Foundations and Trends® in
Databases
Vol. 2, No. 3 (2009) 169–266

DOI: 10.1561/1900000005

now
the essence of knowledge

Caching and Materialization for Web Databases

By Alexandros Labrinidis, Qiong Luo,
Jie Xu and Wenwei Xue

Contents

Foundations and Trends® in
Databases
Vol. 2, No. 3 (2009) 169–266

DOI: 10.1561/1900000005

now
the essence of knowledge

Caching and Materialization for Web Databases

Alexandros Labrinidis[1], Qiong Luo[2], Jie Xu[3] and Wenwei Xue[4]

[1] *University of Pittsburgh, USA, labrinid@cs.pitt.edu*
[2] *Hong Kong University of Science and Technology, Hong Kong, luo@cse.ust.hk*
[3] *University of Pittsburgh, USA, xujie@cs.pitt.edu*
[4] *Hong Kong University of Science and Technology, Hong Kong, wwxue@cse.ust.hk*

Abstract

Database systems have been driving dynamic websites since the early 1990s; nowadays, even seemingly static websites employ a database back-end for personalization and advertising purposes. In order to keep up with the high demand fuelled by the rapid growth of the Internet, a number of caching and materialization techniques have been proposed for web databases over the years. The main goal of these techniques is to improve performance, scalability, and manageability of database-driven dynamic websites, in a way that the quality of data is not compromised. Although caching and materialization are well-understood concepts in the traditional database and networking/operating systems literature, the Web and web databases bring forth unique characteristics that warrant new techniques and approaches.

In this monograph, we adopt a data management point of view to describe the system architectures of web databases, and analyze the research issues related to caching and materialization in such architectures. We also present the state-of-the-art in caching and materialization for web databases and organize current approaches according to the fundamental questions, namely how to store, how to use, and how to maintain cached/materialized web data. Finally, we associate work in caching and materialization for web databases to similar techniques in other related areas, such as data warehousing, distributed systems, and distributed databases.

1

Introduction

Database systems have been driving dynamic websites since the early 1990s, and caching and materialization have been the major techniques to improve the performance, scalability, and manageability of such web databases. Different from a traditional database environment, the software components of a web database, including web servers, database servers, application servers, and possibly additional middleware, are largely independent from one another, even though they work together as a holistic system (Figure 1.1). Caching and materialization techniques for such web databases consider a number of issues at different parts of the system and they bring interesting challenges and opportunities.

In addition to the inherent architectural uniqueness, web databases also come with stringent demands for near-real-time performance (at the speed of thought) and ability to withstand high request volumes (e.g., due to *flash crowds*, giving caching and materialization techniques a pivotal role in such environments.

In this monograph, we look at the entire process of caching and materialization for web databases as a sequence of actions (or verbs). For each action, we identify the possible options and present

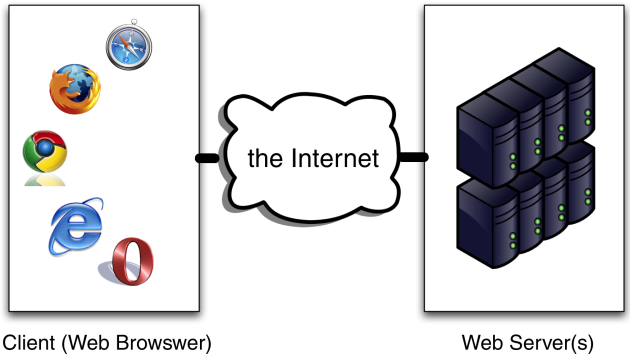


Fig. 1.1 10,000-feet view of a typical web-database architecture.

representative techniques and papers for each one. Specifically, we look into the following actions, as they relate to caching and materialization for web databases:

- **Store:** How is data cached/materialized, including where and at what granularity to cache/materialize data, and how to select the data items to cache/materialize.
- **Use:** How do we use cached/materialized data items to answer new requests?
- **Maintain:** How are caches maintained up-to-date, including when and how to handle updates, how to organize the cache to answer new requests efficiently, and how to perform cache replacement as necessary.

When presenting existing techniques on these three actions, we focus on the performance and quality aspect (in Section 6). In brief, we examine the following three types of evaluation metrics: (1) Quality of Service (QoS) metrics, including response time, throughput, and availability; (2) Quality of Data (QoD) metrics, mainly about data freshness and accuracy, and (3) user-centric metrics, usually in the form of quality contracts.

Among the above evaluation metrics, throughput is a major one to consider for web databases, and one way to achieve a high throughput is through high-performance hardware configurations. However, enhancing system throughput using high-end configurations is not

always desirable; a major factor is the *Total Cost of Ownership (TCO)* [44]. TCO is the direct and hidden lifetime IT cost of purchasing, operating, managing, and maintaining a computing system [31]. Example cost items include hardware, software, network communication, administration, personnel training, and technical support. For a database-backed website, an increase in the throughput through high-end configurations may result in the increase of all those TCO cost items and in turn the TCO itself. Therefore, system throughput should be considered together with TCO. In particular, transparent and adaptive caches are often an ideal candidate to increase system throughput automatically with little increase on TCO.

Note that an important factor to consider in any caching and materialization technique for web databases is whether the scheme supports database transaction semantics, even though many web applications today work well with a weaker consistency guarantee. Eric Brewer proposed the CAP Theorem [23], which states that, for any distributed computing system, only two out of the following three key requirements can be satisfied: Consistency, Availability, and Partition tolerance. The theorem was later formally proved by Gilbert and Lynch [61] for both asynchronous and partially synchronous network models. As such, in real-world distributed web databases, where availability and scalability are crucial, it is unavoidable to make compromises on consistency. Therefore, caching and materialization are a good match to such systems in that they can help on both availability and partition tolerance and can benefit from the relaxed consistency requirement. In practice, many commercial database-backed websites today have focused their technical development, including deploying caches throughout all levels of the system, on ensuring service availability and tolerance upon data partitioning/distribution, while sacrificing consistency to various degrees. Well-known examples of such websites include Amazon, EBay, and Twitter.

Roadmap: The rest of this monograph is structured into four separate components as follows:

- **Architecture** — this part describes a typical web-database architecture and discusses some of the complexities intrinsic in such setups (Section 2).

- **Taxonomy** — this part presents the three "verbs" in detail, along with the possible alternatives. We also give a detailed description of a typical web-database architecture (Sections 3–5, followed by a discussion on metrics, in Section 6).
- **Projects** — this part describes some representative web-database projects, while explaining the choices each project adopted under the presented taxonomy (Section 7).
- **Related work** — this section presents related work from other areas (Section 8).

We conclude this monograph with a short discussion on some open problems and future directions.

2

Typical Architecture

As we saw in the 10,000-feet view of the typical web-database architecture (Figure 1.1), we can roughly partition the whole system into three different components: the client, the Internet, and the server.

- The *client* component is essentially the end-user's computer and web browser, which may also contain a local cache.
- The *Internet* component contains a plethora of different, interconnected cogs that make possible the connection of the client to the server. A Domain Name Server (DNS) [51] is typically used to decode the name of the web address into an IP address. Given this IP address, routers are used for the client to establish a connection with the server, using the HTTP protocol [82]. It should be noted that the IP address of a web server can be differentiated according to the end-user's IP address to take advantage of server proximity (when there are alternatives due to replication). In this environment, there are of course multiple opportunities for caching and materialization; we describe these later.
- The *server* component essentially includes multiple, different servers that are collectively seen as the web server for

the end-user. In particular, the entry point for a cluster of servers would normally be a *load balancer* or *web switch* that is routing incoming web traffic to the different web servers available in the cluster. Of course, the entry point for a single web server is the web server itself. The web server is responsible for handling and responding to HTTP requests and is typically connected to an application server, that is used to capture the logic of the web application. The application server typically connects to a database server, which is used to store data. As Figure 2.1 indicates, there are multiple opportunities for caching within the server component; we describe these later as well.

This description is roughly equivalent to that of a single, small-scale data center operation. Many companies today go beyond such architecture in one of the two ways:

1. By replicating operations over multiple (dedicated) data centers that are distributed geographically to decrease the effective network "distance" to the end-users, while

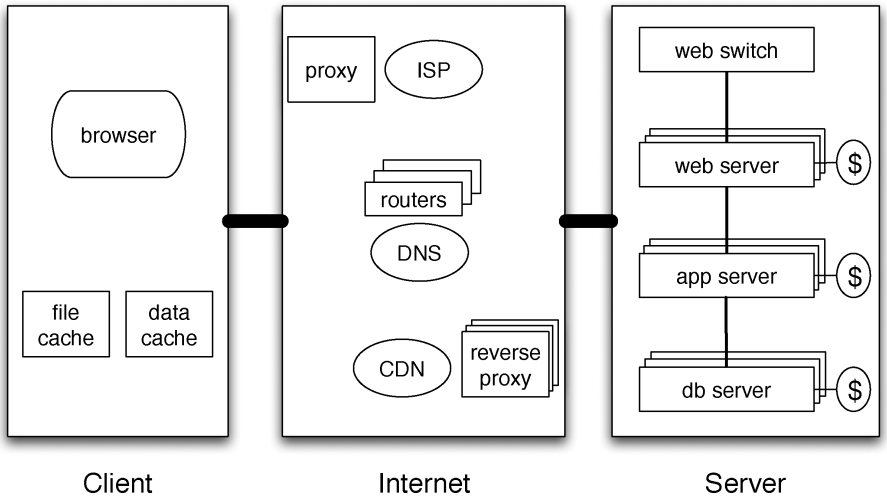


Fig. 2.1 Typical web-database architecture ($ signs indicate possible location for caches at the server).

increasing availability and throughput. Routing to a specific data center is performed by exploiting DNS resolution, as explained earlier.
2. By utilizing a Content Distribution Network (CDN), such as the one by Akamai, where content (but not necessarily operations) is replicated and distributed geographically.

As web services become more prevailing, the server "lines" in the above model become more blurred, usually spanning multiple organizations. These days, it is not uncommon for a single web page downloaded at an end-user's browser to involve requests to multiple, different web servers performing different functionality. For example, if we consider the main web page of an online newspaper, that single web page may, in turn, invoke requests to multiple web servers, for the following types of services:

- *Banner advertising* — a separate web server from which a banner image (advertisement) is downloaded. A separate server is crucial for third-party accuracy (and honesty) in reporting the number of times a certain ad was placed on a site's web pages.
- *Text advertising* — an HTML fragment served from another server that contains a textual advertisement which is targeted to the specific page and/or audience. A separate server is typical in such cases, given the prevalence of a few major text-based advertising networks (e.g., Google AdWords,[1] Yahoo SearchMarketingm,[2] and Microsoft adCenter[3]) with proprietary algorithms driving ad selection and placement.
- *Web Analytics* — a separate request (typically made through an embedded `img` request), that is used to record advanced statistics about site visitors. A separate server for the majority of websites is typical just because of the convenience it provides to low-budget web server operators (e.g., Google

[1] http://adwords.google.com/
[2] http://searchmarketing.yahoo.com/
[3] https://adcenter.microsoft.com/

Analytics[4]). Such services are essentially the natural extension of the *hit counters* from the early days of the Web.

- *Media* — a content distribution network is often used to serve media files (e.g., images, audio, or video clips), because of size and streaming requirements. In such settings, serving the content from the server that is located the closest to the end-user often has a big impact.
- *Data from other sources* — it is common for some of the data presented on a web page to originate from other sources. The popularity of AJAX [6] is leading to such data "importing" being done in an online fashion, with the original page including the *code* to import that data from the authoritative source directly, instead of just including the data as part of the page. A characteristic example of this is stock market information, where an online newspaper (e.g., NY Times) typically has a fragment of the web page devoted to a summary of the stock market; this fragment uses AJAX to import the latest values from other sources (and to continually keep them up-to-date while the stock market is open).
- *Other online properties* — given the growing trend toward syndication, it is common to include content from other online "properties" and therefore different web servers. For example, an online newspaper can include content from its associated magazine or from a news network that it belongs, e.g., the Pittsburgh Post-Gazette website[5] includes in its main web page fragments from the Associated Press news network. Although such content could be hosted in the same web server, this is often not the case, because of organizational boundaries. This is sometimes taken to the extreme, when all content on a web page is drawn from other sources, for example in a personalized web portal (e.g., iGoogle).

[4] http://www.google.com/analytics/
[5] http://www.post-gazette.com

3

Store How are Data Cached/Materialized?

Having described the typical web-database architecture in detail, in this section we address the question of where to cache/materialize web data, at what granularity, and how to select appropriate data to cache/materialize (Figure 3.1).

3.1 [Store] Location of Caching/Materialization

Web caching can be performed at one of the following four locations: (i) server, (ii) reverse proxy, (iii) proxy, and (iv) client. The different

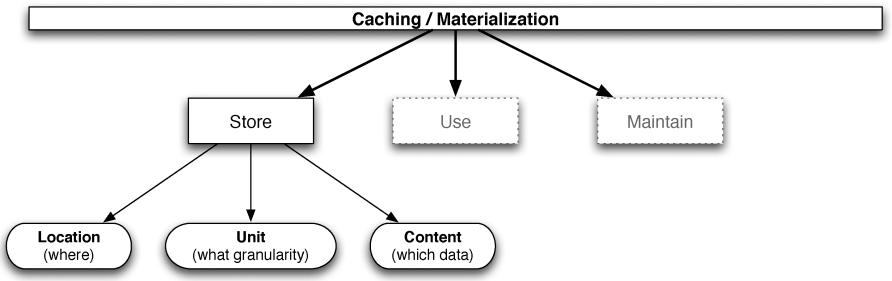


Fig. 3.1 Taxonomy for store question.

location options (i)–(iv) are ordered in increasing order of distance between the cache and the database-backed website.

Figure 2.1 shows the different locations of web caching. As explained in the previous section, a *server* denotes all machines "responsible" for a database-backed website, including the database server, the application server and the web server. We use the two terms "database-backed website" and "database-driven website" interchangeably in this monograph. A *reverse proxy* is set up by the server, or is owned by a *Content Delivery Network* (CDN) and contracted to be used by the server. It is also called a *CDN cache* or a *web portal*. A *proxy* is set up for a group of Internet users, e.g., by an *Internet Service Provider* (ISP), and is also called a *forward proxy* or an *edge server*. Finally, a *client* denotes a machine where the user's web browser is located. A *client cache* is also called a *browser cache*. We discuss these options in detail in the next paragraphs.

- **Server Caches:** Database-backed websites today often have a multi-tier architecture to improve scalability. Based on this architecture, we further classify a server-side cache into one of the following three types: (i) a database cache, (ii) a mid-tier cache, and (iii) a web server cache.
  - *Database Cache*: A database cache contains materialized database views. These views are managed by the same DBMS instance as the original database tables.
  - *Mid-Tier Cache*: A mid-tier cache contains materialized views stored at the application server. This cache is usually a DBMS with some extensions for caching purposes. This DBMS can either be the same instance or one different from the database server. For the purposes of this monograph we are interested primarily in the caching/materialization of data.

    In addition to raw database data, web application servers perform in-memory caching of programming language-level objects, such as *Enterprise JavaBeans* (EJBs), that are derived from the back-end database [79].

— *Web Server Cache*: A web server cache contains web pages or fragments. A *web fragment* is part of a web page, e.g., an HTML segment wrapped around database query results. A web page is also called a *document* and a fragment called an *object*.

— *Combination of Server-Side Caches*: A multi-tier server-side cache that combines all three types of the server-side caches is possible.

- **Reverse Proxy:** We categorize existing approaches to reverse proxy caching into *single cache* and *multi-cache*. The former considers caching on an individual reverse proxy, while the latter on multiple proxies as a group. The caches in the group are usually reverse proxies that belong to the same organization, e.g., a content delivery network. As a whole, they serve the caching needs of one or more websites.
- **Proxy:** Proxy caches have been widely used throughout the Internet to improve client response time and reduce server workload.
- **Client Caches:** In comparison with the large number of publications on server-side or proxy caching, there is less existing work on client-side caching for web databases. One reason is that client-side caching benefits only clients and browser caches are already sufficient for the purpose.

  Typically, the local client cache has been a *file cache*, which has been used to store static files, usually images. Increasingly, browser cases are being used for caching data, primarily driven by the *"AJAX effect"*. AJAX[1] basically provides the ability to update fragments of a web page without the need to refresh the entire page. This, on the one hand, enables interactive, data-driven web applications and, on the other hand, opens up an entire spectrum of location caching and materialization options, not previously available.

[1] http://en.wikipedia.org/wiki/Ajax_(programming).

3.2 [Store] Unit of Caching/Materialization

Data in a web cache are organized into logical units of certain granularity. A cache unit is the finest granularity of cached data for cache management. A web cache can contain four types of units: (i) raw data, (ii) views, (iii) fragments, and (iv) pages.

- **Raw Data:** Raw data refers to the base data originally stored as tables in a database. In other words, the whole or a subset of tables in the original database is replicated in the cache. This is called *table-level caching*.
- **Views:** Instead of storing raw data, another alternative is to store the results of queries, in other words to store *views*. In general, views include materialized views, query results, and horizontal or vertical partitions of database tables. Caching of views is often referred to as *query-level caching*. The type of views stored has a big impact on reusability; we elaborate this in the next section.
- **Fragments (or WebViews [102]):** *Fragment-level caching* refers to the caching of dynamic web fragments. A fragment is loosely defined as any part of an HTML page. We are typically interested in fragments that contain data which were queried from a database, although other types of fragments are also possible [131].
- **Pages:** *Page-level caching* refers to the caching of entire web pages, no matter a page is dynamic or static. Most multi-cache approaches to reverse proxy caching employ page-level caching. Current browsers also typically employ object-level caching, be it an entire page or embedded objects within a page (i.e., an image, a video clip, etc.).

3.3 [Store] Selecting Content for Caching/Materialization

Traditional caching approaches will cache every query result and decide when the cache is full on which item to *evict* using a *cache replacement algorithm*; these are discussed in Section 5.5.

Materialization goes a step beyond traditional caching, as it often implies an additional commitment: the "promise" to keep the materialized results up-to-date, by refreshing the materialized version if updates occur on the source data. In such a setting, it is important to select which data to keep materialized: the main concern in this case is *time* (to process the updates) and not *space*, as is the case with traditional caching.

Labrinidis and Roussopoulos [105, 107] first studied the approach of adaptively selecting WebViews to materialize, based on both performance and freshness metrics. To address the online selection problem (i.e., selecting which WebViews to materialize to strike the balance between the performance and data freshness), Labrinidis and Roussopoulos [107] propose an adaptive algorithm, OVIS(θ). Given the user-specified data freshness threshold θ, OVIS(θ) operates in two modes: passive and active. In the passive mode, OVIS(θ) collects statistics and monitors the freshness degree for the served data. When it periodically goes into the active mode, it distinguishes between two cases based upon the observed freshness value. When the observed freshness degree is higher than the threshold θ, OVIS(θ) identifies a freshness surplus and materializes more WebViews to improve response time. On the other hand, when the observed freshness degree is less than the threshold θ, the algorithm identifies a freshness deficit and stops materializing WebViews to increase the freshness value. Which views are selected to change the materialization policy is decided by greedy algorithms to guarantee that OVIS(θ) meets the response-time constraint.

In general, the selection problem is closely related to the maintenance policy or policies available. For example, the OVIS(θ) algorithm assumes a recomputation-based update model, and as such it can allow for some updates to be unapplied. This would not have been possible with an incremental-based update model. For more details on these options, see Section 5.

