heavy workload situations.

Specific cache groups may even exhibit a recursive loading behavior that jeopardizes their caching performance. Once cache filling is initiated, the enforcement of cache constraints may require multiple phases of record loading. Such behavior always occurs, when two NU-DC columns of a cache table must be maintained, e.g., *C.name* and *C.type* in Fig. 5. A set of values appears in *C.name*, for which *C* is loaded with the corresponding records of C_B to keep column *C.name* domain complete. These records, in turn, populate *C.type* with a set of (new) values which must be made complete, thereby possibly introducing new values into *C.name* and so on.

Cache groups are called *safe* if there is no possibility for recursive load behavior to happen. Upon a miss on a filling column, we want the initiated cache loading to stop after a *single pass* of filling operations through the tables of the cache group. The conditions a safe cache group must meet are explored in [12].

Entry points for query evaluation. A cache table column can be correctly tested and used by an equality predicate *only if the referenced value is complete*. But how do we know that? Of course, candidate values in filling columns are explicitly made complete, and all cache table columns of type U are even domain complete.

Returning to Fig. 5, we find that *C.cnr, O.id,* and *P.pnr* are domain complete. If cache probing is successful for $C.cnr = 1$, $O.id = \alpha$, or $P.pnr = z$, respectively, we can evaluate, in addition to the predicate type *COP* is designed for, the three predicates ($C.cnr = 1$ and $C.cnr = O.cnr$ and $O.pnr = P.pnr$) or ($O.id = \alpha$ and $O.pnr = P.pnr$) or ($P.pnr = z$).

Obviously, cache-supported query evaluation gains much more flexibility and power, if we can correctly decide that other cache columns are domain complete as well. Let us refer again to *COP*. Because $C.cnr \rightarrow O.cnr$ is the only RCC that induces loading of records in *O*, we know that *O.cnr* is domain complete (called *induced domain completeness*).

Note, additional RCCs ending in *O.cnr* would not abolish the DC of *O.cnr*, though any additional RCC ending in a different column would do: Assume an additional RCC ending in *O.id* induces a new value β, which implies the insertion of $\sigma_{id = \beta} O_B$ into *O*—just a single record *o*. Now a new value 7 of *O.cnr*, so far not present in *O.cnr*, may appear, but all other records of $\sigma_{cnr = 7} O_B$ fail to do so.

For this reason, a cache table loaded by RCCs on more than one column cannot have an induced DC column. The same is true for a cache table that carries a filling column and is loaded by an RCC on a different column. Therefore, induced DC is *context dependent,* which leads us to the following definition: A cache table column *S.c* is *induced domain complete*, if it is the only column of *S* that is loaded via one or more RCCs or that is a filling column.

To summarize our discussion of cache groups concerning their population and the domain completeness of their columns: A cache table *T* can be loaded via one or more filling columns or one or more RCCs ending in one or more of its columns. A column of *T* is domain complete if it is a U column or a filling column with a complete list of candidate values or induced domain complete.

5.5 Generalization of Predicates

Cache groups enable specific PSJ queries to be evaluated in the cache. The inherent mechanism is to guarantee value or domain completeness in cache table columns and to maintain via RCCs predicate completeness across a cache group which support selection operations for equality predicates and equi-joins, respectively. Varying the fundamental idea of cache groups, we can apply the probing and completeness conditions needed for local predicate evaluation to other types of SQL predicates. A generalization of constraint specification and probing mechanisms leads us to the key observation [11] that the cache group approach can be extended to

- simple predicates with other comparison conditions $\Theta \in \{<, >, \leq, \neq, \geq\}$
- range predicates or even
- complex predicates composed of them by Boolean operators ($\vee$, $\wedge$, $\neg$).

Furthermore, it is conceivable, however much more complex, to establish predicate completeness for aggregation, recursion, and other SQL predicates (Exists, Subquery, etc.). The usefulness and realization aspects of such extensions have to be explored yet.

