
Caching and Materialization for Web Databases

Caching and Materialization for Web Databases

Alexandros Labrinidis

*University of Pittsburgh
USA*

labrinid@cs.pitt.edu

Qiong Luo

*Hong Kong University of Science and Technology
Hong Kong*

luo@cse.ust.hk

Jie Xu

*University of Pittsburgh
USA*

xujie@cs.pitt.edu

Wenwei Xue

*Hong Kong University of Science and Technology
Hong Kong*

wwxue@cse.ust.hk

now

the essence of knowledge

Boston – Delft

Foundations and Trends[®] in Databases

Published, sold and distributed by:

now Publishers Inc.
PO Box 1024
Hanover, MA 02339
USA
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:

now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is A. Labrinidis, Q. Luo, J. Xu and W. Xue, Caching and Materialization for Web Databases, Foundation and Trends[®] in Databases, vol 2, no 3, pp 169–266, 2009

ISBN: 978-1-60198-324-4

© 2010 A. Labrinidis, Q. Luo, J. Xu and W. Xue

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, mechanical, photocopying, recording or otherwise, without prior written permission of the publishers.

Photocopying. In the USA: This journal is registered at the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923. Authorization to photocopy items for internal or personal use, or the internal or personal use of specific clients, is granted by now Publishers Inc for users registered with the Copyright Clearance Center (CCC). The 'services' for users can be found on the internet at: www.copyright.com

For those organizations that have been granted a photocopy license, a separate system of payment has been arranged. Authorization does not extend to other kinds of copying, such as that for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. In the rest of the world: Permission to photocopy must be obtained from the copyright owner. Please apply to now Publishers Inc., PO Box 1024, Hanover, MA 02339, USA; Tel. +1-781-871-0245; www.nowpublishers.com; sales@nowpublishers.com

now Publishers Inc. has an exclusive license to publish this material worldwide. Permission to use this content must be obtained from the copyright license holder. Please apply to now Publishers, PO Box 179, 2600 AD Delft, The Netherlands, www.nowpublishers.com; e-mail: sales@nowpublishers.com

Foundations and Trends[®] in Databases

Volume 2 Issue 3, 2009

Editorial Board

Editor-in-Chief:

Joseph M. Hellerstein

Computer Science Division

University of California, Berkeley

Berkeley, CA

USA

hellerstein@cs.berkeley.edu

Editors

Anastasia Ailamaki (EPFL)

Michael Carey (UC Irvine)

Surajit Chaudhuri (Microsoft Research)

Ronald Fagin (IBM Research)

Minos Garofalakis (Yahoo! Research)

Johannes Gehrke (Cornell University)

Alon Halevy (Google)

Jeffrey Naughton (University of Wisconsin)

Christopher Olston (Yahoo! Research)

Jignesh Patel (University of Michigan)

Raghu Ramakrishnan (Yahoo! Research)

Gerhard Weikum (Max-Planck Institute)

Editorial Scope

Foundations and Trends® in Databases covers a breadth of topics relating to the management of large volumes of data. The journal targets the full scope of issues in data management, from theoretical foundations, to languages and modeling, to algorithms, system architecture, and applications. The list of topics below illustrates some of the intended coverage, though it is by no means exhaustive:

- Data Models and Query Languages
- Query Processing and Optimization
- Storage, Access Methods, and Indexing
- Transaction Management, Concurrency Control and Recovery
- Deductive Databases
- Parallel and Distributed Database Systems
- Database Design and Tuning
- Metadata Management
- Object Management
- Trigger Processing and Active Databases
- Data Mining and OLAP
- Approximate and Interactive Query Processing
- Data Warehousing
- Adaptive Query Processing
- Data Stream Management
- Search and Query Integration
- XML and Semi-Structured Data
- Web Services and Middleware
- Data Integration and Exchange
- Private and Secure Data Management
- Peer-to-Peer, Sensornet and Mobile Data Management
- Scientific and Spatial Data Management
- Data Brokering and Publish/Subscribe
- Data Cleaning and Information Extraction
- Probabilistic Data Management

Information for Librarians

Foundations and Trends® in Databases, 2009, Volume 2, 4 issues. ISSN paper version 1931-7883. ISSN online version 1931-7891. Also available as a combined paper and online subscription.

Caching and Materialization for Web Databases

Alexandros Labrinidis¹, Qiong Luo²,
Jie Xu³ and Wenwei Xue⁴

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,
wwwxue@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.