4

Use Using Cached Content

In this section we deal with the "use" verb as it related to caching and materialization for web databases. In particular, we first describe *query scheduling*, i.e., how to determine the proper order of executing user requests, and then describe *query processing*, i.e., what alternatives existing when processing user requests in conjunction with cached/materialized data (Figure 4.1).

4.1 [Use] Query Scheduling

For cases where caching or materialization cannot provide a (quick) answer to a user request, the request often get "translated" to one ore more queries submitted to the back-end database. Given the inherent interactive nature of most web applications, there is significant pressure for executing such queries in an efficient manner. As such, *query scheduling* becomes an important issue.

Query scheduling has been studied extensively in the context of traditional database systems and of real-time (database) systems [1, 77, 130]. In the context of web servers, in general, and web-databases, in

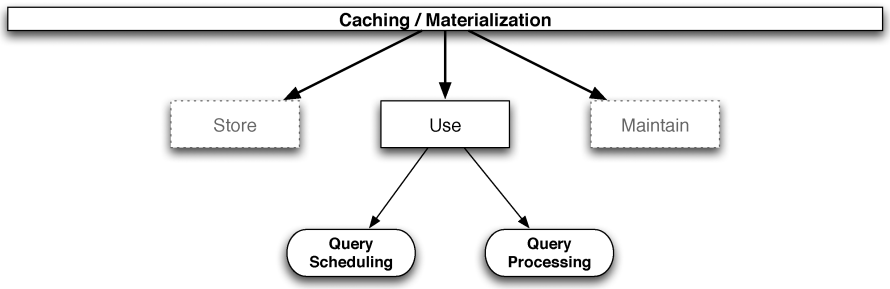


Fig. 4.1 Taxonomy for use question.

particular, there are two defining characteristics that scheduling policies need to address:

- the need to **handle overload** — web traffic can exhibit bursty behavior (e.g., flash crowds), so handling overload gracefully is crucial.
- the need to **consider updates** — web databases need to be online, handling both queries and updates, while respecting the end-user's desire for interactivity (i.e., real-time performance).

A *flash crowd* on the Web, also referred to as the *slashdot effect*, is used to describe the phenomenon when a website catches the attention of a large number of people, and gets an unexpected and overloading surge of traffic. The term originates from a 1973 English language novella by science fiction author Larry Niven, one of a series about the social consequence of inventing an instantaneous, practically free transfer booth that could take one anywhere on Earth in milliseconds (i.e., teleporting). One consequence not foreseen by the builders of the system was that with the almost instantaneous reporting of newsworthy events, tens of thousands of people worldwide along with criminals would flock to the scene of anything interesting, hoping to experience or exploit the instant disorder and confusion so created. More info at http://en.wikipedia.org/wiki/Flash_crowd.

Schroeder and Harchol-Balter's work [147] is one of the first works to identify smart scheduling as a way to alleviate the negative effects

of overload in web servers and propose using the Shortest-Remaining-Processing-Time (SRPT) first policy as the solution. Recently, Guirguis et al. [67] have proposed a parameter-free policy, called *ASETS* that works best for scheduling web transactions (that have deadlines) under any workload condition, including overload situations. The proposed policy is essentially a hybrid between the SRPT policy (that is known to work best under overload conditions) and the Earliest-Deadline-First (EDF) policy (that is known to work best under low-utilization conditions). Instead of setting a threshold in the system where the scheduling policy switches from EDF to SRPT and vice versa, ASETS essentially assigns a scheduling policy to each transaction in the system. The authors also present extensions to handle the scheduling of transactions that are part of a workflow (i.e., in the presence of dependencies), which is more suited to the fragment hierarchies present in the creation of web pages.

When considering both queries and updates, one has to worry about timeliness of receiving query results, but also about the freshness of those results. There is a significant amount of related work in the context of real-time databases, which we cover in Section 8.2.4. In the context of web databases, Qu and Labrinidis [74] propose a meta-scheduling framework in order to fairly allocate processing of updates and queries, according to *user preferences*. They propose keeping two separate queues (one for queries and one for updates) and execute tasks within each queue according to well-established policies, but allocate resources between the two queues by taking into account user preferences on QoS and QoD (more on this in Section 6.3).

4.2 [Use] Query Processing

Depending on the unit of cached data, query processing (i.e., serving a user request) can consider previously stored data (e.g., cached results). In particular, a cache for web databases can be equipped with four types of processing capabilities: (i) *no query processing*, (ii) *containment-based query processing*, (iii) *semantic query processing*, and (iv) *full-fledged SQL query processing*. Type (i) is also called passive caching, whereas types (ii)–(iv) are called active caching [119, 120, 121, 122].

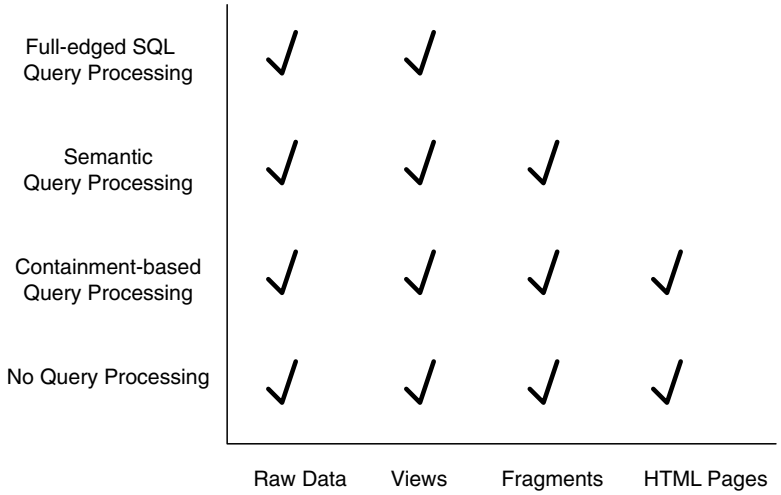


Fig. 4.2 Applicability of query processing options to different units of caching.

The unit of caching (i.e., raw data, views, fragments of HTML pages) determines what types of processing capability a cache may have (Figure 4.2). Obviously, all caches can support the trivial, *no query processing* option. Typically, all caches could also support the query containment functionality (especially the exact match flavor); all that is needed is a way to uniquely describe the cached data/object. What we refer to as semantic query processing cannot be applied when the unit of caching is entire pages, whereas full-fledged SQL processing can only be supported by a cache if the unit of caching is either views or raw data.

In the next paragraphs we describe the different alternatives, after first discussing the fundamental driver behind content reuse via caching/materialization, namely the existence of a relationship between the results of two queries.

The relationship between the results of two SQL queries, in short, the relationship of two queries, can be of four types: exact match, containment, overlapping, and disjoint [37, 42, 121]. *Exact match* means the results of the two queries are equivalent. *Containment* means the result of one query is a proper subset of that of the other query. *Overlapping* means that the results of two queries have a non-empty

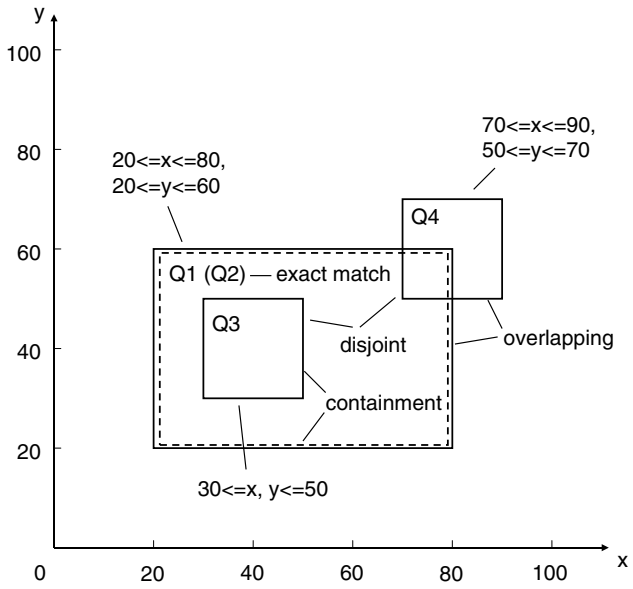


Fig. 4.3 Illustration of query relationships.

intersection and the two queries are neither an exact match nor satisfy the containment relationship. *Disjoint* means the intersection of the two query results is empty.

Figure 4.3 shows an illustration of the four types of query relationships. In the figure, the relationships between the pairs of queries (Q1, Q2), (Q1, Q3), (Q1, Q4) and (Q3, Q4) are exact match, containment, overlapping, and disjoint, correspondingly. The predicates of the queries are all denoted in the figure. Given these query relationships, we say that Q2 and Q1 are *equivalent*, Q3 is *contained* in Q1, Q4 *overlaps* Q1, and Q3 and Q4 are *disjoint*.

For the four types of processing capabilities, *no query processing* only checks the exact match relationship between the cached queries and a new query while containment-based query processing checks containment as well as exact match. Both semantic query processing and full-fledged SQL query processing check all four types of relationships. The difference is the former uses a special-purpose query processing module, whereas the latter uses a general-purpose DBMS product to

handle query processing in the cache. We present representative works from each of the four categories next.

4.2.1 [Use::QueryProcessing] No Query Processing

This option represents the traditional, exact match-based passive caching. Each web page is stored with its URL, possibly tagged with the query parameters for a dynamic page, in the cache [12, 29, 139, 140, 141, 151, 157, 172]. When the cache sees an HTTP request, it returns the cached page of the corresponding URL. Alternatively, a web page, often a dynamic web page, can be divided into individual fragments and these fragments are stored separately [26, 43, 104, 105, 106, 107, 136, 138]. When the cache composes a dynamic web page, missing fragments of the page are fetched from the server [43].

A query result is regarded as a fragment in this option, similar to an image or a video file. The cache returns a cached query result only if the new query in an HTTP request is an exact match with the cached query [119, 121].

4.2.2 [Use::QueryProcessing] Containment-Based Query Processing

This option answers a new query locally in the cache if the result of the new query is completely contained in the cached result of a previous query. Otherwise, the new query is forwarded to the server without any local processing and the query result returned by the server is merged into the cache.

4.2.3 [Use::QueryProcessing] Semantic Query Processing

This option answers a new query locally in the cache, if the query result partially overlaps the cached results of previous queries. The new query is decomposed into a probe query that is evaluated in the cache and a remainder query that is transmitted to and evaluated at the server. The result of the remainder query is then merged into the cache. Compared to the containment-based processing, semantic query processing requires a stronger server cooperation capability for the server to handle

remainder queries and more complex cache management techniques to evaluate queries whose results overlap the cached results.

4.2.4 [Use::QueryProcessing] Full-Fledged SQL Query Processing

This option uses a full-fledged DBMS as the cache in order to be able to optimize and execute queries at the cache. A distributed evaluation plan is generated for a query and is optimized in a way similar to that in distributed databases. The plan efficiently utilizes both data in the local cache and data at the remote server. Of course, such an option introduces complications if transactional consistency is required (and there are updates).

5

Maintain Cache Maintenance

In this section of our taxonomy, we present the different issues stemming out of the "maintain" verb of caching materialization in web databases (Figure 5.1).

5.1 [Maintain] Cache Initialization: Proactive versus Reactive

The timing of the initialization of a web cache can be either proactive or reactive. Essentially, *reactive caching* is done as a *response* to user queries, whereas *proactive caching* is performed in *anticipation* of user

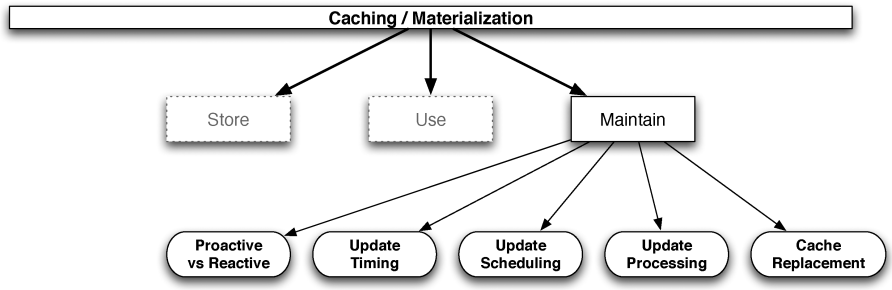


Fig. 5.1 Taxonomy for maintain question.

queries. With reactive caching, the cache is initially empty and data are inserted/deleted from the cache during its operation. With proactive caching, the cache is essentially pre-populated with data during initialization and actively maintained during its operation. We often refer to proactive caching as *materialization*, whereas reactive caching maps to the "traditional" notion of caching.

To better understand the differences between the two flavors of caching, we look into the operation of the cache when a user query arrives (at the cache) and when an update arrives (at the server) in Figure 5.2. In the case of pure reactive caching, any decisions on how to handle cached data get triggered by incoming user queries (Q1). If there is a cache miss (e.g., because of an expired TTL, which we explain in the next section), then the cache issues a request to the server (Q2). The server's reply (Q3) is usually stored in the cache and returned to the user (Q4). In case of a cache hit, the answer is given back to the user immediately (Q1 $\rightarrow$ Q4). In the case of pure proactive caching, updates also trigger changes in the cache contents, for example, to update cached data that are expected to be accessed again in the near future. As such, an incoming update (U1) could generate an update to be propagated to the cache (U2), in order for the cache contents to be fresh.

Of course, there are many variations along the spectrum between pure proactive caching and pure reactive caching, as identified from the answers to the following questions:

- What are the options for the server to communicate with the cache? We discuss this further in the next section (Section 5.2).

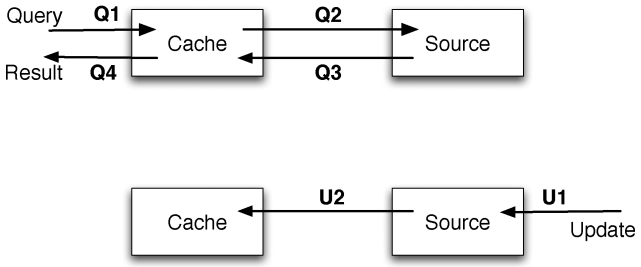


Fig. 5.2 Query processing under proactive and reactive caching.

- What happens when the cache fills up and we need to make room for new data? This is the cache replacement issue, we discuss it in Section 5.5.
- In the case of proactive caching, how do we predict which data items to cache, in anticipation of future requests? This is the selection issue, which we discussed in Section 3.3.
- What happens if the cached data can partially fulfill the user query? We discussed this in Section 4.2.

5.1.1 [Maintain::CacheInitialization] Proactive

View materialization at any of the three tiers of a server falls into the category of proactive caching [8, 104, 108, 118, 156, 173]. All views to materialize are defined when the cache is deployed, and data in a materialized view are continually updated over time.

5.1.2 [Maintain::CacheInitialization] Reactive

The prevalent work in the literature on caching dynamic web contents is reactive. The web objects [12, 26, 29, 43, 123, 136, 151, 172] or query results [9, 10, 37, 110, 119, 120, 121, 122] are obtained from the server when and only when there is a cache miss.

5.2 [Maintain] Timing for Updates

While caches provide low-access latency, cache coherency must be maintained for them to be useful. In other words, the cached objects must be updated so that the difference between them and the original objects at data sources does not exceed a threshold, which indicates the maximum staleness degree that clients can tolerate. Given that the source data are frequently updated and there are bandwidth constraints, the problem of effectively disseminating updates to maintain the cache as close (in value) to the source data as possible is called the *cache coherency* or the *cache consistency* problem. There are two "traditional" consistency models in the web caching community, *strong consistency* where no stale data from the cache can be provided after the modification completes at data sources, and *weak consistency*

where stale data can be returned to users as valid results. Note that both strong and weak consistency is used for web data, and have no database transaction semantics.

In order to maintain cache consistency, a lot of update dissemination mechanisms have been investigated. Two most widely used schemes are (1) data sources *push* updates to caches and (2) clients *pull* updates from data sources. Another simple scheme is to set a Time-to-live (TTL) value for each cached object, to mark whether this cached object is valid or not. This TTL scheme is usually used in combination with polling. In addition to push, pull, and TTL, there are some other schemes including piggyback validation/invalidation, leases and its two variants, adaptive leases and volume leases. We discuss these schemes (summarized in Table 5.1) in detail next.

5.2.1 [Maintain::UpdateTiming] Cache Invalidation — Push

To maintain strong cache consistency, an intuitive and important approach is cache invalidation (i.e., push) [127, 115, 28]. In this approach, update dissemination is driven by the servers. A server keeps track of all the clients that ever accessed, and hence possibly have cached, an object. Upon detecting the modifications on that object, the server sends invalidation messages or the updated object to all those clients.

Regarding what to send, there are two alternatives, invalidation-only and updates. Under the invalidation-only scheme, only notification of changes will be sent out, and the actual data/objects will be fetched

Table 5.1. Timing of updates to cached/materialized data items — Summary of options.

Maintain::UpdateTiming	
Cache Invalidation — Push	Invalidation [127, 115, 28, 48], DBCache [8, 118], MTCache [108]
	Invalidation in wireless environments [16, 89]
Cache Validation — Pull	Alex Protocol [32, 73]
Time-to-Live	Fixed TTL [170, 80], Adaptive TTL [73, 128]
Piggybacking	PCV [99], PSI [100], Combination [40, 101]
Leases	Leases [63], Adaptive leases [53], Volume leases [40, 112, 174, 175]

from the data source with the next request. Under the update scheme, the updated object will be sent without the need for further requests. The cost of the invalidation-only scheme is on control messages: at least one acknowledgment (ACK) message for each invalidation has to be sent. Also, a write at the server will be delayed until all ACK messages are received. However, since the objects themselves are much larger than ACK messages in most cases, the update scheme costs much more bandwidth than invalidation-only. The overhead will be paid off only if strong consistency is required and both the frequency of updates and data accesses are high. One hybrid approach is to send updates for more time-sensitive data and invalidations for all others.

Under the push approach, the message overhead is well controlled since messages will be sent only if there are updates for an object. Under the assumption that the update rate is much less than the access rate, this scheme will not create unnecessary traffic in the network. However, it also has obvious limitations. First, the servers need to maintain per client state information for each object, and hence possibly a huge client list for each popular object. Such lists require a significant amount of memory space and processing overhead. Second, strong consistency is hard to achieve in the presence of network failures or node failures. The server has to delay all the writes whenever there is at least one client, with the object to be modified in its cache, inaccessible.

To share the server's load, another implementation alternative is to employ the publish/subscribe model, in which clients explicitly subscribe to servers which own data that the clients want. A server will then send invalidations only to those registered clients, which are active and interested in the updates [48].

Finally, for cases where the update rate is not always less than the access rate for all or just some of the data, it makes sense to have a selection process, similar to view selection in traditional databases/data warehouses (as mentioned in Section 3.3).

5.2.2 [Maintain::UpdateTiming] Cache Validation — Pull (or Poll)

Another fundamental approach to maintain cache coherency is through cache validation (i.e., client-driven data polling). Instead of servers

sending invalidation messages to clients, clients periodically send validation messages with "If-Modified-Since" headers to data sources, and verify if the cached objects have been modified since last polled.

Under this approach, no state information needs to be maintained at the server side. All that the server has to do is to respond to clients' requests. However, this approach may incur a large amount of unnecessary message overhead in case the objects change infrequently. Moreover, the data polling time will be counted as part of the query response time, in case the objects did change. Therefore, the result is either too many 304 ("Not Modified") responses, or longer user perceived response time.