6 Seamless Processing of Web Objects

Obviously, all these ideas of constraint-based DB caching are not restricted to the relational model.or to SQL predicates. They may be applied equally well to other data models and the caching needs of their applications, e. g., to XML documents and XQuery operations [27]. This observation delivers another argument for the opportunities and benefits of the upcoming XML database management systems (XDBMS). If they are native, that is, if they provide for the variety of XML language models (such as SAX, DOM, XPath, and XQuery [27]) specific access models to fine-grained storage structures tailored to the processing requirements of XML documents [10], then there is no need anymore to perform frequent, different, and heterogeneous type conversions often complained in RDBMS-based e*-applications. Message data and DB data could be managed and stored in the same way. Hence, queries on DB-based data could be directly evaluated on its native XML storage structures. Their result sets shaped as XML fragments could be forwarded and stored in the various DB and Web caches up to the user thereby only handled by a single and, therefore, homogeneous processing model.

The currently futuristic view of XDBMS dominance was already taken in [13] where innovative DBMS architectures were explored. As a consequence of such a technological change, the myriads of SQL applications would become legacy applications to be emulated on, say, XQuery interfaces—nowadays a rather weird imagination.

7 Open Problems

We have considered the entire user-to-data path in Web applications and have discussed the caching problems occurring under a view which separated the specified problems. Web caching achieved by four different kinds of caches targets at the minimized communication effort and freshness of single Web objects, whereas DB caching attempts to perform as much query evaluation as possible (and cost-effective) in caches close to the edge of the Internet—both to primarily reduce the user-perceived delay of Web requests. In contrast to Web caching where only identifier-based access is supported for Web objects, declarative and set-oriented query processing of database records is in the focus of DB caching.

In the future, fragment-enabled fine-granular caching can essentially improve the effectiveness of all kinds of Web caches. Furthermore, various protocol refinements seem possible to improve caching and content delivery of uncacheable HTTP content [28]. For example, pushing the functionality of uncacheable content generation to the network edge may have a substantial effect. Another promising area is the monitoring and recognizing of access patterns in caches and exploiting their results in prefetching schemes. For DB caching, we seem 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.

So far, all aspects of cache maintenance [6] were excluded. How difficult is it to cope with the units of loading and unloading? Let us call such a unit cache instance (CI), which is a collection of records satisfying all RCCs of a cache group for a single root record. Depending on their complexity, CIs may exhibit good, bad, or even ugly maintenance properties. The good CIs are disjoint from each other and the RCC relationships between the contained records form trees, for example, a cache group consisting of customer and order only (CO). Then a newly referenced candidate value (NU) of *C.ctype* causes a forest of such trees to be loaded, which, in case of unloading, can be removed without interference with other CIs. The bad CIs form DAGs and weakly overlap with each other. Cache group *COP* in Fig. 6 is an example where several CIs may share records of cache table *P*. Hence when loading a new CI, one must beware of duplicates. Accordingly, shared records must be removed only together with their last sharing CI. To maintain cache groups with cross constraints can be characterized as ugly, because CIs may strongly overlap so that duplicate recognition and management of shared records may dominate the work of the cache manager.

Improvement of adaptivity is another important problem, much more difficult than in Web caches. 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 RCC 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 trade-off point of cache operation. For example, starting with the cache tables and join paths exhibiting the highest degrees of reference locality, the cache group design can be expanded by additional RCCs and/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. The kind of self-administration or self-tuning opens a new and complex area of research often referred to as autonomic computing.

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.

References

[1] Akamai Technologies Inc.: Akamai EdgeSuite. http://www.akamai.com/en/html/services/edgesuite.html

[2] Altinel, M., Bornhövd, C., Krishnamurthy, S., Mohan, C., Pirahesh, H., Reinwald, B.: Cache Tables: Paving the Way for an Adaptive Database Cache. Proc. 29th Int. Conf. on Very Large Data Bases (VLDB'03), Berlin (2003) 718–729

[3] Amiri, K., Park, S., Tewari, R., Padmanabhan, S.: DBProxy: A Dynamic Data Cache for Web Applications. Proc. 19th Int. Conf. on Data Engineering (ICDE'03), Bangalore, India. (2003) 821–831

[4] Anton, J., Jacobs, L., Liu, X., Parker, J., Zeng, Z., Zhong, T.: Web Caching for Database Applications with Oracle Web Cache. Proc. 2002 ACM SIGMOD Int. Conf. on Management of Data, Madison, Wisc. (2002) 594–599

