SOCA
DOI 10.1007/s11761-011-0094-7

SPECIAL ISSUE PAPER

A data access framework for service-oriented rich clients

**Qi Zhao · Xuanzhe Liu · Xingrun Chen ·
Jiyu Huang · Gang Huang · Hong Mei**

Received: 15 April 2011 / Revised: 21 October 2011 / Accepted: 24 October 2011
© Springer-Verlag London Limited 2011

Abstract Facilitated by the SOA and new Web technologies, Service-Oriented Rich Clients (SORCs) compose various Web-delivered services in Web browser to create new applications. The SORCs support client-side data storage and manipulation and provide more features than traditional thin clients. However, the SORCs might suffer from data access issues, mainly due to both client-side incompatible data sources and server-side improper or even undesirable cache strategies. Addressing the data access issues, this paper proposes a data access framework for SORCs. The main contributions of this paper are as follows. First, the framework makes the SORCs accommodate heterogeneous local storage solutions and diverse Web browsers properly. The framework abstracts the underlying details of different local storages and selects the most proper data sources for current SORC in use. Secondly, the framework provides a cache mechanism, which supports client-side customized cache strategies. An adaptive technique for the strategies is also proposed to adjust cache strategies based on users' historical actions to achieve better performance.

Q. Zhao · X. Liu (✉) · X. Chen · J. Huang · G. Huang · H. Mei
Key Laboratory of High Confidence Software Technologies,
Ministry of Education, School of Electronics Engineering
and Computer Science, Peking University, Beijing 100871, China
e-mail: liuxzh@sei.pku.edu.cn

Q. Zhao
e-mail: zhaoqi06@sei.pku.edu.cn

X. Chen
e-mail: chenxr07@sei.pku.edu.cn

J. Huang
e-mail: huangjy07@sei.pku.edu.cn

G. Huang
e-mail: huanggang@sei.pku.edu.cn

H. Mei
e-mail: meih@pku.edu.cn

Keywords Data access · Rich client · Service composition

1 Introduction

In the recent a few years, the Web applications have evolved a lot. At the server-side, Service-Oriented Architecture (SOA) has been widely adopted. The servers publish data and functionalities through some Web-delivered services (such as SOAP and RESTful Web-delivered services, RSS/Atom feeds and so on). At the client-side, the Web browsers are now capable of providing rich user experiences facilitated by the popular Web technologies (such as HTML 5, JavaScript). Therefore, newly emerging Service-Oriented Rich Clients (SORCs) can compose various Web-delivered services to create Rich Internet Applications (or RIA for short) [1–3]. For example, the new version of Twitter[1] is a typical RIA that is built upon its own APIs (a group of RESTful services). The client-side of new Twitter is a SORC, which no longer displays only HTML pages, but retrieves data from the server-side APIs and deals with the data and generates user interface (UI). SORCs can cope with several limitations of the thin clients supported in traditional Web applications, such as poor user experience, unnecessary round-trip server access [4].

With the evolution of the application architecture, the data storage and manipulation at the client-side evolve as well. In the traditional Web applications, the servers and the client exchange Web pages by means of forms where data and presentation elements are mixed. The browsers just directly display the pages. However, with the wide adoption of Service-Oriented Computing, the SORCs can leverage the Web-delivered services. Data exchanged between the servers

[1] Twitter: http://www.twitter.com.

Published online: 06 November 2011

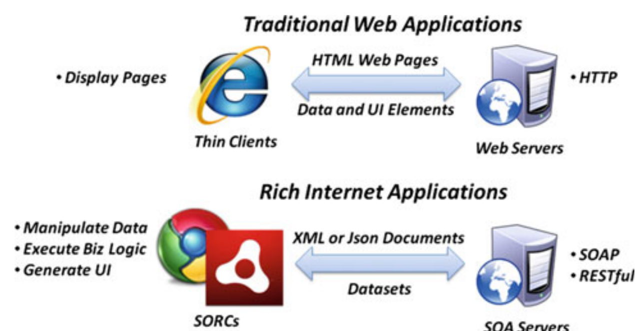

Fig. 1 Comparison of the thin clients and the SORCs

and the SORCs are no longer pages and forms, but structured datasets in form of XML or JSON.[2] The SORCs parse and cache the results of service invocations, execute business logics and then generate user interface (UI). Compared with the thin clients, the SORCs are able to store and manipulate structured datasets at the client-side. Such capability makes the SORCs provide more interactive UI, which may be very close to the experience of desktop clients. Moreover, since the datasets are in more logical structures, the SORCs can retrieve them in asynchronous and intelligent fashion, hence significantly reduce the network traffic overhead [5] (Fig. 1).

The client-side data storage and manipulation bring some challenging issues to the SORCs. The SORCs suffer from data access issues when they retrieve, persist and cache data. For example, when storing data at local storage, the SORCs might face the incompatible client-side storage solutions (e.g. Flash LSO[3], HTML 5[4]or Google Gears[5]) and heterogeneous Web browsers (e.g. IE or Firefox). Another example is that, the cache mechanism of the SORCs could apply object-level strategies to parse the relationships among the datasets and eliminate cache redundancy. Unfortunately, current HTTP cache in browsers is designed for caching the Web pages. It just caches a page as an indivisible body and cannot handle the relationships among datasets. Such data access issues increase the difficulties in SORCs development.