To achieve good performance of the cache validation scheme, one critical question is how to determine the polling frequency. Polling too often leads to large message and latency overhead, and polling too infrequently causes high staleness degree of returned data. The two polling methods which are often used are synchronous validation and asynchronous validation.

Synchronous Validation: Synchronous validation is done at object request time. It is also called *polling-every-time*. Under this approach, each time a user sends a request for a cached object, the cache sends a validation message to the data source. The data source will send back a "Not Modified" response if the object hasn't been changed, or an up-to-date version if the object has been changed since last validation. In the polling-every-time approach, strong cache coherency is maintained at the cost of a large number of messages and a large processing delay.

Asynchronous Validation: To overcome the limitations of the synchronous validation, the asynchronous validation method makes clients periodically contact the data sources and proactively validate their cached objects without waiting for the object requests.

5.2.3 [Maintain::UpdateTiming] Push or Poll — Discussion

Given the two canonical cache consistency mechanisms, we need to investigate two related questions. First, do we need both push and

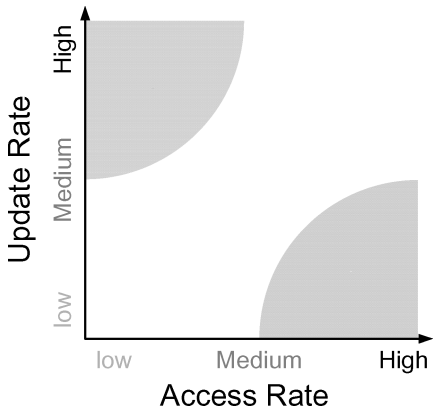


Fig. 5.3 Spectrum of access/update rate.

pull? Is there an all-time winner between the two? Second, who should make the decision about which one is the more appropriate approach?

First, the performance of both push and pull depends on the access rate and the update rate of cached objects (Figure 5.3). If frequently updated objects tend to be unpopular and have few accesses, then polling the server on accesses should incur the least overhead; the alternative, pushing from the server would have a prohibitively high number of messages. On the other hand, frequently accessed objects tend to be unchanged and have few updates, pushing is certainly a better strategy than pulling, as the server would only send messages when indeed there is an update (i.e., less often); in this case, the alternative, polling from the server would lead to too many unnecessary validations messages.

Since the correlation between objects' access frequency and update rate is of vital importance in choosing the cache coherency method, researchers studied the characteristic of web-related activities [18, 22, 52, 73, 169] and quantified this correlation by analyzing representative real traces. Surprisingly, inconsistent results were presented (see page 227 for more details).

5.2.4 [Maintain::UpdateTiming] Time-to-Live

Rather than push and pull, a simple approach we mentioned before is to assign a time-to-live (TTL) value, such as seven days or two hours, to each object in the cache. Within the TTL, the object is considered

valid and will be returned to the users directly. After the TTL elapses, the object is considered invalid, and the next request for this object will trigger the cache validation to pull an up-to-date version from the data source if it has been modified.

The TTL scheme is easy to implement in the HTTP protocol by using the "expires" or "max-age" header fields. The challenge lies in estimating the valid duration of each object, so that we can precisely set the time out values. If the duration is set too long, the probability to return a stale data item goes up; if it is set too short, the network cost and server load will be increased.

Explicit TTL: One simple alternative of the TTL approach is to assign an explicit TTL value for each object when the web developer creates the object.

Implicit TTL: In a lot of cases, an explicit TTL is missing, so that the client or the proxy cache has to resort to heuristics and make an a priori estimation as the implicit TTL value. It could be a fixed value, as in Worrell's thesis [170], that all documents are assigned the same TTL, or as in the WebExpress project [80] for mobile environments, which allows users to set a fixed TTL for all objects, but with the capability to change it for specific objects. Fixed TTL is a simple heuristic and easy to implement, however, it makes sense for different objects to have different TTLs [128]. The improved Alex protocol [73] which we mentioned earlier is also widely used to determine the TTL values, and is referred to as the adaptive TTL approach [28, 99, 165, 135].

5.2.5 [Maintain::UpdateTiming] Piggybacking

For both cache validation and cache invalidation, batch message processing and piggybacking can further improve performance on the network overhead and the server load. The main idea is that the cache will embed additional "if-modified-since" messages on every chance of communicating with the server. Given that this scheme is mostly suited for files, we describe it in detail in Section 8.

5.2.6 [Maintain::UpdateTiming] Leases

The lease approach maintains strong consistency as push does, as long as the lease is valid. Meanwhile, it is more resilient to network failures compared with push. Under a push-based protocol, when a network failure happens and the server cannot connect to one or more clients for invalidation, it must wait, potentially indefinitely, to finish writes to the cached object. In contrast, the server in the lease approach only delays writes until unreachable clients' leases expire, which also frees servers from waiting for idle clients before modifications. Another advantage of the leases approach is that it improves scalability. Instead of pushing invalidations/modifications to all the clients with the cached objects, the server now only takes care of those active clients who hold valid leases. The state information maintained at servers is also reduced.

The lease approach is apparently a hybrid method of push and pull. As the lease duration becomes long enough (infinite), it becomes pure push; and as the lease duration becomes zero, it is equivalent to polling. As such, it combines advantages from both push and pull, and adapts to environmental change very well. However, it leaves an important and tough question to answer, namely how to determine the lease duration. Given that this scheme is mostly suited for files, we describe the different alternatives in detail in Section 8.

5.3 [Maintain] Processing of Updates

Besides the timing of updates, another closely related aspect is the processing of updates. In the early stage of the Internet, static web pages were dominant. All the elements in the web cache were static HTML pages. The update process was quite simple, retrieving the updated HTML pages from the server. Nowadays, dynamic web pages and database-driven web pages are pervasive. This brings us two possible update processing schemes: recomputation and incremental updates.

Cached objects can be divided into two classes: base data items and derived composite data items. Base data items refer to the static HTML web pages, HTML fragments, and base data coming from back-end databases. Derived composite data items include everything constructed from base data items, such as a dynamic web page composed

of multiple fragments, e.g., the S&P 500 stock index calculated from the price of the 500 U.S. stocks, or a materialized view from the back-end database.

For base data items, an update is simply equivalent to writing a new version into the cache. However, for derived composite data items, the update processing strategy can be classified as *recomputation*, i.e., recalculating from scratch every time the composite data are queried; or as *incremental updating*, which incrementally modifies the cached objects and always builds the up-to-date version from the previous one.

In this section, we will first discuss preprocessing techniques, then elaborate on the two update processing alternatives, recomputation and incremental updates, respectively.

5.3.1 [Maintain::UpdateProcessing] Preprocessing

Before processing an update, some approaches determine either which cached objects are affected by the update [85], or whether the update has any effect on the derived relations [20, 21, 56, 111].

If we consider a database-driven website, then we assume a generalized "architecture" where each web page is composed of fragments, many of which are generated using data from a database. As such, without loss of generality, we can assume that a web page is the root node in a tree, where each subtree represents the generation "path" for a particular fragment rooted at the top node of the subtree. This, in turn, implies that there is a hierarchy for the generation of the page, with the bottom nodes of this hierarchy essentially being data drawn from the database (which we refer to as base data). In between nodes can either be HTML/XML fragments or database views [104, 106]. Given this "architecture", we trivially see that not all updates to base data would cause a web page that has been cached to become stale. As such, having a technique to determine if a certain update would cause a cached object (in the above hierarchy) to become stale can greatly increase the efficiency of the caching/materialization scheme used. There are two classes of techniques to address this. The first class essentially looks at the cached objects as different files and using a graph model tracks the dependencies between them (e.g., [85]). The second class is inspired by the work in view maintenance (e.g., [21]).

Given that all these techniques have not been specifically designed for web databases, we present additional details in Section 8.

5.3.2 [Maintain::UpdateProcessing] Recomputation

There are many cases where recomputation is the best choice. First of all, recomputation may be the only available choice; this depends on the unit of caching and the capability level of the cache (as was explained in the previous section). Secondly, the availability of additional storage for auxiliary objects or intermediate results is also crucial for most incremental maintenance solutions. Thirdly, the characteristics of the workload play a crucial role in determining whether recomputation or incremental maintenance is the best option. A high update rate of base data compared to the rate of access would make recomputation the most favorable update processing option (assuming of course that it is beneficial to cache/materialize), as many incremental updates would be computed unnecessarily. Conversely, a high access rate combined with a low update rate suggests that the incremental maintenance solution would be best.

Web databases for supporting stock website or trading applications are a good example to explore further. The NYSE workload had an update rate of up to 696 updates per second during peek time back in 2000 [105], and the update workload was at least three times more intensive than the query workload. As a result, recomputation is widely used in such web databases [4, 105].

5.3.3 [Maintain::UpdateProcessing] Incremental Updates

Recomputation is acceptable when updates are intensive and queries are much less frequent. However, it might be prohibitively expensive when queries are frequent and recomputation of the derived data has a high cost. For maintaining views, Hanson [76] clearly shows that when views are simple (selection/projection) and indexes are available, recomputation outperforms incremental updates in most cases. However, when joins are involved, recomputation becomes very expensive, and incremental updates win most of the time.

To perform incremental updates one needs to determine which part of the composite data was affected by the update and only refresh that part (i.e., to build the up-to-date object from the last snapshot and the current update only). Such incremental updates are called *autonomously computable* in [20]. If an update is autonomously computable, the relevant instance can be calculated by the current instance and the update only. No additional data from the base relations are required. Blakeley et al. [20] provide sufficient and necessary conditions to determine if an update is autonomously computable.

5.4 [Maintain] Scheduling of Updates

Having discussed the approaches of updating the cached objects, we summarize the literature on the update scheduling problem. Since cached objects are frequently updated by their data sources, to keep every cached object up-to-date is hard to achieve, given the bandwidth constraint and the size of the cached objects. The cache synchronization problem is fully defined as determining which objects in the cache will be refreshed and in what order.

Labrinidis and Roussopoulos [106] study how to propagate updates on database relations to WebViews affected by these updates in a way that maximizes the QoD of the views. WebViews are HTML page fragments that typically contain data derived from a database. Their goal is to maximize the aggregated freshness of the database. Toward this goal, they propose a QoD-aware update scheduling algorithm (QoDA) that considers both updates to base tables and materialized views, and is adaptive and tolerates surges in the update stream. The algorithm maintains a set of tables or views that are currently stale and selects every time to refresh the object that would have the greatest negative impact on the overall QoD, if it was not refreshed. The two issues that make this work different are (1) view dependencies have to be considered, and (2) popularity of the objects is taken into account during scheduling. The authors demonstrate that QoDA consistently outperforms FIFO by up to two orders of magnitude on aggregated QoD of the entire database.

Best-effort Cache Synchronization: Olston and Widom [129] study synchronizing cached objects with the original objects at the data sources, in the presence of bandwidth constraints. The synchronization is performed based on source cooperation. The authors propose a best-effort synchronization algorithm that adaptively sets the update threshold for each object based on the current bandwidth situation.

Database Synchronization: Cho and Garcia-Molina [38] assume that the local database is updated uniformly over time. Then given the frequency we synchronize the local database, several important questions are answered based upon their theoretical proof. First, should we synchronize all objects at the same rate, or should we synchronize them with different rates that are proportional to their frequency of changes? Surprisingly, they demonstrate that the uniform allocation policy, which synchronizes all elements at the same rate, is always better than the proportional allocation policy, and that we can always calculate the optimal resource allocation by using the *method of Lagrange multipliers*. Second, given the synchronization frequency and resource allocation, we need to decide in what order we update the objects in the local database. There are three alternatives: (1) fixed order, in which we update all elements in the same order repeatedly; (2) random order, in which we update them repeatedly, but randomly select an element to start with in each iteration; and (3) purely random order, in which we arbitrarily select any object to update no matter when it was updated last time. It turns out that the fixed order performs the best among all. By following the uniform resource allocation policy and the fixed order, the optimal synchronization policy is achieved to significantly improve the freshness of the local cache.

There is additional related work in closely related areas; we summarize it in Section 8.

5.5 [Maintain] Cache Replacement Policies

Most of the existing work on web caching for dynamic data does not consider the issue of cache replacement [8, 12, 26, 29, 108, 118, 120, 136, 140, 141, 151, 172, 173]. Many others adopt the traditional Least

Recently Used (LRU) [43, 119, 121, 122, 139, 140, 157] and Least Frequently Used (LFU) policies [139, 157] for cache replacement. In these policies, when the cache size becomes inadequate, the cache unit that has not been used for the longest time (LRU), or the cache unit that has the least number of accesses (LFU), will be evicted from the cache.

Finally, a few approaches adopt a benefit-based policy for cache replacement [9, 123]. The policy deletes the cache unit that has the least utility. The benefit of a cache unit is usually evaluated based on the per-bit utilization of answering queries.

Given that most of the work in cache replacement is done in the context of "traditional" web caching (i.e., for static files) or of distributed systems, we present additional details in Section 8.

6

Metrics Performance and Quality Metrics

In order to evaluate the effectiveness and quality of a web caching/ materialization strategy one can employ three types of metrics:

- standard Quality of Service (QoS) metrics;
- Quality of Data (QoD) metrics, which are important for dynamic data; and
- user-centric approaches, which measure user satisfaction and typically consider both QoS and QoD.

In the following subsections we present these metrics in detail. We conclude the section with a short presentation on Service Level Agreements.

6.1 [Metrics] QoS Metrics

The Quality of Service (QoS) experienced by web users has gained great attention in recent years. It is of vital importance for all kinds of web service, e-business, online financial service, and more important, for the emerging streaming applications, i.e., audio/video services. QoS is becoming a dominant factor of any Internet-based web service [41].

The most widely used QoS metrics include user perceived response time, data availability, and system throughput.

6.1.1 [Metrics::QoS] Response Time

The key functionality of web caching is to distribute copies of popular objects from web servers to locations closer to the users. Thus user perceived response time, the latency between a user issuing a request and receiving a response back, is the most important measurement of a web caching system. The smaller response time, the better. Typically, we are interested in the end-to-end response time, which includes the "transit" time of the results (i.e., the time it takes the results to reach the end-user's computer from the server). However, some works focus only on the response time at the server (i.e., with essentially zero network transmission cost) in order to isolate and study just the server's behavior.

If somehow a user's request is associated with a response time deadline (equivalent to that of a soft real-time system), then one additional QoS metric is that of *tardiness*. Tardiness measures the amount of "deviation" from the deadline; if the result to a user request is delivered past the deadline, the tardiness is the amount of time past the deadline. However, if the result to a user request is delivered before the deadline, then the tardiness is zero (i.e., no benefit in being early).

Another QoS metric that is based on response time is *slowdown*. Slowdown measures the ratio of the actual execution time of serving a user request over the "ideal" execution time of serving the user request, i.e., the time it would take to execute it if it were the only request in the system. Given this definition, slowdown can be greater or equal to 1.0. The closer the slowdown is to 1.0 (the ideal value) the better it is. One disadvantage of using slowdown is the requirement to know ahead of time the "ideal" execution time. One big advantage of slowdown is that it is a more "natural" metric to users, who can easily understand the meaning of a 20% slowdown of the processing of their requests.

Given a QoS metric (e.g., response time, tardiness, or slowdown), the question remains on what to optimize in terms of the values of this metric over multiple requests/user queries over time. In other words, *how to aggregate multiple measurements*. By far, the most typical

aggregator is the *average value*. Although the average value is easy to understand and to implement, it is not always the most representative aggregator function. In particular, just trying to optimize for the average value will often lead to *starvation*, where some of the users will face very high response times, although the overall average value would be quite low.

Another alternative for aggregating over multiple values is to consider the *minimum* or the *maximum* value, depending on whether lower or higher values are better for the metric we are considering. For example, trying to optimize (i.e., minimize) for the maximum response time is essentially trying to make the worst case as good as possible, thus eliminating the starvation problem. Of course, this can happen at the expense of the average case.

An alternative that balances between the worst case and the average case was proposed in [149] for scheduling in data stream management systems. The main idea is to use the L2 norm of the response time (or slowdown), which is defined as follows:

Definition 6.1. The ℓ_2 norm of response times for N requests is equal to $\sqrt{\sum_1^N R_i^2}$, where R_i is the response time of the i-th user request.

Using the L2 norm as the aggregator function was shown to balance the trade-off between optimizing for the average case and optimizing for the worst case.

6.1.2 [Metrics::QoS] Availability

The always-on characteristic of the Web is making *availability* a first-class citizen among performance metrics. Availability is used to measure whether the data present or the web service is ready for immediate use. The larger the value the larger the probability that the data and service are accessible at any given moment.

Availability can suffer because of congestion/overload conditions at two points: (a) at the server and (b) at the network path between the end-user and the server. Furthermore, such conditions can occur

because of three reasons:

- Increased "legitimate" load, because of the success/popularity of a website, which is often referred to as a flash crowd or the slashdot effect. In some cases the high load can occur because of high update volume, which overwhelms the database driving the website.
- Unintentional software problems/bugs that introduce extra delays and/or waste system resources (e.g., memory leaks).
- Intentional efforts to sabotage the web server (or the network path that leads to it), for example through a (distributed) denial of service attack.

Caching and materialization helps mitigate such congestion and increase availability, avoiding the "traditional" single point of failure problem. As such, availability increases by replicating data across different locations, however, on the other hand, such replication brings consistency/freshness issues, essentially creating a trade-off, as we illustrate later in this section.

Associated with availability is the time-to-repair (TTR) metric. TTR represents the time it takes to get a data item or service back online. The smaller the TTR value, the better.

6.1.3 [Metrics::QoS] Throughput

Although end-users do not case about throughput (or ever see it), throughput is an important system performance metric, that measures how efficiently the current environment is processing user requests. Throughput is used to assess how many data or web service requests can be handled by a system in a given time period. Web caching/materialization can be employed at the web server to alleviate congestion (especially during peak times) thus improving the servers' scalability/throughput [104].

6.2 [Metrics] QoD Metrics

Quality of Data (QoD) metrics are used to evaluate how "good" the served data are. Goodness of data can be measured in freshness,

accuracy, and other metrics. The definition heavily depends on the semantics and requirements of the application.

6.2.1 [Metrics::QoD] Freshness

When we cache data locally, a portion of the cached objects may get temporarily out-of-date, due to the delay between source data updates and the refresh of the local copy. In general, data freshness measures how fresh cached data are compared with its up-to-date version (at the source). There are multiple definitions for data freshness, which we explore next.

In Squirrel [86], a binary value is assumed as the freshness function. Opposite to the freshness degree, Iyengar and Challenger [85] and Dingle and Partl [50] define staleness degree to assess how obsolete the object is. In [85], one of the functionality of the Data Update Propagation (DUP) algorithm is to calculate the staleness degree for each object. They use a weighted graph to describe the object dependencies. At what degree a version of an object is obsolete is determined from the sum of the weights of edges from the object node n_1 to another node n_2, where the object is consistent with the latest version of n_2. The obsoleteness degree is compared with an explicitly specified threshold value, if it falls below the value, the object stays, otherwise it will be invalidated or replaced with a up-to-date version.