Contents

1	Introduction	1
2	Typical Architecture	5
3	Store How are Data Cached/Materialized?	9
3.1	Store Location of Caching/Materialization	9
3.2	Store Unit of Caching/Materialization	12
3.3	Store Selecting Content for Caching/Materialization	12
4	Use Using Cached Content	15
4.1	Use Query Scheduling	15
4.2	Use Query Processing	17
5	Maintain Cache Maintenance	23
5.1	Maintain Cache Initialization: Proactive versus Reactive	23
5.2	Maintain Timing for Updates	25
5.3	Maintain Processing of Updates	31
5.4	Maintain Scheduling of Updates	34
5.5	Maintain Cache Replacement Policies	35

6	Metrics Performance and Quality Metrics	37
6.1	[Metrics] QoS Metrics	37
6.2	[Metrics] QoD Metrics	40
6.3	[Metrics] Quality Contracts	43
6.4	[Metrics] Service Level Agreements	44
7	Projects	49
7.1	Browser Caches	49
7.2	Proxy Caches	50
7.3	Server-Side Caches	53
8	Related Work in Other Areas	57
8.1	General Web Caching	57
8.2	Database Caching and Materialized Views	61
8.3	Caching in Client–Server Databases	67
8.4	Caching in Distributed Databases	69
8.5	Caching in Distributed Systems	71
8.6	Industrial Products/Standards	78
9	Open Research Problems	81
9.1	Cloud Computing	81
9.2	User-Centric Computing	82
9.3	Mobile Computing	83
9.4	Green Computing	84
9.5	Non-technical Challenges	85
10	Summary	87
	Acknowledgments	89
	References	91

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

2 Introduction

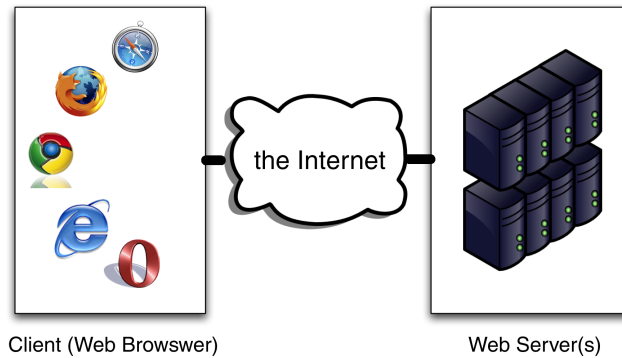


Fig. 1.1 10,000-feet view of a typical web-database architecture.

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).

4 *Introduction*

- **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-foot 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

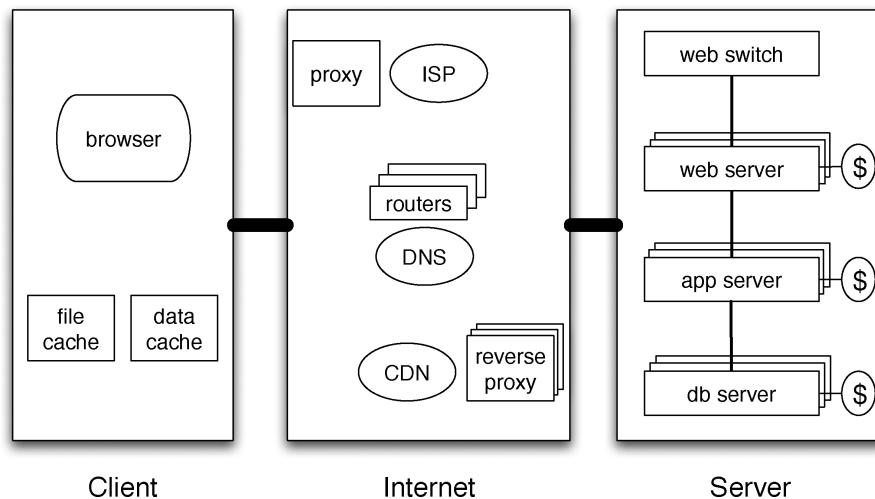


Fig. 2.1 Typical web-database architecture (\$ signs indicate possible location for caches at the server).

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 adCenter³) 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/>

Analytics⁴). 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 website⁵ 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

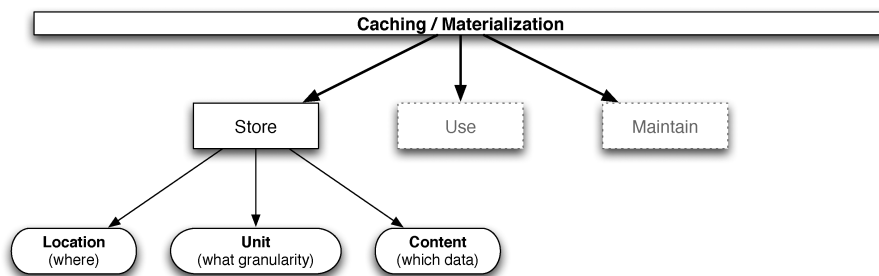


Fig. 3.1 Taxonomy for store question.