[5] Bello, R. G., Dias, K., Downing, A., Feenan, J. J., Jr., Finnerty, J. L., Norcott, W. D., Sun, H., Witkowski, A., Ziauddin, M.: Materialized Views in Oracle. Proc. 24th Int. Conf. on Very Large Data Bases (VLDB'98), New York (1998) 659–664

[6] Bornhövd, C., Altinel, M., Mohan, C., Pirahesh, H., Reinwald, B.: Adaptive Database Caching with DBCache. Data Engineering Bulletin 27:2, (2004) 11-18

[7] Brodie, D., Gupta, A., Shi, W.: Accelerating Dynamic Web Content Delivery Using Keyword-Based Fragment Detection. Proc. Int. Conf. on Web Engineering, Munich, LNCS 3140, Springer (2004) 359-372

[8] Dar, S., Franklin, M., Jónsson, B., Srivastava, D., Tan, M.: Semantic Data Caching and Replacement. Proc. 22nd Int. Conf. on Very Large Data Bases (VLDB'96), Mumbai (1996) 330–341

[9] Goldstein, J., Larson, P.-A.: Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, CA (2001) 331–342

[10] Haustein, M., Härder, T.: Fine-Grained Management of Natively Stored XML Documents. submitted (2005)

[11] Härder, T., Bühmann, A.: Query Processing in Constraint-Based Database Caches. Data Engineering Bulletin 27:2 (2004) 3-10

[12] Härder, T., Bühmann, A.: Value Complete, Domain Complete, Predicate Complete—Magic Words Driving the Design of Cache Groups, submitted (2005)

[13] Halverson, A., Josifovski, V., Lohman, G., Pirahesh, H., Mörschel, M.: ROX: Relational Over XML. Proc. 30th Int. Conf. on Very Large Data Bases (VLDB'04), Toronto (2004)

[14] IBM DB2 Universal Database (V 8.1). http://www.ibm.com/software/data/db2/

[15] IBM Cloudscape. http://www.ibm.com/software/data/cloudscape/

[16] Jin, S., Bestavros, A.: GreedyDual*: Web Caching Algorithms Exploiting the Two Sources of Temporal Locality in Web Request Streams. Proc. 5th Int. Web Caching and Content Delivery Workshop (2000)

[17] Keller, A., Basu, J.: A Predicate-Based Caching Scheme for Client-Server Database Architectures. VLDB Journal 5:1 (1996) 35–47

[18] Krishnamurthy, B., Rexford, J.: Web Protocols and Practise: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement, Addison-Wesley, Reading, MA (2001)

[19] Larson, P.-A., Goldstein, J., Zhou, J.: MTCache: Mid-Tier Database Caching in SQL Server. Proc. 20th Int. Conf. on Data Engineering (ICDE'04), Boston, MA (2004) 177-189

[20] Larson, P.-A., Goldstein, J., Guo, H., Zhou, J.: MTCache: Mid-Tier Database Caching for SQL Server. Data Engineering Bulletin 27:2 (2004) 35-40

[21] Levy, A. Y., Mendelzon, A. O., Sagiv, Y., Srivastava, D.: Answering Queries Using Views. Proc. 14th ACM Symposium on Principles of Database Systems (PODS'95), San Jose, CA (1995) 95–104

[22] Oracle Corporation: Internet Application Sever Documentation Library, http://otn.oracle.com/documentation/appserver10g.html

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

[24] The TimesTen Team: Mid-tier Caching: The TimesTen Approach. Proc. 2002 ACM SIGMOD Int. Conf. on Management of Data, Madison, Wisconsin (2002) 588–593

[25] Tsimelzon, M., Weihl, B., Jakobs, L.: ESI language specification 1.0 (2001), http://www.esi.org/language_spec_1-0.html

[26] Weikum, G.: Web Caching. In: Web & Datenbanken – Konzepte, Architekturen, Anwendungen. Erhard Rahm/Gottfried Vossen (Hrsg.), dpunkt.verlag (2002) 191-216

[27] W3C Recommendations. http://www.w3c.org (2004)

[28] Zhou, Z., Mao, Y., Shi, W.: Workload Characterization of Uncacheable HTTP Content. Proc. Int. Conf. on Web Engineering, Munich, LNCS 3140, Springer (2004) 391-39