Unlike [50, 85, 86], which define freshness/obsoleteness degree as a per-object value, both Adelberg et al. [4] and Cho and Garcia-Molina [38] emphasize a more "global" view and define freshness as the fraction of the database that is up-to-date. Let $\mathcal{S} = \{e_1, \ldots, e_N\}$ be a database of N elements. The freshness of an element e_i at time t is:

$$F(e_i;t) = \begin{cases} 1 & \text{if } e_i \text{ is up-to-date at time t} \\ 0 & \text{otherwise.} \end{cases}$$

Then the freshness of database $\mathcal{S}$ at time t is

$$F(\mathcal{S};t) = \frac{1}{N} \sum_{i=1}^{n} F(e_i;t), \tag{6.1}$$

and the freshness averaged over time $\bar{F}(\mathcal{S})$ as:

$$\bar{F}(\mathcal{S}) = \lim_{t\to\infty} \frac{1}{t} \int_0^t F(\mathcal{S};t)dt.$$

Labrinidis and Roussopoulos [105] improved the freshness metric by taking object popularity into account. Their workload study found that both the query and update distributions are highly skewed, which well supports the importance of including popularity into the freshness metric. As a result, they define the overall freshness degree as the popularity-weighted sum of the freshness probabilities of all the objects in the database, including views. As in [4, 38], the freshness function for object d_i is defined as

$$b_{fresh}(d_i)^t = \begin{cases} 0 & \text{if object } d_i \text{ is stale at time t} \\ 1 & \text{if } d_i \text{ is not stale at time t} \end{cases}$$

The freshness probability for a view object v, $p_{fresh}(v)$, is defined as the probability of accessing a fresh version of v during interval T,

$$p_{fresh}(v) = \frac{1}{T} \times \int_{t_i}^{t_j} b_{fresh}(d_i)^t.$$

Assume $f_a(v_i)$ is the access frequency of view v_i, then the freshness probability of the database $p_{fresh}(db)$ is:

$$p_{fresh}(db) = \sum_{v_i \in \mathcal{V}} f_a(v_i) \times p_{fresh}(v_i). \tag{6.2}$$

Olston and Widom [129] further imported another attribute, the importance of an object, into the freshness metric. They define importance function as $\mathcal{I}(\mathcal{O},t)$ for object $\mathcal{O}$ at time t, popularity function as $\mathcal{P}(\mathcal{O},t)$, and the overall weight $\mathcal{W}(\mathcal{O},t)$ is defined as:

$$\mathcal{W}(\mathcal{O},t) = \mathcal{I}(\mathcal{O},t) \times \mathcal{P}(\mathcal{O},t).$$

6.2.2 [Metrics::QoD] Accuracy

In addition to freshness, which is based on time, another metric to assess quality of data is deviation of the value, or its accuracy [19, 129, 148]. Accuracy is defined as:

$$\mathcal{D}_v(\mathcal{O},t) = \Delta(\mathcal{V}(\mathcal{O},t), \mathcal{V}(C(O),t)),$$

where $C(O)$ is the cached copy of object O, $V(O)$ represents the value of object O at time t, and $\Delta(V_1, V_2)$ can be any function quantifying the difference between two versions of an object. In [19] and [148], the authors use $|\mathcal{V}(\mathcal{O},t) - \mathcal{V}(C(O),t)|$ to measure the coherence between two versions. They use the temporal coherency requirements to guide the process of update dissemination.

Although the accuracy metric is "perfect" for quantifying how different a cached data item is when compared to its source, the metric's applicability is somewhat limited because it is not always easy to construct $\Delta()$ functions to measure the difference of the two versions.

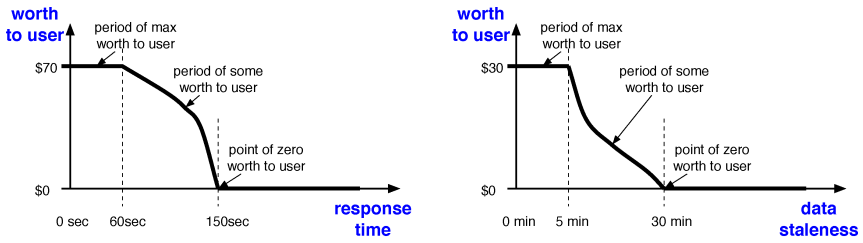
6.2.3 [Metrics::QoD] Lag

Lag is defined in [129] as $\mathcal{D}_l(\mathcal{O},t) = u$, when $C(O)$, the cached version of O, is u updates behind O. This means that O has been updated u times since the last refresh. It is also called number of *Unapplied Updates (UU)* in [4], which is optimistic and assumes that the object is always fresh unless there are updates received but not applied yet. In some cases the object freshness is not easy to estimate, so unapplied updates is a good alternative that systems can accurately measure.

6.3 [Metrics] Quality Contracts

Having visited current approaches to Quality of Service (QoS) and Quality of Data (QoD), Labrinidis et al. [103] identify that the most important limitation is that they do not have strong *support for user preferences*. In practice, it is beneficial to have users supply their *preferences* on how the system should balance the trade-off between QoS and QoD, in other words, instruct the system on how to best allocate resources in order to maximize user satisfaction. Toward this goal, Labrinidis et al. proposed *Quality Contracts* (QCs) [103], a framework based on the micro-economic paradigm, that provides an intuitive and powerful way for users to specify preferences for QoS and QoD.

In the QC framework, users are allocated virtual money, which they spend to execute their queries. The Quality Contract (QC) essentially specifies how much money a user is wiling to pay. The amount of money the server receives in the end (i.e., the system profit) will depend on



(a) QoS graph ($0 to $70) (b) QoD graph ($0 to $30)

Fig. 6.1 **QC example:** The QoS metric is response time, whereas the QoD metric is data staleness. Data staleness is measured as the time between the last instant when the physical world has changed and the instant when the local storage has been updated (i.e., time since the last update on a data item access by the query).

how well it executes the user's query. In this model, servers try to maximize their income, whereas users try to "stretch" their budget to run successfully as many queries as they can.

A Quality Contract (QC) is essentially a collection of graphs representing QoS/QoD requirements from the user. Figure 6.1 is an example for an ad hoc query submitted by a user. This QC consists of two graphs: a QoS graph (Figure 6.1a) and a QoD graph (Figure 6.1b). This example illustrates the following salient features of QCs. First, *QCs allow users to combine different aspects of quality*, as the user has expressed preferences for both QoS and QoD. Second, *users can specify the relative importance of each component of the overall quality by allocating the query budget accordingly.* In this example, $70 are allocated for optimal QoS, whereas $30 are allocated for optimal QoD which is less important than QoS. Third, *users can easily specify the relative importance of each query by allocating their budgets accordingly.*

Quality Contracts were inspired by prior work that utilized microeconomic models for resource allocation (e.g., Mariposa [154] and the work by Ferguson et al. [57]). Recent work by Florescu and Kossmann advocates the use of *real cost* (i.e., an economic model) as one of the system optimization objectives.

6.4 [Metrics] Service Level Agreements

A Service Level Agreement (SLA) is a formal definition of the business relationship between a service provider and its customer. It specifies the

mutual understandings about what the customer could expect from the service provider, the obligations of both customer and provider, how well the service needs to be executed and the procedures to be followed. The typical components of an SLA include description of the service and parties, the validation period, the expected service level objectives, the procedures to follow including monitoring and reporting, the penalty for the service provider not fulfilling the obligation, and constraints and exclusions.

Verma [163] provides an overview of Service Level Agreements in the network management domain. Within the context of the computer networks, especially IP networks, SLAs are typically provided for three type of services, network connectivity services that provide customer access to the network or connect them to each other as an intranet; hosting services that provides web servers to operate websites; and the integrated services of the connectivity and hosting. Verma also identified three different models that are used to support SLAs: *the insurance approach*, *the provisioning approach*, and *the adaptive approach*. The insurance approach is the most commonly used one in industry. In this approach, the service provider offers the same level of service to all customers (in a best-effort mode) and periodically modifies service level objectives. In the provisioning approach, the service provider negotiates different types of service level objectives with different customers. Different system configurations need to be made and resources will be allocated according to a customer's different needs. The last approach is the adaptive approach that dynamically modifies the system configurations for different customers. Verma [163] then describes how these three different approaches are used in each of the three different service environments, and the advantages of moving from the insurance approach to more dynamic ones.

SLAs can be viewed as legal documents and, as such, original SLAs are described in natural language. There has been a lot of effort to make the entire procedure, definition, negotiation, deployment, monitoring and enhancement, an automated process. The first step is the SLA templates [142] for a service provisioning system. It includes several automatically processed fields in the natural language written SLA. The limitation is that it is only suitable for a small set of SLAs that

provides the same type of service and use the same Quality of Service (QoS) parameters. Additionally, the offer is hard to be changed over the time. To facilitate automatic SLA process, many companies presented their own Web Service platforms.

Jin et al. [88] propose a service composition model to capture the composition relationships between service providers and customers and help making decisions on the creation stage of SLAs. In this model, a web service is composed of a set of operations. Each operation is implemented by a sequence of activities, where an SLA is attached to. The SLA itself is modeled as a distribution of a Quality of Service metric. By using Business Process Simulation Environment (BPSE) as simulation tool and focus on the HP Process Manager system, they perform a service level sensitivity analysis, identify the impact of changing suppliers and/or their SLAs on the service provider's capability of fulfilling their obligations. Their simulation results suggest that having information of impact of various service levels is important for defining service level objectives in the creation stage of SLAs.

Negotiation of SLAs is very important for maintaining Quality of Service. To facilitate automated negotiations of SLAs for web services in a Service-Oriented Architecture (SOA), a lot of different models and approaches were proposed. Su et al. [155] propose a negotiation server to perform bargain-type negotiation automatically. The negotiation server uses constraint satisfaction, rule-based conflict resolution, and event systems to conduct negotiations after goals and requirements are registered by both parities. Hung et al. [84] propose a Web Service Negotiation language including negotiation message, protocol, strategy and a framework the language can be used. Li et al. [113] extend Su's negotiation server, and propose an automated negotiation framework based on a finite state automata and a set of negotiation protocols. Cappiello et al. [30], Chen et al. [36], and Zulkernine et al. [179] all propose policy-based Negotiation Broker (NB) framework. The high-level business goals, the negotiation parameters including preferences, constraints, and values are expressed as a policy specification by each of the negotiating parties. The NB then maps the policy specifications to low-level negotiation strategy models and parameters. The broker then conducts negotiations automatically. The differences are that [30, 36]

require negotiating parties to have knowledge about strategy models to input their choice of strategy and parameters in the specification, whereas Zulkernine et al. [179] lift this requirement and conduct the mapping automatically.

IBM presented its Web Services framework [97] in 2001. Keller and Ludwig [93] then proposed Web Service Level Agreement (WSLA) framework for specifying and monitoring SLAs for Web Services. The framework contains a language and a run-time architecture comprising several SLA monitoring services. The language defines a type system based on XML schema, allows both parties to define the service description including service operations, SLA parameters and metrics, and the obligations including service level objectives and action guarantees. After the specification is submitted, the WSLA monitoring services are automatically configured to enforce the SLA. The principles, SLA establishment scenarios, lessons, and design goals are documented in [93]. A Java-implemented WSLA framework, the SLA Compliance Monitor, has been published as part of the IBM Web Services Toolkit.

7

Projects

In this section, we review representative projects on caching for web databases. The projects are clustered based on the location of caching/materialization, i.e., browser caches, proxy caches, and server-side caches (Table 7.1).

7.1 Browser Caches

In comparison with the large number of publications on server-side or proxy caching, there is less existing work on client-side caching for web databases. One reason is that client-side caching benefits only clients and browser caches are mostly sufficient for the purpose.

Rabinovich et al. [136] propose an approach to construct dynamic web pages at the browser rather than at the proxy. Their approach uses the *Edge Side Includes* (ESI) [54] markup language to assemble web fragments stored at a client cache into pages. We will describe ESI in more detail later in the related work on industrial products and standards.

Xiao et al. [172] study the sharing of cache contents between a number of clients and their proxy. The authors design a P2P cache management scheme that exploits data locality and reduces document duplicates in the group of browser caches.

Table 7.1. Classification of projects according to location of cache.

Store::Location	
Server	db cache: SkyServer [156], WebView [104] mid-tier cache: DBCache [8, 118], MTCache [108] web server cache: WebView [104, 105, 107] combination: Server Farm [26], Weave [173]
Reverse Proxy	single cache: CachePortal [26], DBProxy [9, 10], Oracle Web Cache [12] multi-cache: Cascaded Caching [157], Cooperative Caching [138, 139, 140, 141]
Proxy	Active Caching [29, 120], Bypass Caching [123, 124], DCCP [151], Dynamic Content Caching [43], Form-Based Proxy [119, 121, 122]
Client	Browsers-Aware Caching [172], CSI [136]

There has been renewed interest in caching at the client side lately. First of all, the upcoming new HTML 5.0 standard[1] introduces new caching capabilities, in general, and, in particular, *application caches*.[2] This feature should considerably increase the caching options available to the end-user.

The proliferation of AJAX brought upon JavaScript libraries that support enhanced caching capabilities at the client side. The most notable of these is the *Google Gears* framework[3] that was released in 2007. Gears supports among other things a database module that can store data locally (as part of the client-side part of a web application) and a module that caches and serves application resources (HTML, JavaScript, images, etc).[4]

7.2 Proxy Caches

Proxy caches have been widely used throughout the Internet to improve client response time and reduce server workload. Specifically for web databases, we review several representative proxy caching frameworks, including the active proxy [120], the form-based proxy [119, 121, 122], CachePortal [26], a page fragment-based caching proxy [43], DBProxy [9], the bypass caching [123], and the Oracle Web Cache.

[1] http://en.wikipedia.org/wiki/HTML_5
[2] http://www.w3.org/TR/html5/offline.html#appcache
[3] http://en.wikipedia.org/wiki/Google_Gears
[4] http://gears.google.com/

The DCCP protocol proposed by Smith et al. [151] utilizes user-specified equivalence between different documents to accelerate dynamic web caching at proxies. In the *Active Cache* scheme [29], the server provides cache applets along with corresponding web documents to store in the proxy cache. These applets are invoked upon cache hits to dynamically process the documents when necessary, reducing communication to the server. Luo et al. [120] extend cache applets to query applets that enable active caching of SQL query results. This active caching of SQL query results is done through checking query containment between a new query and a cached query and if applicable, evaluating the new query over the results of a cached query. The authors identify sufficient but not necessary conditions for polynomial-time containment checking between queries to achieve efficient query processing at the proxy.

The *form-based proxy* [119, 121, 122] also investigates containment-based query processing as a variant of its active caching scheme. In the form-based proxy caching framework, the authors propose active caching techniques for two common classes of queries issued from HTML forms to database-backed websites: *keyword-based queries* and *function-embedded queries*. The first class of queries contains keyword search predicates and the second embeds calls to table-valued functions. The form-based proxy works with query and function templates that describe high-level semantics of the queries issued from HTML forms. By extensive experiments using both real and synthetic query traces, the authors demonstrate that the proposed active query caching largely outperforms traditional, exact match-based passive caching.

CachePortal [26] is a reverse proxy caching framework proposed by Candan et al. for database-backed websites. The proxies in the framework are ordinary web caching proxy servers. The two main components in the system are the sniffer and the invalidator at the server side. The sniffer records HTTP requests received as well as database queries generated by the application server, and creates mapping between the HTTP requests and the database queries. The invalidator identifies the queries whose results are affected by a database update and removes all cached pages whose fragments are generated from these query results. CachePortal uses several commercial products in the

system deployment and evaluation, including an Oracle DBMS and a BEA web and application server.

Datta et al. [43] present a proxy implementation for dynamic web caching. The system can be configured to run either as a reverse proxy or as a proxy. The proxy stores various web fragments and composes the page to be returned for an HTTP request using these cached fragments given the layout information, which can be dynamically determined by the server. In other words, both page fragments and the layout of the page can be dynamically generated and updated. Missing fragments for a request are shipped from the server together with the layout.

DBProxy [9] is a semantic data cache at the edge server. Results of SQL queries are stored as materialized views in the cache, independent from the original database schema. The list of queries whose results are currently stored is kept in a cache index. The system uses template-based query containment checking algorithms to answer a new query using the cached results of previous queries [10]. The query templates are automatically inferred from similarities between predicates of multiple queries. These template-based algorithms achieve a higher efficiency than general containment checking algorithms [109] by taking advantage of the semantic restrictions of query templates. As a result, they are scalable to a large number of query predicates within each template.

In DBProxy [9], the query results in the cache are reactively loaded at the query time. These query results are stored in a local database whose schema is dynamically changing with the query workload. If prior knowledge of the workload is available, an initial schema can be set proactively for the cache database to reduce subsequent schema modifications. Updates on the original tables at the database server are propagated to the cache by a consistency protocol based on update subscription. In addition, a background garbage collection process is used to remove cached tuples that do not belong to any cached queries.

Malik et al. [123] present a *bypass proxy caching framework* for scientific database federations. Their framework focuses on conserving global wide-area network bandwidth in the federation rather than local response time. This focus is for scientific databases to be a good citizen in the public Internet.

A bypass cache loads and evicts database objects according to their expected yields. The yield of an object is estimated using the result sizes of the queries that can be answered using the object, i.e., the saving of network traffic achieved by caching the object. The rate of network traffic saving achieved by the queries answered using the object during its lifetime in the cache is recorded in the rate profile of the object. The authors develop three algorithms for the bypass cache management: (i) a rate-profile algorithm that uses previous queries to predict future workload patterns, (ii) an online algorithm that has theoretical performance guarantees and assumes no query workload patterns, and (iii) a randomized algorithm with minimal space requirement. Subsequently, the authors investigate how to estimate the query result sizes required for yield computation [124]. The estimation is adaptive, based on statistical learning techniques, including classification and regression, over query templates.

In bypass caching, the authors experimentally demonstrate that query containment relationships are infrequent in scientific workloads, and thus, semantic query processing is ineffective in such scenario. As a result, table-level rather than query-level caching is employed in a bypass cache. The tables in bypass caching are dynamically inserted into or deleted from a cache according to their yield estimation upon the current query workload. The evaluation of a query at a bypass cache is either completely local or completely remote. This decision is made based on the yield estimation of database objects involved in query processing.

Oracle Web Cache [12] is an industrial reverse proxy cache product. It performs page-level caching. It employs several techniques to support dynamic web caching, including name disambiguation of cache contents, intelligent session state management for users, fragment caching for personalization, and heuristic invalidation for consistency maintenance. Additionally, it manages cache consistency by setting different granularities of invalidation options for the applications.

7.3 Server-Side Caches

Recall the server side of a web database includes the web server, the application server, the database server, and the server application.

In the following, we discuss projects that perform caching on various layers at the server.

For database-backed web servers whose contents rapidly change, Iyengar and Challenger [85] propose *Data Update Propagation (DUP)* to maintain data dependencies between cached objects (e.g., dynamic HTML web pages) and base data (e.g., fragments in the back-end database). They utilize *object dependency graphs* to map relations between base data and cached objects. They provide a metric to measure the obsoleteness of an object. Considering the object hierarchy and the obsoleteness degree, they decide when and which pages should be replaced.

Based on DUP, Challenger et al. [35] propose graph traversal algorithms to identify all cached objects that are affected by a database update. These objects are either invalidated or updated.

At the mid-tier of a database-backed website, database caches are often deployed to store database objects. Such caches are usually a DBMS clone with some extensions for caching purposes. This DBMS can either be the same instance as or a different one from the database server.