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”. AJAX¹ 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_\(programming\)](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, $\text{OVIS}(\theta)$. Given the user-specified data freshness threshold θ , $\text{OVIS}(\theta)$ operates in two modes: passive and active. In the passive mode, $\text{OVIS}(\theta)$ 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 , $\text{OVIS}(\theta)$ 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 $\text{OVIS}(\theta)$ meets the response-time constraint.

In general, the selection problem is closely related to the maintenance policy or policies available. For example, the $\text{OVIS}(\theta)$ 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

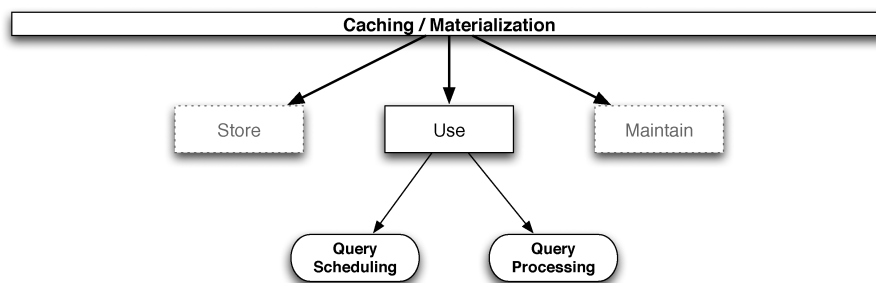


Fig. 4.1 Taxonomy for use question.

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].

Full-edged SQL Query Processing	✓	✓		
Semantic Query Processing	✓	✓	✓	
Containment-based Query Processing	✓	✓	✓	✓
No Query Processing	✓	✓	✓	✓
	Raw Data	Views	Fragments	HTML Pages

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

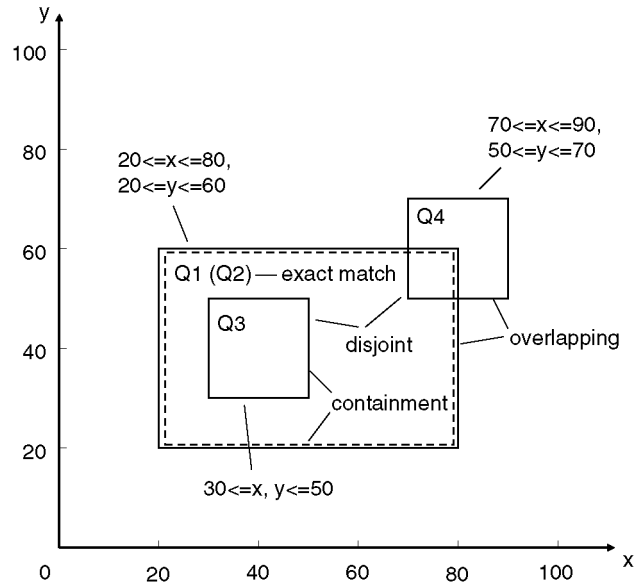


Fig. 4.3 Illustration of query relationships.

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

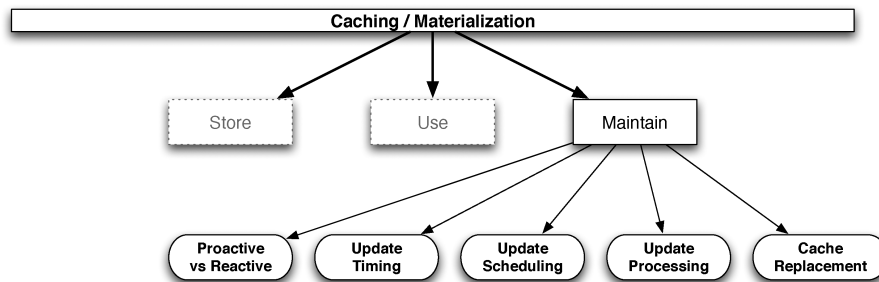


Fig. 5.1 Taxonomy for maintain question.

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 → 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).

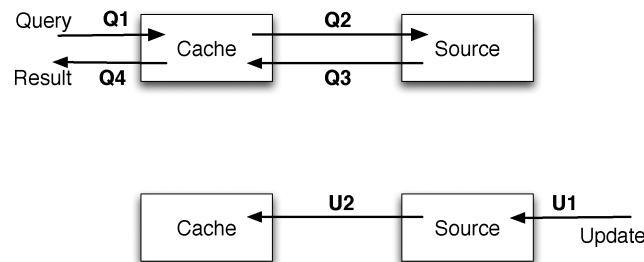


Fig. 5.2 Query processing under proactive and reactive caching.