Two representatives of mid-tier database caches are *DBCache* [8, 118] and *MTCache* [108]. Both caches are industrial-strength implementations. DBCache is developed using the IBM DB2 database server, whereas MTCache uses the Microsoft SQL server.

Both DBCache [8, 118] and MTCache [108] employ table-level caching. The views in both caches are predefined by the Database Administrator (DBA) given the knowledge of the query workload. View definitions in DBCache can be dynamically changed based on the current query results. In comparison, in MTCache the entire database schema together with the statistics are replicated to the cache, with all cache tables initialized to be empty. Data in this shadow database are reactively loaded from the original database at runtime.

Both DBCache and MTCache can generate distributed query plans that efficiently combine local processing over cached data with remote processing over the original database data. In DBCache [118], the cache and the server are treated as a federated DB2 database. Altinel et al. [8] propose a two-headed query plan called a *Janus* plan. Such a plan

contains a probe query that will always be executed. The result of the probe query determines whether a local query or a remote query will be executed subsequently. In comparison, MTCache focuses on generating and optimizing dynamic plans for queries that have run-time instantiated parameter values [108].

Work on server-side materialization often employs built-in utilities of a commercial DBMS for consistency maintenance. As examples, DBCache [8, 118] uses the DPropR utility in DB2 for cache invalidation and MTCache [108] uses SQL Sever transactional replication to propagate database changes to cached tables or views.

Guo et al. [68] propose SQL language extensions to allow explicit specification of currency and consistency constraints for applications. The authors define rigorous semantics for these constraints and develop techniques to fully integrate the constraints into query optimization and execution. The authors also present a prototype implementation of the proposed constraints in MTCache [108].

Furthermore, Bernstein et al. [17] design a new Relaxed Currency (RC) model for transaction serializability in mid-tier caches. The model allows update transactions to read stale data that satisfies given freshness constraints. The authors present algorithms for constraint guarantee and prove the correctness of the proposed algorithms.

In comparison with general-purpose database caches, the *SkyServer* [156] website stores materialized views of astronomy data in SQL databases and utilizes materialized views to speed up the answering of many types of spatial queries originated from various astronomy web applications. The SkyServer website only updates its data by offline load processes [156].

A multi-tier server-side cache that combines all three types of the server-side caches is possible. For instance, Yagoub et al. [173] propose a declarative website specification that enables a three-tier materialization strategy. It stores HTML pages, XML fragments as well as materialized views. As a result, this approach is a mixture of query-level, fragment-level, and page-level caching. What content is cached in each of the three tiers and how the content is maintained upon various events and conditions are all specified by the users before the website is deployed. The processing of cached HTML pages or XML fragments

falls into the category of no query processing. In comparison, the processing of cached database data corresponds to full-fledged SQL query processing. Their experimental results show that such a combination of different materialization strategies is necessary to achieve the best performance.

Finally, *WebView materialization* [104, 105] has been shown an attractive solution to dynamic web caching. Labrinidis and Roussopoulos [104] study the strategy of saving query results that generate web page fragments inside the DBMS. The authors demonstrate that this materialization strategy is not as efficient as materialization at the web server, particularly if the queries are not complex. The authors propose a cost model that analytically evaluates three different materialization policies: no materialization, materialization at the database server or at the web server. The model considers both the processing parallelism at multiple servers and the performance impact of database updates. The model also helps select WebViews to be materialized.

8

Related Work in Other Areas

Caching and materialization have been long studied in the context of the "traditional" Web (i.e, for mostly static files), in database engines, in client–server and distributed databases, in distributed and peer-to-peer systems, and as part of commercial product offerings. In this section, we discuss work in these areas related to caching and materialization for web databases.

8.1 General Web Caching

8.1.1 Data Consistency and Freshness

Cao and Liu [28] compare the *Time-to-Live* (TTL) weak consistency method, which is widely used in the Internet, with two strong consistency methods, *polling-every-time* and *invalidation*. Their experiments show that although polling-every-time performs much worse than the other two, invalidation achieves a similar response time to TTL. Bright and Raschid [24] propose to employ user-provided profiles at the browser to specify the latency–recency data requirement for web applications.

Li et al. [114] proposed to adaptively select pages to cache, balancing the response time, and the invalidation frequency of the cached web pages. Given a system-specified freshness threshold, Li et al. propose a freshness-driven adaptive dynamic content caching scheme to assure that the delivered content is either fresh or not older than the threshold. The algorithm keeps watching the response time and the length of the invalidation cycle, which is the time taken to check the validity of all pages in the cache. If the response time is larger than the length of the invalidation cycle, the number of cached query types is increased to lower the response time. If the invalidation cycle is longer than the query response time, the number of cached query types is decreased to shorten the invalidation cycle. By making the response time close to the invalidation cycle in an equilibrium point, data freshness is assured.

Bhide et al. [19] propose that the proxy computes a *Time-to-Refresh (TTR)* attribute with each cached data item for the pull approach and registers a temporal coherency requirement with each cached data item for the push approach. Their argument is that the server–prediction approach requires previous history on relevant data, which are not suitable for web data that is highly dynamic and inherently unpredictable. They propose two technologies to combine push and pull based on a client's observation, *PaP (Push and Pull)* and *PoP(Push or Pull)*. They demonstrate that both of them meet the diverse temporal coherency requirements, and are resilient to failures, efficient and scalable as well.

8.1.2 Cache Replacement

Cao and Irani [27] present a cost-based document replacement scheme for proxy caching. The main idea behind this seminal paper is to consider both the size of the document and the "cost", namely the network delay, of retrieving the document again if it were not cached. Even though this work is geared toward static content, the idea has been widely applied in web caching in general, including caching of dynamic data.

Wolman [168] states in his PhD thesis that the cache replacement algorithm is relatively indifferent to web caching because in practice

a cache with a size of gigabytes is sufficient for all cacheable requests most of the time. None of the papers in our monograph show a strong impact of the cache replacement policy on the performance.

8.1.3 Cooperative Caching

Ramaswamy and Li [139] study cooperative caching of web documents based on the expiration times of individual caches in the group. The expiration times of cached data reveal the access contention in the cache; the authors propose a document placement scheme based on this concept. This scheme performs a global management of the total disk space of all caches and effectively reduces duplicates cached without performance degeneration. The authors further investigate the architecture design of a cache group for cooperative web caching [140]. In this work, they develop dynamic and hash-based protocols for document lookup and update within the group as well as a new utility-based scheme for document placement. Finally, the authors study the impact of automatic page fragmentation on web caching [138], and propose two efficient schemes to divide a set of caches into multiple cooperative groups to optimize the caching performance [141].

Tang and Chanson [157] formulate object placement for multiple websites over a cache group as an optimization problem. They propose a dynamic programming solution to solve the optimization problem, assuming the access frequency of every object in each cache is known a priori. Since all web objects to be cached need to be known before the caching decisions are made, this approach requires a strong server cooperation capability for the cache group.

8.1.4 Web Cache Updates

Banga et al. [15] propose *optimistic deltas* as incremental updates for latency reduction over slow networks. They put a layer of proxies on either end of a slow link. The server-side proxy optimistically sends data (which are possibly stale) to the client-side proxies during the idle time. Having transferred all the data just once, in the remaining correspondence, only a confirmation that the data are not modified or a delta, which is the change between the older version and the current

one, will be transferred. Banga et al. provide data analysis to support their assumption, that changes between two versions are relatively small in comparison to the actual web documents.

Mogul et al. [126] further quantify the benefits of delta encoding using real traces. Their results show that incremental updates provide significant improvements in the response size (i.e., network overhead) and the response delay (i.e., user-perceived performance). They also find that data compression helps, and that the combination of delta encoding and data compression yields the best results.

8.1.5 Workload Characterization

There have been a lot of workload characterization studies for web content, dealing with both static content (focusing primarily on the access patterns) and dynamic content (addressing both access and update patterns).

Bestavros [18] analyzes web logs from Boston University and finds that the more popular a file is, the less frequently the file changes. Gwertzman and Seltzer [73] later collect logs from mainly the school environment as well, and confirm Bestavros's observation [18] that popular files tend to be unchanged.

In contrast, Douglis et al. [52] get the result that more popular resources change more frequently rather than less from traces of two large corporate networks, and suggest that the divergence of these results may come from the environmental difference. Labrinidis and Roussopoulos [106] also found a strong correlation between popularity of web pages and update frequency, using access and updates traces from a stock market website.

Finally, Breslau et al. [22] gather 6 representative web traces across the university environment and the corporation environment to investigate the same correlation. Their result shows that the statistical correlation between a document's access frequency and its update rate is generally quite low and varies from trace to trace. Therefore, it is best to assume that there is no correlation while designing a cache coherency mechanism. This result demonstrates that we do need both push and pull approaches.

8.2 Database Caching and Materialized Views

There have been a large number of publications on answering queries using materialized views in relational databases. The book edited by Gupta and Mumick [71] contains a thorough monograph of the state-of-the-art in this regard. In this section, we present a sample of the related work. In particular, we look into five questions/issues: query answerability, update applicability, efficient updating of materialized views, updates in soft real-time databases, and a few innovative uses of database caching.

8.2.1 Query Answerability

Larson and Yang [109] present theoretical conditions to determine whether and how a query can be answered using a single materialized view defined via Selection-Projection-Join (SPJ) expressions. Rajaraman et al. [137] study answering queries using template-based views that have restricted binding patterns for variables in the templates. Goldstein and Larson [62] study view utilization in a transformation-based query optimizer. The authors propose a fast and scalable algorithm to compute sub-query expressions from materialized views defined via SPJ and group-by operations. The authors also develop a special index on view definitions to reduce the number of candidate views that need to be examined.

Halevy [75] introduces three classes of applications for the problem of answering queries using materialized views, including query optimization and database design, data integration, and semantic caching in client–server systems. The author further describes algorithms and theoretical results proposed for the problem in each class of application.

8.2.2 Update Applicability

Blakeley et al. [20, 21] provide sufficient and necessary conditions for detecting update irrelevance by validating Boolean expressions. They can handle updates for SPJ views. The limitation is that the algorithms to prove the satisfiability of Boolean expressions are quite expensive under normal conditions.

Elkan [56] presents a mechanism to determine whether a query is independent of an update for Datalog. He shows a model-theoretic definition of independence, its basic properties and a proof-theoretic condition for a conjunctive query to be independent of an update. He also introduces a practical induction scheme to deal with recursive queries.

Levy and Sagiv [111] further consider irrelevant updates for Datalog with negated base relations, recursive rules, and arithmetic inequalities. They reduce the independence problem into the equivalence problem for Datalog programs, and then propose schemes to detect the two subclasses of equivalence, query-reachability and uniform equivalence.

8.2.3 Efficient Updating of Materialized Views

Hanson [76] analytically compares three different view materialization strategies, query modification which is a strategic recomputation, immediate view synchronization [21], and deferred view synchronization [76, 144]. The result shows that under different database structures, different view definitions (i.e., selection, projection, join, and aggregates), and different query/update activity patterns, the most efficient strategy is different. Thus, there is no clear winner.

Vista [164] does a thorough monograph in her PhD thesis and supports the same statement that whether to choose the incremental view maintenance or re-evaluation could not be decided a priori, and should be guided by the actual query load. As a result, the decision is better to be made by the database query optimizer at the time of view maintenance.

There is also prior work on efficient updating of materialized views. Blakeley et al. [21] present a maintenance mechanism for materialized views. The mechanism filters database updates that do not affect the views and propagates the remaining updates to the views by recompuation. Abiteboul et al. [2] propose an incremental algorithm to update materialized views over semi-structured data represented in a graph-based data model called OEM. The algorithm works based on the view specification as well as special data structures generated during view population.

Materialized Views for Relational Data: There has been a lot of work exploring the incremental maintenance problem of materialized views for relational databases. Gupta and Mumick [70] present a taxonomy of the view maintenance problem based on three different dimensions in the problem space: amount of information, expressiveness of view definition language, and type of modification. We classify existing algorithms into three categories, counting algorithms, algebraic differencing, and production rules, based on the approach they adopt.

Counting Algorithms: One of the most widely used incremental view maintenance algorithms is the counting algorithm. It was first proposed in [21]. This scheme maintains a multiplicity counter for each view tuple to correctly handle insertions and deletions. Gupta et al. [69, 72] later use a counting algorithm to track the number of alternative derivations of each tuple in the materialized view. Their counting algorithm in [69] is suggested to handle non-recursive views only. Then they extend it in [72] to include recursive views by proposing the *Deletion and Rederivation (DRed)* algorithm, which deletes view tuples that have alternative derivations from the overestimate and then rederives new tuples with alternative derivations.

Algebraic Differencing: Several papers adopt algebraic change propagation schemes [64, 65, 134]. Qian and Wiederhold [134] present an iterative algorithm for the incremental recomputation of relational expressions based on the algebraic differencing approach. Griffin et al. [65] improve Qian and Wiederhold's [134] work with a recursive algorithm, and corrects the minimality condition preservation. Griffin and Libkin [64] extend the algebraic approach to multiset algebra operations (bags), and with aggregations. They propose an approach based on equational reasoning of bag-valued expressions and list the advantages of this approach over the algorithmic approaches. Finally, they prove that their change

propagation algorithm performs much better than recomputation in both time and space efficiency.

Production Rule: Production rules specify data manipulation operations when certain conditions are met or when certain events occur. Ceri and Widom [33] use production rules to maintain views as general SQL queries without duplicates and aggregations. They first perform a syntactic analysis on the view definition to determine if incremental maintenance is possible. If yes, the system automatically derives set-oriented production rules to maintain the materialized views.

All of the three categories can well handle views over SPJ expressions without duplicates, negation, aggregation, or recursion. The only two schemes that can properly handle duplicates are [72] and [64]. They are also the only two that can handle aggregations. The work by Gupta et al. [72] can deal with negation and recursion as well, hence is the least restrictive one among all.

Most of these traditional view matching and maintenance techniques are applicable to either a database server or a mid-tier database cache. In general, they are applicable to any cache that contains materialized views derived from the original tables at the database server. On the other hand, the unique characteristics of a web database make certain modifications of the techniques necessary. For example, the communication cost between the cache and the database server must be considered when selecting materialized views in the cache to answer a query, if the cache is not co-located with the db server. The cache needs to generate an efficient distributed query plan that utilizes both the local materialized views and the remote original tables based on cost estimation. In order to minimize the influence on website performance, database updates must be carefully scheduled before they are propagated to the materialized views in the cache [106]. Adaptive, performance-driven view selection techniques are essential for this [104, 105, 107].

Materialized Views for Semi-Structured Data : Recently, the XML semi-structured data model has started attracting a lot of attention. Several papers [2, 146, 178], which specifically aim at XML caching applications, have focused on incrementally maintaining materialized views over XML documents.

XML views in the cache are mainly the results of the previous queries. All these XML view maintenance algorithms focus on how to efficiently issue queries to the data source. The differences are in data models and view specification languages.

Zhuge and Garcia-Molina's [178] work is one of the earliest papers. It assumes a directed tree-structured data model in what they called *graph structured database* (GSDB). Instead, Abiteboul et al. [2] assume a more general graph-based data model and the query language Lorel. Their experimental results show that their algorithm is always more efficient than recomputation, even when there are thousands of updates.

Although the results from previous work look promising, the view specification languages assumed are still limited. Balmin et al. [14] extend the materialized view specification language into path expressions (which forms the core of XPath[1] and the XQuery[2] language). Then Sawires et al. [146] study incremental view maintenance on path-expression views. They propose to analyze the source updates, and incrementally update the cache based on the relevance of the updates to the cached results. Their experimental results confirmed the performance benefits of their algorithm.

8.2.4 Updates in Soft Real-Time Databases

Adelberg et al. [4] study the problem of update processing in the context of real-time databases, in which each transaction has a deadline and will be aborted if the deadline is missed. In real-time databases, updates should be processed in a timely fashion so that the database is kept up-to-date, and so do transactions due to deadlines.

[1] http://www.w3.org/TR/xpath
[2] http://www.w3.org/TR/xquery

The authors first focus on update processing for base data and the scheduling of updates and transactions. They classify the strategies into four approaches: (i) do updates first (UF), (ii) do transactions first (TF), (iii) split updates (SU), and (iv) on demand (OD). UF gives updates higher priority and will apply an update whenever it arrives. TF is the other way around, and only applies updates when no transactions are waiting. SU will perform updates first for data of high importance and execute transactions first for data of low importance. With the OD strategy, transactions are given a higher priority. However, whenever a transaction handles a stale data item, it will update the stale data first. Their experimental results suggest that OD achieves the best overall performance. In case OD is not applicable, either UF or TF should be chosen based on the relative importance of database freshness and transaction response time.

Adelberg et al. [5] study the recomputation process of derived objects in real-time databases. They explore the combination of schemes from multiple dimensions, including incremental recomputation and full recomputation, how to batch several updates into a single recomputation, and whether to block the transaction on stale data. Their results indicate that recomputations should be delayed slightly so that several related updates can be combined in a single step. This approach, Forced Delay, strikes the balance between recomputing derived objects and executing transactions timely. Also, the incremental recomputation performs better than full recomputation, although there are cases that incremental updates are not feasible. (For a more detailed discussion on Update Scheduling, see Section 5.4.)

Finally, in the work of Kang et al. [92] users impose freshness requirements, and also *deadlines* by which they need to receive the responses to their queries by. The adaptive algorithm proposed in [92] tries to maximize the number of users whose response times are within the specified deadline, while at the same time meeting the freshness requirements.

8.2.5 Database Caching

Elhardt and Bayer [55] propose a DB cache approach that enhances the availability of traditional database systems. Their approach replaces the

buffer space in a DBMS with a *cache* space in main memory and a *safe* space on disk. Transaction failures are all handled by the cache without disk I/O. The safe space enables fast transaction commit as well as recovery of cache pages upon restart after system failure. This way both commit and recovery only involve the small safe space rather than the original database. To handle media failure such as disk damage, an additional archive safe and an archive database are used. Through a prototype implementation and evaluation, the authors show that the DB cache could achieve high throughput for small to moderate-sized transactions. The DB cache can also handle long, update-intensive transactions well.

Additionally, Hellerstein and Naughton [78] study caching results of expensive user-defined functions in object–relational databases. Memoization is a traditional approach to such function caching, which builds a hash table in the main memory for function results that correspond to different parameter values. In this work, the authors propose a variant of unary hybrid hashing called *Hybrid Cache* that combines memoization and sorting. They experimentally demonstrate that Hybrid Cache outperforms memoization in general.

8.3 Caching in Client–Server Databases

There has been much previous work on client-side query caching for distributed databases with client–server architectures. Keller and Basu [94] propose a predicate-based caching scheme that loads query results into client caches when they are returned by the server. A new query at a client can be evaluated locally if its result is completely contained in the results of previous cached queries. The containment checking is based on the cache descriptions of query predicates at both the client and the server. The authors also investigate techniques to ensure data consistency between the clients and the server when a database update occurs at either side.

Dar et al. [42] develop a client-side semantic caching model. Results of previous queries are organized into semantic regions. A semantic region has a finer granularity than a query and is described by a

predicate. A new query whose result overlaps data in a client cache is decomposed into two sub-queries: a *probe query* evaluated using the local cache and a *remainder query* forwarded to the server.

Kossmann et al. [95] consider the relationship between client-side caching and distributed query optimization. The authors develop a novel approach called *Cache Investment* that deliberately generates a sub-optimal plan for the current client query, with the potential benefit of good data placement in the client cache that enables better plans for future queries.

Caching in client–server databases brings forth the important issue that the client caches should not cause violation of transaction semantics in the whole system. Wilkinson and Neimat [167] study the problem of cache consistency guarantees in client–server databases. In their environment, the database at the server is shared by multiple clients and each client keeps a portion of the database in its own cache. To solve the problem, the authors extend traditional two-phase locking with two new kinds of locks, cache locks and notify locks, and propose corresponding algorithms. Wang and Rowe [166] experimentally compare five cache consistency algorithms between transactions in client–server databases: (i) two-phase locking, (ii) certification, (iii) callback locking, (iv) no-wait locking, and (v) no-wait locking with notification. The performance metrics evaluated in their experiments include throughput and response time. The results show that it is desirable to implement multiple cache consistency algorithms and adaptively switch among them according to current application characteristics. The two-phase locking algorithm provides the general best performance over all algorithms under various workloads and system configurations. Franklin et al. [58] present a taxonomy of algorithms for transactional cache consistency maintenance in client–server databases. The authors further conduct a performance evaluation of six specific algorithms in the taxonomy to study the design trade-offs of the algorithms.

As a dynamic form of replication, client-side caching has been commonly used in distributed client–server databases to improve data availability [42, 58, 94, 166]. A client caching mechanism in such environments should ensure that the failure of a client will not affect the availability of data for applications running at other clients [58].

Hu et al. [83] present an adaptive caching model to support spatial query processing at mobile clients. In this model, the result objects of a previous query are cached reactively and the R-tree index of these objects are cached proactively. The cached index enables the effective reuse of the cached objects to answer subsequent queries.

8.4 Caching in Distributed Databases

Query-level caching has been widely adopted in distributed databases [42, 94, 95], data warehouses [47, 91, 117], mediators [3, 37, 110], and P2P systems [90, 132]. All these approaches cache results of previous queries to answer subsequent queries, in a manner similar to that of proxy caching.

Mediators [3, 37, 110] are similar to reverse proxies because they integrate data from multiple web sources and provide a uniform query interface to the users. A major difference is that mediators often serve a specific application, e.g., a search engine, but reverse proxies serve a general class of web applications. Adali et al. [3] study intelligent maintenance of mediator caches. In their work, a mediator cache uses recorded access statistics of sources to optimize the processing of a query in a distributed, cost-based manner. Chidlovskii et al. [37] and Lee and Chu [110] study semantic query caching in mediator caches. The mediator stores the results of previous queries as semantic regions [42] in the cache and reuse these results to reduce the response time and network traffic of a new query.

The multi-cache category of reverse proxy caching (page 181) bears a similarity to caching in peer-to-peer (P2P) systems [90, 172]. The main difference is that in P2P systems individual peers make their local caching decisions, while in the multi-cache category a global caching decision is made for all caches in a group.

Kalnis and Papadias [91] propose a proxy-server architecture for data warehouses. A proxy cache in this work is called an *OLAP Cache Server* (OCS) and stores dynamic query results of *On-Line Analytical Processing* (OLAP) queries rather than static web pages. The architecture involves multiple networked OCSs and the data warehouse.

An OCS answers an OLAP query locally, or redirects the query to its neighboring OCSs or the data warehouse. Three different policies are designed to control the caching decisions as well as the query plan construction at the OCSs: a centralized, a semi-centralized, and an autonomous policy. The centralized policy uses a central site that has full knowledge of all OCSs to construct the evaluation plan of every OLAP query and to decide what data to cache on each OCS. In the semicentralized policy, the query plans are constructed by the central site but each OCS by itself decides what to cache. There is no central site in the autonomous policy and all decisions are made locally at individual OCSs.

Loukopoulos et al. [117] investigate active caching of OLAP views together with static pages at local area network (LAN) proxies. The targeted scenario is that geographically distributed clients issue ad hoc OLAP queries to a central data warehouse (DW) on the Web. Since the user-perceived performance of answering such queries depends on both the network latency and the server computation, the authors propose to cache part of the DW data at a proxy and to construct the results for subsequent queries locally based on the cached data. Furthermore, they design a cost model to estimate the benefit of caching each OLAP view at the proxy. Based on the cost model, the authors develop a cache replacement algorithm that takes into consideration the OLAP query processing cost. Consequently, the proposed framework can answer OLAP queries more efficiently than an ordinary proxy cache.

Kalnis et al. [90] propose a P2P caching architecture for OLAP queries called PeerOLAP. The peers in PeerOLAP are similar to the proxies in a web database scenario and the data warehouses similar to servers. The authors propose eager versus lazy query processing as well as isolated versus hit-aware caching policies in the architecture. These techniques utilize cooperation between peers to reduce the duplicates among caches and to improve performance.

Patro and Hu [132] propose a scheme to cache query hits at the gateway of a P2P network. The gateway of peers in this work can be viewed as the proxy of a number of clients in web caching. The scheme exploits the locality of the queries going through the gateway. The

caching is transparent to the peers so that no modification is required on them. The authors develop an algorithm to answer queries and to manage the cache at the gateway.

Yu and Vahdat [176] study the problem of numerical error bounding of data items in replicated databases. In such a database, a data item may have multiple replicas and each replica has a single numerical value. To ensure data consistency over the entire replication framework, the difference between a replica value and the real value of an item must be bounded within a pre-defined range. The authors present two algorithms, *Split-Weight AE* and *Compound-Weight AE*, to effectively and efficiently bound the absolute error of a data item. Split-Weight bounds value increase and decrease separately, whereas Compound-Weight bounds both directions of value change holisticly. The authors further propose a third *Inductive RE* algorithm that transforms relative error to absolute error and then applies the previous two algorithms. Additionally, the authors discuss optimization techniques that reduce the space and computation overhead of the algorithms.

Extending the work by Yu and Vahdat [176], Cetintemel and Keleher [34] further proposed two server-side precision-bound maintenance algorithms: Share-Bound and Partition-Bound. Servers in Share-Bound collaboratively maintain the precision bounds, whereas those in Partition-Bound each maintain the precision bounds for a disjoint subset of data items. The authors show that Partition-Bound is a generalization of Yu and Vahdat's algorithms.

8.5 Caching in Distributed Systems

There has been a lot of work dealing with caching and replication/materialization in distributed systems.

Triantafillou and Neilson [162] propose a consistency protocol for distributed file systems. The protocol adopts a strong consistency semantics that every previous update of a file will be seen by a later access of the file. Failures of servers and clients are both efficiently handled in the protocol to ensure file availability. The servers collaborate with one another so that file updates unseen at a server are obtained from other nearby servers as needed. Moreover, a server can continue

its normal operation when it is doing failure handling. A number of availability enhancement methods are further employed, such as allowing unrestricted access of a file copy at a client cache until a server callback. As a result, the protocol provides the user a transparent, centralized view of a replicated file system with a good performance.

Chubby [25] is a distributed lock service used in the Google file system. Availability and reliability are the main design goals of Chubby. In Chubby, fault tolerance is based on distributed consensus of multiple replicas. To reduce the server workload, client-side caches are utilized and are kept consistent by update notification.

Coda [145] is a system that supports both stationary and mobile file access over wired or wireless networks. In order to achieve high data availability, Coda employs *disconnected operations* to complement server replication. In a disconnected operation, the client keeps on running even when it is not connected to any server. The cache at the client stores files needed for remote processing as well as those recently accessed by the client. Before it disconnects from the servers, the client makes sure that all files in its cache are up-to-date. After it is disconnected, file modifications are performed on cached copies with an optimistic replication policy. Finally, when the client recovers its connection, the updated files are sent to corresponding servers.

Gao et al. [59] use a distributed object approach to build an application-specific data replication system at edge servers. The system is targeted for e-commerce applications, and the authors use the TPC-W benchmark in their implementation and evaluation. Based on the specific application semantics, the authors present a design category of the distributed objects replicated in their TPC-W system. Example objects are catalogs, orders, profiles, inventories, and best-seller-lists. Each object manages a specific subset of shared information using simple and effective consistency models. The authors experimentally demonstrate that the system achieves both high availability and good performance by slightly relaxing consistency within individual distributed objects. The throughput and response time of the system are indifferent to network partitioning and the response time is close to that of an ideal system with high speed and reliable connection links.

8.5.1 Clusters

Krishnamurthy and Wang [98] present a method to identify clusters of geographically adjacent clients that contribute a significant number of requests to a website. The clients are clustered in a network-aware, automatic way based on BGP routing information. The authors employ a self-correction and adaptation mechanism to improve the applicability and accuracy of the initial results of clustering. After the clusters are formed, website contents can be cached or redistributed close to the clusters to share the server workload and improve response time.

ALBUM [87] is an affinity-based management system for a cluster of main memory databases serving web applications. The authors observe that, in a web application, such as e-commerce and digital library, queries can be divided into multiple groups. Queries in one group access the same set or overlapping sets of data, whereas the data accessed by queries in different groups are disjoint. The authors utilize such *query affinity* to distribute data among the databases in a cluster so that a query can be executed in a single database to save data transmission and synchronization. More specifically, ALBUM executes each query from a cache server in two stages, a local translation stage and a remote explicit stage. Each stage of execution involves a single cache in the cluster. The first stage decides the set of data that the query will likely access. This decision is then used to determine which cache server will execute the query in the second stage. If the selected cache server does not contain all data for the explicit stage of a query execution, missing data will be collected from a master database.

Tang et al. [158] investigate the problem of assigning requests in a single session to a cluster of web servers. A session in the paper refers to all requests sent over a single TCP connection. The authors prove that the problem is NP-complete, and propose a greedy heuristic algorithm for it. The heuristics are based on the access probabilities of different objects in a session.

H-SWEB [11] is a system that implements a dynamic scheduling mechanism for HTTP requests to a cluster of web servers residing in a local network. The scheduling model in H-SWEB is hierarchical where a server can provide service to either a cluster or a super cluster. A super

cluster can be a cluster of clusters or super clusters recursively. The system is adaptive to the workload change of the clusters.

Zhang et al. [177] propose a load balancing policy called ADAPTLOAD for a homogeneous web server cluster. The policy is workload-aware and self-tuning: the parameters of a server are dynamically adjusted based on characteristics of incoming requests as well as variation of the operation environment. The behavior of ADAPTLOAD is similar to a locality-aware allocation policy, but it requires no locality information. Both static and dynamic web page cachings are supported by the policy. The authors evaluate ADAPTLOAD using a real-world workload from 1998 World Cup and show that it achieves a high cache utilization and low slowdowns.

Cohen and Kaplan [39] study the impact of object aging in *cascaded caches* in web content distribution. The authors propose a weak consistency policy for cached objects based on TTL values. An object copy that a cache fetches from another cache usually has a smaller TTL than that fetched from the original server, i.e., the object ages as it travels among caches. The authors present a model of object distribution, and use different inter-request time distributions (Poisson, Pareto, and fixed-frequency arrivals) and trace-based simulation under various cache settings to evaluate the performance effect of object ages on cache miss rates.

Rodriguez et al. [143] develop analytical models to compare hierarchical and distributed web caching architectures on several performance factors, including client response time, network bandwidth usage, cache workload, and utilization. They show that hierarchical caching with intermediate caches reduces bandwidth consumption. In comparison, distributed caching achieves a good performance in well-interconnected areas without requiring any intermediate cache levels. The authors further investigate a hybrid combination of the two caching architectures and evaluate its performance.

Wu and Liao [171] proposes a *Virtual Proxy*, which not only performs data caching, but also provides various common services, such as search engines, information filtering, and intelligent agents. The authors implement two basic services to demonstrate the effectiveness of a virtual proxy: an object lookup scheme and a searchable proxy.

The former searches objects by maintaining global resource index tables that record object distribution information in a virtual proxy tree, the latter searches the cached objects at a virtual proxy.

8.5.2 Caching in Mobile Environments

Barbara and Imielinski [16] bring the idea of server invalidation into the mobile wireless environment to keep the cache coherent. They propose that a server periodically broadcasts an *invalidation report*, in which the changed data items are indicated, while clients listen to the invalidation report over wireless channels. Then the users that are often disconnected (sleepers) are differentiated from the users which are connected most of the time (workaholics), and different invalidation strategies are applied to different users, to achieve the best overall performance.

Jing et al. propose the Bit-Sequences algorithm [89], which adaptively adjusts the size of the *invalidation report*, to optimize the use of a limited communication bandwidth while retaining the effectiveness of cache invalidation.

8.5.3 Asynchronous Validation

Rabinovich and Spatschek [135] categorize two approaches to implement asynchronous validation, (1) with separate threads on clients and (2) with time triggers.

The most famous asynchronous validation scheme is the **Alex protocol** [32, 73], which originated from the Alex FTP cache [32]. Based on the observation that file lifetime distribution tends to be bimodal (i.e., either a file remains unmodified for a long time or it will be modified frequently), Gwertzman and Seltzer [73] proposed a variant of the original protocol of the Alex FTP cache. This improved scheme is widely used in many systems. In the improved scheme, the update threshold θ_1, which determines how frequently to poll the server, is set as a percentage (*perc*) of the object's age (since last modified). As long as the time since last validation does not exceed the threshold, the object is considered valid. Furthermore, another threshold θ_2 is added as a sanity check to make sure that the old objects will also be validated

sometime, as shown in Equation (8.1).

$$\theta_1 = \min\{(perc \times (send\_time - last\_modified)), \theta_2\}. \tag{8.1}$$

8.5.4 Leases

Leases were first proposed by Gray and Cheriton [63] to maintain file consistency in distributed file systems. To cache an object, the client first acquires a lease from the data source, which specifies a time interval t. Within the time interval, the data source will notify the client for any invalidation or modification of the cached object.

After the lease expires, the server has no obligation to notify the client, it would be up to the client to make sure any cached data are up-to-date. This could happen *proactively* with the client either renewing the lease before it expires or starting to pull modifications from the server, or *reactively*, once the client has a request for a data item whose lease has expired.

Adaptive Leases: Duvvuri et al. investigate the lease duration problem in [53]. They provide experimental results to show that short leases have a larger control message overhead, and long leases have a larger state space overhead on the servers. To strike a balance, they propose adaptive policies to calculate the optimal lease duration. These heuristics are motivated by observations from object lifetimes, renewal frequencies and state space overhead, respectively. Duvvuri et al. implemented their adaptive lease technique and demonstrated its effectiveness and efficacy.

Volume Leases: The concept of volume validation has been introduced in the Andrew File System [81], NFS [125], and [40, 112]. In order to further improve performance, Yin et al. [174, 175] propose the concept of volume leases. By grouping objects into volumes and maintaining consistency at a coarser granularity, the control message overhead (devoted to lease renewals) is amortized over a large number of objects. Moreover, the space overhead on servers, which is used to maintain client state information, is also reduced.

Under the volume leases protocol, a client can read a cached object only if it holds valid leases on both the object and its volume. A server can modify an object if all clients acknowledged its invalidation

messages or either lease (i.e., the object's lease or volume's lease) expires. In this scheme, long object leases and short volume leases can be well combined and achieve the best overall performance, since long object leases avoid high maintenance cost and short volume leases avoid long time delay for data source modifications.

8.5.5 Piggybacking

Krishnamurthy and Wills established a series of mechanisms in this direction, including Piggyback Cache Validation (PCV) [99], Piggyback Server Invalidation (PSI) [100], and their combination [40, 101].

Piggyback Cache Validation (PCV) [99] focuses on piggybacking batch validation requests from proxy caches to the servers. In PCV, when a proxy cache has a chance to communicate with a server, it checks if it cached any objects which have been expired or about to expire from this server. The proxy then piggybacks If-Modified-Since requests for the batch of these potentially stale objects for validation. With enough traffic to support piggybacking, PCV achieves strong cache coherency as well as saves a lot of messages. In order to maintain the traffic so that piggybacking is viable, the client has to be the proxy cache instead of the individual one.

Piggyback Server Invalidation (PSI) [100] works for servers to piggyback objects. In PSI, servers partition resources into volumes, and maintain version information for each volume. When a server receives an If-Modified-Since request from a proxy cache, including an object and the version number for the object's volume, the server piggybacks the list of all objects in the same volume which have been modified since the client provided version. The proxy cache then invalidates objects which are both in its cache and in the list, extending the lifetime of all other objects which are not in the list.

Both the PCV and the PSI mechanisms yield stronger cache coherence and less network cost, by using the piggybacked batch information. Krishnamurthy and Wills [40, 101] further propose to combine the PSI and PCV techniques and create a hybrid approach, where the best overall performance can be achieved. Under the hybrid scheme, the choice of the mechanism depends on the time elapsed since the last

time the proxy requested invalidation for the volume. If the time interval is small, PSI is used, otherwise, the PCV mechanism is used. The reason behind this scheme is for a short period, the number of modified objects tends to be small, and thus sending invalidation is more efficient than sending validation messages. In contrast, for a long time period, the number of modified objects tend to be much larger, so that sending validation requests will potentially reduce the communication cost.

8.6 Industrial Products/Standards

Edge Side Includes [54] is a simple markup language used to specify the generation of dynamic contents in a web page. As such, web caches with an ESI processing capability can cache the ESI directives in a page and generate dynamic content and assemble the page when the page is requested. In the client-side ESI-compliant approach proposed by Rabinovich et al. [136], ESI segments are stored in the browser cache to dynamically construct a web page on demand. One disadvantage of ESI is that because users need to specify how a page fragment is generated, the caching decision is not transparent to the users.

AJAX (*Asynchronous JavaScript and XML*) [6, 152] is a group of advanced technologies used to develop user-interactive Web applications. Example technologies in Ajax include XMLHttpRequest, XHTML, CSS, and DOM. Ajax has been implemented in various languages and libraries, such as ActiveX, Flash, and Java applet. The main feature of Ajax is its asynchronous nature: By bringing code to a web browser and generating HTML there, the browser can send data to and receive data from the server without interfering with the display of the current page. There are several known problems of Ajax today, e.g., browser compatibility issues due to the use of JavaScript and DOM, possible response time delay caused by preloading and handling of the XMLHttpRequest object, and vulnerability to advanced attacks that subvert client–server communication.

The *Akamai* [7, 49] network consists of more than 15,000 servers deployed in thousands of ISP networks all over the world. It serves Akamai's customers by hosting their web contents and applications with performance and reliability guarantees. It provides a number

of configuration options for caching, e.g., HTML cache timeouts and whether to allow session data to be cached. ACMS [150] is a fully functional configuration management system for the Akamai network. The system accepts distributed submissions of configuration information from customers and disseminates this information to the Akamai CDN. ACMS is highly available, distributed, and fault-tolerant. It manages the configuration updates by adapting existing quorum-based algorithms such as *Vector Exchange* and *Index Merging* to support consistency and easy storage recovery.

TimesTen [160, 161] is an in-memory RDBMS that can be deployed at an application server and serves as a mid-tier cache. Disk data stored at the back-end database server is cached and processed in mid-tier main memory. As a result, TimesTen provides low response time and high throughput to data-intensive Internet applications. Cached data in TimesTen can be either subsets of frequently used relational tables existing in the back-end database, or exclusive tables created by the applications. TimesTen processes SQL queries and updates from applications over the cache data. Multiple mid-tier caches can co-exist and store disjoint or overlapping subsets of the same back-end database. The caches are kept synchronized with each other as well as with the back-end database.

Tangosol [159] is a reliable in-memory data grid technology that has been acquired by and integrated with Oracle. It is designed to meet the new requirements of real-time data analysis, computation-intensive middleware and high-performance transactions. The combination of Tangosol with Oracle Fusion Middleware, Oracle TimesTen, and Oracle Database creates an integrated platform that enables extreme transaction processing.

Grundy et al. [66] investigate the use of an object-oriented persistency framework to improve the performance of enterprise application servers. The authors transparently employ an in-memory database at the application server. This database stores all objects and indices and writes transaction logs. Liu [116] empirically studies the performance and scalability of EJB application servers. The author introduces the architecture of EJB clustering and proposes three scaling approaches

based on the EJB clustering scenarios. The author also develops and tests a benchmark application for performance measurement.

Kounev et al. [96] describe their experience in deploying the industry-standard *SPECjAppServer2004* benchmark on the JBoss platform. The authors examine a number of deployment alternatives, e.g., three web containers (Tomcat 4.1, Tomcat 5, and Jetty) and two JVMs. The authors measure and analyze the effect of these alternatives on the overall system performance in both single-node and clustered environments.

Bakalova et al. [13] introduces caching techniques used in the *IBM WebSphere Dynamic Cache* for a variety of objects including Java Servlets, JavaServer Pages (JSPs), WebSphere command objects, Web services objects, and Java objects. The authors provide examples for these dynamic caching techniques, discuss their unique technical requirements and specific implementation methods, and measure their performance using the Trade3 IBM J2EE benchmark application.

WebExpress [80] aims at a more bandwidth-constrained environment, such as a wireless environment. They focus on small changes to the dynamic data (CGI output). They store the shared common base objects, and only transfer the deltas, i.e., the differences between the base objects and the responses.

9

Open Research Problems

Although caching and materialization are well-known techniques that have been used to improve system performance for decades, their full potential to improve performance/scalability of web databases without sacrificing the quality of the data being served has only materialized recently. To this day, the problem is not fully "solved"; in fact, the ever-changing technological landscape is creating additional problems/opportunities for research. In the following paragraphs, we list some of the challenges that we expect to drive research on caching and materialization for web databases in the future.

9.1 Cloud Computing

The typical architecture for web-database servers is quickly evolving from a configuration with a dedicated cluster of machines (and a load balancer in front) to a *cloud computing* infrastructure, potentially used to host multiple web-database server installations at the same time, through virtualization.

Under the cloud computing paradigm,[1] a user's queries and computations are handled by an "amorphous" collection of machines,

[1] http://en.wikipedia.org/wiki/Cloud_computing.

the cloud, for which many of the low-level details have been made transparent to the user. Two of the primary tenets of cloud computing are the following:

- *massive parallelism*, which is easily exploitable when dealing with workloads that have large numbers of small tasks, as is typical for many web-database servers; and
- *tolerance to failures*, which occur frequently due to the high number of machines typically participating in the cloud.

Such an environment brings multiple challenges, but also many opportunities for caching and materialization. However, most of the existing techniques are bound not to scale to typical cloud sizes and would need to be rethought.

Given the unique characteristics of cloud computing, it is no surprise that the debate is still ongoing on what is the proper programming paradigm for it, especially with regards to data-intensive tasks. In particular, the question remains on whether parallel database management systems (and SQL) can handle the scale required or new programming frameworks, such as MapReduce, are most suited for cloud computing over large datasets [45, 46, 133, 153].

9.2 User-Centric Computing

Although cloud computing is expected to make web-database servers more "generic", since their details will be abstracted out, we envision an increase in the *role of the client* in caching/materialization solutions. In the past, caching was mostly beneficial if the same content was shared among multiple users (i.e., favoring proxy caches at ISPs), however, nowadays web pages often have full-blown applications (e.g., using AJAX) that typically require a lot of data, possibly from different sources (through a mashup application). In many cases, local caches can store application state and additional data, providing for a more interactive user experience. This is being reinforced further by the new HTML 5 standard[2] which will support local storage at the client.

[2] http://en.wikipedia.org/wiki/HTML_5.

Given the myriad of options and alternatives, it would be difficult to identify solutions that work well for every user. This is further exacerbated by the move to cloud computing, which aims at consolidating servers to achieve better economy of scale. As such, it would be completely impossible for simple performance and other global system metrics to be able to keep all users happy. Therefore, we believe that considering user preferences and establishing Service Level Agreements (as we saw in Section 6) will become more commonplace in the near future. Given the early stage of work on Service Level Agreements for web-database systems, a lot of open problems still exist in the area.

Going beyond user preferences and Service Level Agreements, *personalization* is expected to have a major impact in web-database services of the future. For example, user queries can be personalized to include either past history from the same user, current context of the user (e.g., geo-location), or past history/feedback from "similar" users. In such an environment, caching/materialization can still be beneficial, but will pose unique challenges.

9.3 Mobile Computing

If cloud computing can be considered as a major disrupting technology on the server side, then mobile computing is undoubtedly a major disrupting technology on the client/user side. In 2008, 26 years after the cell phone was first introduced, we reached a watershed moment, with about one active cell phone for every two humans on earth [60]. Going from zero to adoption by half the planet in 26 years corresponds to the fastest global diffusion of any technology in human history, even including vaccines.

The proliferation of cell phones in general and smartphones in particular is making mobile platforms the defacto entry point for accessing webdatabases and many other applications. This brings many challenges and opportunities for caching and materialization. First of all, the obvious connectivity issue makes data caching/materialization an important technology to consider, as has happened already in the mobile data management research literature of the past decade. Nearly always-on connectivity is more commonplace nowadays (especially in

connection with WiFi networks), but there is always the issue of differentiated quality of service and also the issue of monetary cost, with roaming and other charges.

Beyond the traditional connectivity issues, the pervasiveness of smart-phones is bringing a flood of new, innovative, mobile-only networked applications that are enhanced by the additional capabilities of smart-phones (e.g., GPS, compass, voice recognition, etc.). Such applications could potentially benefit from caching and materialization, especially given the resource constraints inherent in even the latest generation smart-phones (e.g., network bandwidth and battery).

9.4 Green Computing

Although battery is an important resource constraint primarily for mobile computing devices, *energy consumption* in general is an important consideration for server configurations; the bigger the installation the bigger the energy concern and the potential savings from energy-saving solutions. Currently, there are many technologies that can offer specific trade-offs in terms of energy consumption, performance, and cost. For example, some modern processors offer Dynamic Voltage Scaling[3] as a way to reduce energy usage (and performance) when circumstances permit. Another energy-efficient technology of particular interest is that of Solid State Drives (SSDs),[4] that offer faster access speeds than traditional hard disk drives (HDDs), but their price is significantly higher than HDDs.

Utilizing energy-efficient technologies is part of a greater initiative toward *green computing*, where the impact of information technology on the environment is considered and attempts are made to minimize it. Caching and materialization can play an important role in this effort, with all the challenges that this entails. For example, Content Distribution Networks are already being used to cache/materialize (primarily static) content closer to the end-users, which minimizes response times, but also minimizes unnecessary data transport and the need for the origin servers to be always online, thus reducing energy. Caching and

[3] http://en.wikipedia.org/wiki/Dynamic_Voltage_Scaling.
[4] http://en.wikipedia.org/wiki/Solid-state_drive.

materialization can also be employed to manage the trade-off between the different capabilities, cost, and energy consumption profiles of the different machines and storage devices in a server infrastructure.

9.5 Non-technical Challenges

Finally, all the above are primarily technical questions. However, caching can often bring forth non-technical issues, primarily stemming from legal requirements and copyright ownership. For example, which content can be safely stored at the proxy cache of an ISP (and shared among all users)[5] and for how long can it stay there? Also, in cloud computing environments, where multiple web databases are co-located, how can one make sure that the proper safe-guards are in place to limit any possible threats to data privacy and security?

[5] http://www.interesting-people.org/archives/interesting-people/200906/msg00063.html.

10

Summary

In this monograph, we view caching and materialization in web databases as a series of actions and introduce the current approaches on these actions. Specifically, we examine where and at what granularity data are cached/materialized, how we select the data items to cache/materialize, how we use the cached data to answer new requests, and how we maintain the stored data so that they provide both good quality of data and high performance. Additionally, we present evaluation metrics and categorize them. Finally, Figure 10.1 summarizes the options/alternatives for each caching/materialization action presented in this monograph.

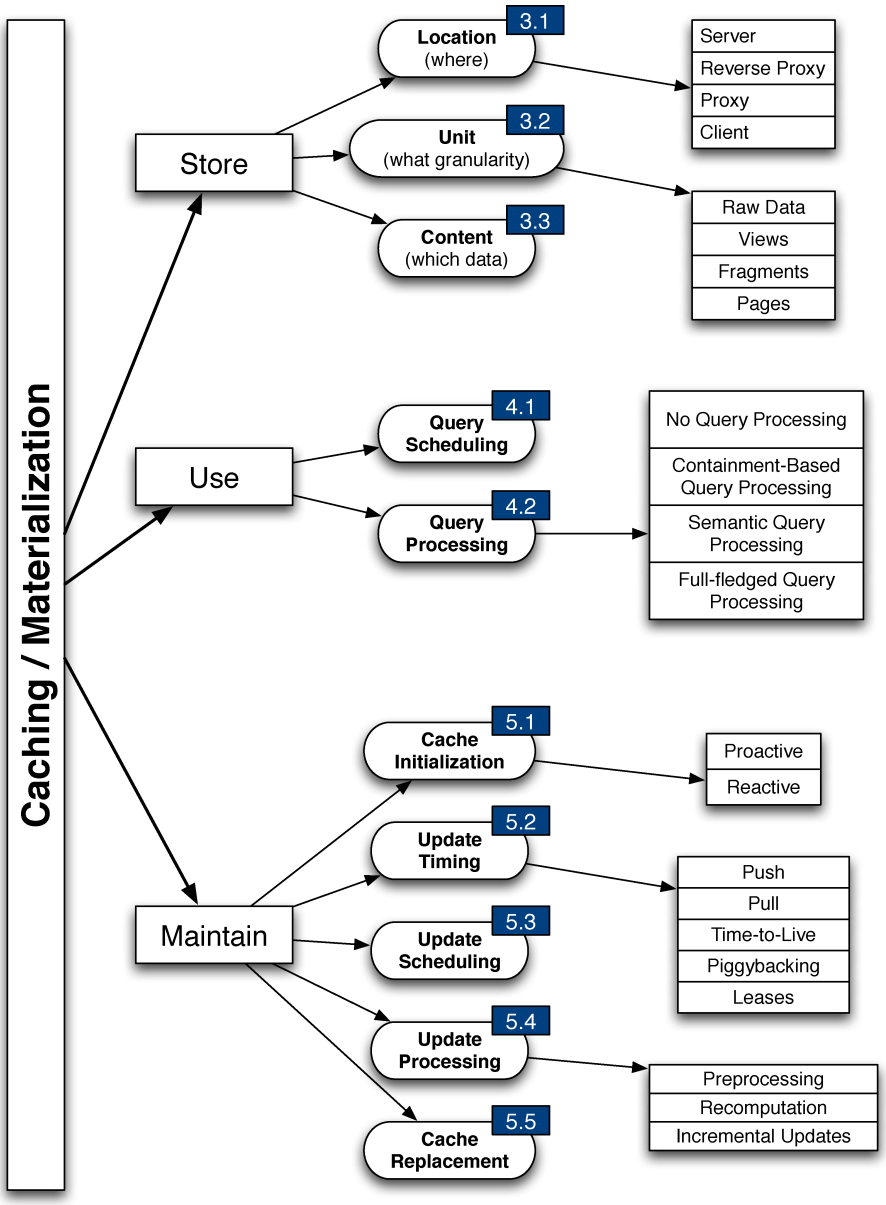


Fig. 10.1 Overview of caching/materialization options.

Acknowledgments

The authors would like to thank Joe Hellerstein and Alon Halevy for their guidance and advice throughout the review process, the anonymous reviewers for their very detailed and helpful comments, and the publisher, James Finlay, for his never-ending patience and perseverance throughout the entire process. This monograph would not have been possible without their combined efforts.

This material is based on work supported in part by the National Science Foundation under grants CAREER IIS-0746696, IIS-0534531, and ITR ANI-0325353, and by the Hong Kong Research Grants Council under grant 617307.

References

[1] R. K. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," *ACM TODS*, vol. 17, no. 3, pp. 513–560, 1992.

[2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener, "Incremental maintenance for materialized views over semistructured data," in *VLDB*, pp. 38–49, 1998.

[3] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian, "Query caching and optimization in distributed mediator systems," in *SIGMOD Conference*, pp. 137–148, 1996.

[4] B. Adelberg, H. Garcia-Molina, and B. Kao, "Applying update streams in a soft real-time database system," in *SIGMOD Conference*, pp. 245–256, 1995.

[5] B. Adelberg, B. Kao, and H. Garcia-Molina, "Database support for efficiently maintaining derived data," in *EDBT*, pp. 223–240, 1996.

[6] Ajax. http://en.wikipedia.org/wiki/AJAX.

[7] Akamai. http://www.akamai.com.

[8] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald, "Cache tables: Paving the way for an adaptive database cache," in *VLDB*, pp. 718–729, 2003.

[9] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan, "DBProxy: A dynamic data cache for web applications," in *ICDE*, pp. 821–831, 2003.

[10] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan, "Scalable template-based query containment checking for web semantic caches," in *ICDE*, pp. 493–504, 2003.

[11] D. Andresen and T. McCune, "Towards a hierarchical scheduling system for distributed WWW server clusters," in *HPDC*, p. 301, 1998.

[12] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, and T. Zhong, "Web caching for database applications with oracle web cache," in *SIGMOD Conference*, pp. 594–599, 2002.

[13] R. Bakalova, A. Chow, C. Fricano, P. Jain, N. Kodali, D. Poirier, S. Sankaran, and D. Shupp, "Websphere dynamic cache: Improving J2EE application performance," *IBM Systems Journal*, vol. 43, no. 2, pp. 351–370, 2004.

[14] A. Balmin, F. Ozcan, K. S. Beyer, R. Cochrane, and H. Pirahesh, "A framework for using materialized xpath views in XML query processing," in *VLDB*, pp. 60–71, 2004.

[15] G. Banga, F. Douglis, and M. Rabinovich, "Optimistic deltas for WWW latency reduction," in *USENIX Annual Technical Conference*, 1997.

[16] D. Barbara and T. Imieliski, "Sleepers and workaholics: Caching strategies in mobile environments," *ACM SIGMOD Record*, vol. 23, no. 2, pp. 1–12, 1994.

[17] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma, "Relaxed-currency serializability for middle-tier caching and replication," in *SIGMOD Conference*, pp. 599–610, 2006.

[18] A. Bestavros, "Demand-based document dissemination to reduce traffic and balance load in distributed information systems," in *SPDP*, p. 338, 1995.

[19] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, "Adaptive push–pull: Disseminating dynamic web data," *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 652–668, 2002.

[20] J. A. Blakeley, N. Coburn, and P.-A. Larson, "Updating derived relations: Detecting irrelevant and autonomously computable updates," *ACM Transactions on Database Systems*, vol. 14, no. 3, pp. 369–400, 1989.

[21] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa, "Efficiently updating materialized views," in *SIGMOD Conference*, pp. 61–71, 1986.

[22] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *INFOCOM*, pp. 126–134, 1999.

[23] E. A. Brewer, "Towards robust distributed systems (abstract)," in *PODC '00: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, p. 7, New York, NY, USA: ACM, 2000.

[24] L. Bright and L. Raschid, "Using latency–recency profiles for data delivery on the web," in *VLDB*, pp. 550–561, 2002.

[25] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *OSDI*, pp. 335–350, 2006.

[26] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal, "Enabling dynamic content caching for database-driven web sites," in *SIGMOD Conference*, pp. 532–543, 2001.

[27] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *USENIX Symposium on Internet Technologies and Systems*, 1997.

[28] P. Cao and C. Liu, "Maintaining strong cache consistency in the world wide web," *IEEE Transactions on Computers*, vol. 47, no. 4, pp. 445–457, 1998.

[29] P. Cao, J. Zhang, and K. Beach, "Active cache: Caching dynamic contents on the web," *Distributed Systems Engineering*, vol. 6, no. 1, pp. 43–50, 1999.

[30] C. Cappiello, M. Comuzzi, and P. Plebani, "On automated generation of web service level agreements," in *Proceedings of IEEE International conference on Advanced Information Systems Engineering (CAiSE)*, pp. 264–278, 2007.
[31] D. Cappucio, B. Keyworth, and W. Kirwin, Total cost of ownership: The impact of system management tools, 1996.
[32] V. Cate, "Alex — A global filesystem," in *USENIX File System Workshop*, pp. 1–12, 1992.
[33] S. Ceri and J. Widom, "Deriving production rules for incremental view maintenance," in *VLDB*, pp. 577–589, 1991.
[34] U. Cetintemel and P. Keleher, "Efficient distributed precision control in symmetric replication environments," in *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, 2002.
[35] J. Challenger, A. Iyengar, and P. Dantzig, "A scalable system for consistently caching dynamic web data," in *INFOCOM*, pp. 294–303, 1999.
[36] J.-H. Chen, R. Anane, K.-M. Chao, and N. Godwin, "Architecture of an agent-based negotiation mechanism," in *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pp. 379–384, Washington, DC, USA: IEEE Computer Society, 2002.
[37] B. Chidlovskii, C. Roncancio, and M.-L. Schneider, "Semantic cache mechanism for heterogeneous web querying," *Computer Networks*, vol. 31, no. 11–16, pp. 1347–1360, 1999.
[38] J. Cho and H. Garcia-Molina, "Synchronizing a database to improve freshness," in *SIGMOD Conference*, pp. 117–128, 2000.
[39] E. Cohen and H. Kaplan, "Aging through cascaded caches: Performance issues in the distribution of web content," in *SIGCOMM*, pp. 41–53, 2001.
[40] E. Cohen, B. Krishnamurthy, and J. Rexford, "Improving end-to-end performance of the web using server volumes and proxy filters," in *SIGCOMM*, pp. 241–253, 1998.
[41] M. Conti, M. Kumar, S. K. Das, and B. A. Shirazi, "Quality of service issues in internet web services," *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 593–594, 2002.
[42] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *VLDB*, pp. 330–341, 1996.
[43] A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, and K. Ramamritham, "Proxy-based acceleration of dynamically generated content on the world wide web: An approach and implementation," *ACM Transactions on Database Systems*, vol. 29, no. 2, pp. 403–443, 2004.
[44] J. S. David, D. Schuff, and R. St. Louis, "Managing your total it cost of ownership," *Communications of the ACM*, vol. 45, no. 1, pp. 101–106, 2002.
[45] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[46] J. Dean and S. Ghemawat, "Mapreduce: A flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, 2010.
[47] P. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton, "Caching multidimensional queries using chunks," in *SIGMOD Conference*, pp. 259–270, 1998.

[48] J. Dilley, M. Arlitt, S. Perret, and T. Jin, "The distributed object consistency protocol," in *HP Labs Technical Report*, 1999.
[49] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally distributed content delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, 2002.
[50] A. Dingle and T. Partl, "Web cache coherence," in *World Wide Web Conference*, Paris, France, 1996.
[51] DNS: Domain Name System. http://en.wikipedia.org/wiki/Domain_name_system.
[52] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul, "Rate of change and other metrics: A live study of the world wide web," in *USENIX Symposium on Internet Technologies and Systems*, pp. 147–158, 1997.
[53] V. Duvvuri, P. Shenoy, and R. Tewari, "Adaptive leases: A strong consistency mechanism for the World Wide Web," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 5, pp. 1266–1276, 2003.
[54] Edge Side Includes. http://www.esi.org.
[55] K. Elhardt and R. Bayer, "A database cache for high performance and fast restart in database systems," *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 503–525, 1984.
[56] C. Elkan, "Independence of logic database queries and update," in *PODS*, pp. 154–160, 1990.
[57] D. F. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini, *Economic Models for Allocating Resources in Computer Systems*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1996.
[58] M. J. Franklin, M. J. Carey, and M. Livny, "Transactional client–server cache consistency: Alternatives and performance," *ACM Transactions on Database Systems*, vol. 22, no. 3, pp. 315–363, 1997.
[59] L. Gao, A. Nayate, and J. Zheng, "Improving availability and performance with application-specific data replication," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 106–120, 2005.
[60] J. Garreau, "Our Cells, Ourselves: Planet's fastest revolution speaks to the human heart," *The Washington Post*, February 2008.
[61] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.
[62] J. Goldstein and P.-Å. Larson, "Optimizing queries using materialized views: a practical, scalable solution," in *SIGMOD Conference*, pp. 331–342, 2001.
[63] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *SOSP*, pp. 202–210, 1989.
[64] T. Griffin and L. Libkin, "Incremental maintenance of views with duplicates," in *SIGMOD Conference*, pp. 328–339, 1995.
[65] T. Griffin, L. Libkin, and H. Trickey, "An improved algorithm for the incremental recomputation of active relational expressions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 3, pp. 508–511, 1997.

[66] J. Grundy, S. Newby, T. Whitmore, and P. Grundeman, "Extending a persistent object framework to enhance enterprise application server performance," in *ADC*, pp. 57–64, 2002.

[67] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs, "Adaptive scheduling of web transactions," in *Proceedings of the 25th International Conference on Data Engineering*, April 2009.

[68] H. Guo, P.-A. Larson, R. Ramakrishnan, and J. Goldstein, "Relaxed currency and consistency: How to say "good enough" in SQL," in *SIGMOD Conference*, pp. 815–826, 2004.

[69] A. Gupta, D. Katiyar, and I. S. Mumick, "Counting solutions to the view maintenance problem," in *Workshop on Deductive Databases, JICSLP*, 1992.

[70] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Bulletin on Data Engineering*, pp. 145–157, 1995.

[71] A. Gupta and I. S. Mumick, eds., *Materialized Views: Techniques, Implementations, and Applications*. Cambridge, Mass: MIT Press, 1999.

[72] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *SIGMOD Conference*, pp. 157–166, 1993.

[73] J. Gwertzman and M. Seltzer, "World wide web cache consistency," in *USENIX Annual Technical Conference*, 1996.

[74] H. Qu and A. Labrinidis, Preference-aware query and update scheduling in web-databases, April 2007.

[75] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001.

[76] E. N. Hanson, "A performance analysis of view materialization strategies," in *SIGMOD Conference*, pp. 440–453, 1987.

[77] J. R. Haritsa, M. Livny, and M. J. Carey, "Earliest deadline scheduling for real-time database systems," in *Proceedings of RTSS '91*, pp. 232–243, IEEE Computer Society, 1991.

[78] J. M. Hellerstein and J. F. Naughton, "Query execution techniques for caching expensive methods," in *SIGMOD Conference*, pp. 423–434, 1996.

[79] J. M. Hellerstein and M. Stonebraker, *Readings in Database Systems*. Cambridge, Mass: MIT Press, 2005.

[80] B. C. Housel and D. B. Lindquist, "WebExpress: A system for optimizing web browsing in a wireless environment," in *International Conference on Mobile Computing and Networking*, pp. 108–116, 1996.

[81] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, 1988.

[82] HTTP: Hypertext Transfer Protocol. http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

[83] H. Hu, J. Xu, W. S. Wong, B. Zheng, D. L. Lee, and W.-C. Lee, "Proactive caching for spatial queries in mobile environments," in *ICDE*, pp. 403–414, 2005.

[84] P. C. K. Hung, H. Li, and J.-J. Jeng, "WS-negotiation: An overview of research issues," in *HICSS '04: Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, p. 10033.2, Washington, DC, USA: IEEE Computer Society, 2004.

[85] A. Iyengar and J. Challenger, "Data update propagation: A method for determining how changes to underlying data affect cached objects on the web," in *IBM Research Technical Report*, 1998.

[86] S. Iyer, A. Rowstron, and P. Druschel, "Squirrel: A decentralized, peer-to-peer web cache," in *PODC*, 2002.

[87] M. Ji, "Affinity-based management of main memory database clusters," *ACM Transactions on Internet Technology*, vol. 2, no. 4, pp. 307–339, 2002.

[88] L.-J. Jin, V. Machiraju, and A. Sahai, "Analysis on service level agreement of web services," Technical report, HP Laboratories, 2002.

[89] J. Jing, A. Elmagarmid, A. S. Helal, and R. Alonso, "Bit-sequences: An adaptive cache invalidation method in mobile client/server environments," *Mobile Networks and Applications*, vol. 2, no. 2, pp. 115–127, 1997.

[90] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K.-L. Tan, "An adaptive peer-to-peer network for distributed caching of OLAP results," in *SIGMOD Conference*, pp. 25–36, 2002.

[91] P. Kalnis and D. Papadias, "Proxy-server architectures for OLAP," in *SIGMOD Conference*, pp. 367–378, 2001.

[92] K.-D. Kang, S. H. Son, and J. A. Stankovic, "Managing deadline miss ratio and sensor data freshness in real-time databases," *IEEE TKDE*, vol. 16, no. 10, pp. 1200–1216, 2004.

[93] A. Keller and H. Ludwig, "The WSLA framework: Specifying and monitoring service level agreements for web services," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003.

[94] A. M. Keller and J. Basu, "A predicate-based caching scheme for client–server database architectures," *The VLDB Journal*, vol. 5, no. 1, pp. 35–47, 1996.

[95] D. Kossmann, M. J. Franklin, and G. Drasch, "Cache investment: Integrating query optimization and distributed data placement," *ACM Transactions on Database Systems*, vol. 25, no. 4, pp. 517–558, 2000.

[96] S. Kounev, B. Weis, and A. Buchmann, "Performance tuning and optimization of J2EE applications on the JBoss platform," *Journal of Computer Resource Management*, vol. 113, 2004.

[97] H. Kreger, Web services conceptual architecture (WSCA 1.0), 2001.

[98] B. Krishnamurthy and J. Wang, "On network-aware clustering of web clients," in *SIGCOMM*, pp. 97–110, 2000.

[99] B. Krishnamurthy and C. Wills, "Study of piggyback cache validation for proxy caches in the WWW," in *USENIX Symposium on Internet Technology and Systems*, pp. 1–12, 1997.

[100] B. Krishnamurthy and C. Wills, "Piggyback server invalidation for proxy cache coherency," in *World Wide Web Conference*, pp. 185–194, 1998.

[101] B. Krishnamurthy and C. E. Wills, "Proxy cache coherency and replacement — towards a more complete picture," in *ICDCS*, pp. 332–339, 1999.

[102] A. Labrinidis, "Web views," in *Encyclopedia of Database Systems*, (L. Liu and M. T. Özsu, eds.), pp. 3524–3525, Springer US, 2009.
[103] A. Labrinidis, H. Qu, and J. Xu, "Quality contracts for real-time enterprises," in *First International Workshop on Business Intelligence for the Real Time Enterprise*, 2006.
[104] A. Labrinidis and N. Roussopoulos, "Webview materialization," in *SIGMOD Conference*, pp. 367–378, 2000.
[105] A. Labrinidis and N. Roussopoulos, "Adaptive webview materialization," in *WebDB*, pp. 85–90, 2001.
[106] A. Labrinidis and N. Roussopoulos, "Update propagation strategies for improving the quality of data on the web," in *VLDB*, pp. 391–400, 2001.
[107] A. Labrinidis and N. Roussopoulos, "Exploring the tradeoff between performance and data freshness in database-driven web servers," *The VLDB Journal*, vol. 13, no. 3, pp. 240–255, 2004.
[108] P.-Å. Larson, J. Goldstein, and J. Zhou, "MTCache: Transparent mid-tier database caching in SQL server," in *ICDE*, pp. 177–189, 2004.
[109] P.-Å. Larson and H. Z. Yang, "Computing queries from derived relations," in *VLDB*, pp. 259–269, 1985.
[110] D. Lee and W. W. Chu, "Semantic caching via query matching for web sources," in *CIKM*, pp. 77–85, 1999.
[111] A. Y. Levy and Y. Sagiv, "Queries independent of updates," in *VLDB*, pp. 171–181, 1993.
[112] D. Li and D. R. Cheriton, "Scalable web caching of frequently updated objects using reliable multicast," in *USENIX Symposium on Internet Technologies and Systems*, pp. 1–12, 1999.
[113] H. Li, S. Y. W. Su, and H. Lam, "On automated e-business negotiations: Goal, policy, strategy, and plans of decision and action," *Journal of Organizational Computing and Electronic Commerce*, vol. 13, pp. 1–29, 2006.
[114] W.-S. Li, O. Po, W.-P. Hsiung, K. S. Candan, and D. Agrawal, "Freshness-driven adaptive caching for dynamic content web sites," *Data & Knowledge Engineering*, vol. 47, no. 2, pp. 269–296, 2003.
[115] C. Liu and P. Cao, "Maintaining strong cache consistency in the world-wide web," in *ICDCS*, p. 12, 1997.
[116] J. Liu, "Performance and scalability measurement of COTS EJB technology," in *SBAC-PAD*, p. 212, 2002.
[117] T. Loukopoulos, P. Kalnis, I. Ahmad, and D. Papadias, "Active caching of on-line-analytical-processing queries in WWW proxies," in *ICPP*, pp. 419–426, 2001.
[118] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton, "Middle-tier database caching for e-business," in *SIGMOD Conference*, pp. 600–611, 2002.
[119] Q. Luo and J. F. Naughton, "Form-based proxy caching for database-backed web sites," in *VLDB*, pp. 191–200, 2001.
[120] Q. Luo, J. F. Naughton, R. Krishnamurthy, P. Cao, and Y. Li, "Active query caching for database web servers," in *WebDB (Selected Papers)*, pp. 92–104, 2000.

[121] Q. Luo, J. F. Naughton, and W. Xue, "Form-based proxy caching for database-backed web sites: Keywords and functions," *The VLDB Journal*, 2007.

[122] Q. Luo and W. Xue, "Template-based proxy caching for table-valued functions," in *DASFAA*, pp. 339–351, 2004.

[123] T. Malik, R. C. Burns, and A. Chaudhary, "Bypass caching: Making scientific databases good network citizens," in *ICDE*, pp. 94–105, 2005.

[124] T. Malik, R. C. Burns, N. V. Chawla, and A. Szalay, "Estimating query result sizes for proxy caching in scientific database federations," in *Supercomputing*, p. 36, 2006.

[125] J. C. Mogul, "Recovery in spritely NFS," *Computing Systems*, vol. 7, no. 2, pp. 201–262, 1994.

[126] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for http," in *SIGCOMM*, pp. 181–194, 1997.

[127] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the sprite network file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 134–154, 1988.

[128] M. Nottingham, "Optimizing object freshness controls in web caches," in *Web Caching and Content Delivery Workshop*, 1999.

[129] C. Olston and J. Widom, "Best-effort cache synchronization with source cooperation," in *SIGMOD Conference*, pp. 73–84, 2002.

[130] Özgür Ulusoy and G. G. Belford, "Real-time transaction scheduling in database systems," *Information Systems*, vol. 18, no. 9, pp. 559–580, 1993.

[131] S. Papastavrou, G. Samaras, P. Evripidou, and P. K. Chrysanthis, "A decade of dynamic web content: A structured survey on past and present practices and future trends," *IEEE Communications Surveys and Tutorials*, vol. 8, no. 1–4, pp. 52–60, 2006.

[132] S. Patro and Y. C. Hu, "Transparent query caching in peer-to-peer overlay networks," in *IPDPS*, p. 32, 2003.

[133] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pp. 165–178, New York, NY, USA: ACM, 2009.

[134] X. Qian and G. Wiederhold, "Incremental recomputation of active relational expressions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, no. 3, pp. 337–341, 1991.

[135] M. Rabinovich and O. Spatschek, *Web Caching and Replication*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[136] M. Rabinovich, Z. Xiao, F. Douglis, and C. R. Kalmanek, "Moving edge-side includes to the real edge — The clients," in *USENIX Symposium on Internet Technologies and Systems*, 2003.

[137] A. Rajaraman, Y. Sagiv, and J. D. Ullman, "Answering queries using templates with binding patterns," in *PODS*, pp. 105–112, 1995.

[138] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglis, "Automatic fragment detection in dynamic web pages and its impact on caching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 6, pp. 859–874, 2005.

[139] L. Ramaswamy and L. Liu, "An expiration age-based document placement scheme for cooperative web caching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 5, pp. 585–600, 2004.
[140] L. Ramaswamy, L. Liu, and A. Iyengar, "Cache clouds: Cooperative caching of dynamic documents in edge networks," in *ICDCS*, pp. 229–238, 2005.
[141] L. Ramaswamy, L. Liu, and J. Zhang, "Efficient formation of edge cache groups for dynamic content delivery," in *ICDCS*, p. 43, 2006.
[142] G. D. Rodosek and L. Lewis, "Dynamic service provisioning: A user-centric approach," in *Proceedings of the 12th International Workshop on Distributed Systems: Operations and Management (DSOM'01)*, 2001.
[143] P. Rodriguez, C. Spanner, and E. W. Biersack, "Analysis of web caching architectures: Hierarchical and distributed caching," *IEEE/ACM Transactions on Networking*, vol. 9, no. 4, pp. 404–418, 2001.
[144] N. Roussopoulos and H. Kang, "Principles and techniques in the design of ADMS±," *IEEE Computer*, vol. 19, no. 12, pp. 19–25, 1986.
[145] M. Satyanarayanan, "The evolution of Coda," *ACM Transactions on Computer Systems*, vol. 20, no. 2, pp. 85–124, 2002.
[146] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan, "Incremental maintenance of path-expression views," in *SIGMOD Conference*, pp. 443–454, 2005.
[147] B. Schroeder and M. Harchol-Balter, "Web servers under overload: How scheduling can help," *ACM Transactions on Internet Technology*, vol. 6, no. 1, pp. 20–52, 2006.
[148] S. Shah, K. Ramamritham, and P. J. Shenoy, "Maintaining coherency of dynamic data in cooperating repositories," *VLDB*, pp. 526–537, 2002.
[149] M. A. Sharaf, P. K. Chrysanthis, A. Labrinidis, and K. Pruhs, "Algorithms and metrics for processing multiple heterogeneous continuous queries," *ACM Transactions on Database Systems (TODS)*, pp. 5.1–5.44, March 2008.
[150] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein, "ACMS: The Akamai configuration management system," in *NSDI*, pp. 245–258, 2005.
[151] B. Smith, A. Acharya, T. Yang, and H. Zhu, "Exploiting result equivalence in caching dynamic web content," in *USENIX Symposium on Internet Technologies and Systems*, 1999.
[152] G. F. Stefano Di Paola, "Subverting Ajax," in *CCC*, 2006.
[153] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin "Mapreduce and parallel DBMSs: Friends or foes?" *Communications of the ACM*, vol. 53, no. 1, pp. 64–71, 2010.
[154] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A wide-area distributed database system," *The VLDB Journal*, vol. 5, no. 1, 1996.
[155] S. Y. Su, C. Huang, J. Hammer, Y. Huang, H. Li, L. Wang, Y. Liu, C. Pluempitiwiriyawej, M. Lee, and H. Lam, "An Internet-based negotiation server for e-commerce," *The VLDB Journal*, vol. 10, no. 1, pp. 72–90, 2001.
[156] A. S. Szalay, J. Gray, A. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. vandenBerg, "The SDSS skyserver: Public access to the sloan digital sky server data," in *SIGMOD Conference*, pp. 570–581, 2002.

[157] X. Tang and S. T. Chanson, "Coordinated management of cascaded caches for efficient content distribution," in *ICDE*, pp. 37–48, 2003.
[158] X. Tang, S. T. Chanson, H. Chi, and C. Lin, "Session-affinity aware request allocation for web clusters," in *ICDCS*, pp. 142–149, 2004.
[159] Tangosol. http://www.tangosol.com.
[160] The TimesTen Team, "In-memory data management in the application tier," in *ICDE*, p. 637, 2000.
[161] The TimesTen Team, "Mid-tier caching: The timesten approach," in *SIGMOD Conference*, pp. 588–593, 2002.
[162] P. Triantafillou and C. Neilson, "Achieving strong consistency in a distributed file system," *IEEE Transactions on Software Engineering*, vol. 23, no. 1, pp. 35–55, 1997.
[163] D. Verma, "Service level agreements on IP networks," *Proceedings of the IEEE*, vol. 92, pp. 1382–1388, 2004.
[164] D. Vista, "Optimizing incremental view maintenance expressions in relational databases," PhD thesis, University of Toronto, 1997.
[165] J. Wang, "A survey of web caching schemes for the Internet," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, pp. 36–46, 1999.
[166] Y. Wang and L. A. Rowe, "Cache consistency and concurrency control in a client/server DBMS architecture," in *SIGMOD Conference*, pp. 367–376, 1991.
[167] W. K. Wilkinson and M.-A. Neimat, "Maintaining consistency of client-cached data," in *VLDB*, pp. 122–133, 1990.
[168] A. Wolman, "Sharing and caching characteristics of internet content," PhD thesis, University of Washington, 2002.
[169] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy, "Organization-based analysis of web-object sharing and caching," *USENIX Symposium on Internet Technologies and Systems*, 1999.
[170] K. Worrell, "Invalidation in large scale network object caches," Master thesis, University of Colorado, Boulder, 1994.
[171] S. Wu and C.-C. Liao, "Virtual proxy servers for WWW and intelligent agents on the internet," in *HICSS*, p. 200, 1997.
[172] L. Xiao, X. Zhang, A. Andrzejak, and S. Chen, "Building a large and efficient hybrid peer-to-peer internet caching system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 6, pp. 754–769, 2004.
[173] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez, "Caching strategies for data-intensive web sites," in *VLDB*, pp. 188–199, 2000.
[174] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Using leases to support server-driven consistency in large-scale systems," in *ICDCS*, pp. 285–294, 1998.
[175] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Volume leases for consistency in large-scale systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 4, pp. 563–576, 1999.
[176] H. Yu and A. Vahdat, "Efficient numerical error bounding for replicated network services," in *VLDB*, pp. 123–133, 2000.

[177] Q. Zhang, W. Sun, E. Smirni, and G. Ciardo, "Workload-aware load balancing for clustered web servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 3, pp. 219–233, 2005.

[178] Y. Zhuge and H. Garcia-Molina, "Graph structured views and their incremental maintenance," in *ICDE*, pp. 116–125, 1998.

[179] F. Zulkernine, P. Martin, C. Craddock, and K. Wilson, "A policy-based middleware for web services SLA negotiation," in *Proceedings of the IEEE International Conference on Web Services (ICWS08)*, 2008.