The traditional Web applications also suffer from the data access issues when their application servers (e.g. Weblogic or JBoss) access the diverse databases (e.g. Oracle and MySQL). In this situation, the server-side data access frameworks (e.g. Hibernate[6]) take charge of coordinating applications and data sources [6] and make data access simpler and more efficient [7,8]. Consequently, to address the data access issues for SORCs, we believe that a data access framework for SORCs is required.

In this paper, we propose a data access framework for SORCs to resolve the client-side data access issues. The main contributions of this paper are the following:

- The framework makes the SORCs capable of accommodating heterogeneous local storage solutions and diverse Web browsers properly. The framework abstracts the underlying details of different local storages, coordinates the incompatible issues and provides a unified local data source. A local data sources selection approach can assist the SORCs to select the most proper data sources based on current SORC and Web browser being used;
- The framework provides a cache mechanism. The cache mechanism allows the SORCs developers to customize client-side cache strategies and resolves the data access issues caused by improper cache strategies. Our mechanism provides some useful features, including the fine granularity of cache strategies, client-side expired time and so on. An adaptive technique for the strategies is also proposed. The adaptive approach adjusts cache strategies based on users' historical actions to achieve better performance.

The rest of the paper is organized as follows. In Sect. 2, we introduce the data access issues for the SORCs. Sections 3 and 4 present our solutions for the data access issues. Section 5 describes an illustrative case and provides the experimental evaluation. Finally, we make some discussions in Sect. 6 and conclude this paper in Sect. 7.

2 Data access issues for service-oriented rich clients

In the SORCs, contents can reside in the client-side data sources as persistent client-side objects, rather than in the traditional server-side databases, due to the emerging local storage technologies [9]. Accordingly, the SORCs access the data sources that can be at the server-side or at the client-side. These two data sources might be used separately for different storage requirements, or used together when necessary [10]. However, we argue that, when accessing different types of data sources, the SORCs might suffer from some challenging issues as follows.

2.1 Challenging issues for accessing client-side data in SORCs

Rather than the traditional thin clients, a major feature of SORCs is the capability to store and process data directly at client-side. The client-side storage and computing capacities make SORCs capable of providing richer and more

[2] JSON: http://www.json.org.

[3] Flash Local Shared Objects, http://www.adobe.com/products/flashplayer/articles/lso/.

[4] HTML 5, http://www.w3.org/TR/html5/.

[5] Google Gears, http://gears.google.com/.

[6] Hibernate, http://www.hibernate.org.

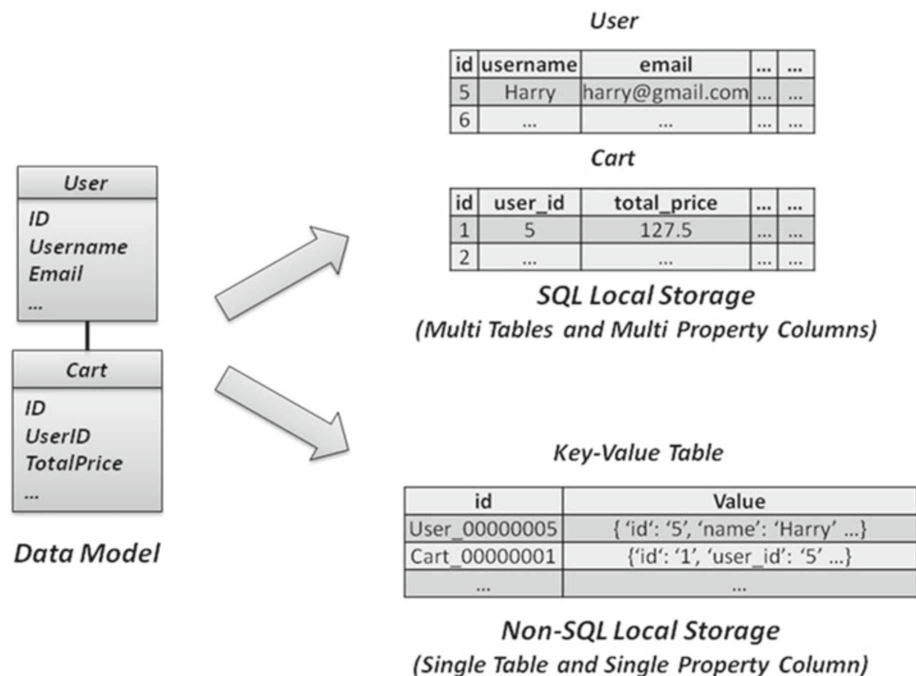

Fig. 2 Heterogeneous local storages

interactive UI. It also significantly reduces network traffic by leveraging intelligent and asynchronous requests that only deliver small blocks of data. Therefore, with the increasing number of SORCs, the requirements of local storage rise rapidly. It indicates that SORCs should retrieve and persist data at client-side (i.e. local), or even have to use local storage [9,10]. For example, if an entity represents content owned by an individual user, it had better do local persistence to reduce network round-trips. Furthermore when working in offline mode, the SORCs might retrieve data from local data sources temporarily. For these reasons, a number of local storage solutions emerge, e.g. Flash LSO and HTML 5. The solutions provide client-side data sources and allow the SORCs to store data locally. All currently popular Web browsers natively support the local storage more or less. Some old browsers (e.g. IE 6) might provide the similar capacity via plug-ins.

Although the requirements and technologies seem to be ready for SORCs, the local storage option has not been widely used. The fundamental reason is that the SORCs suffer from the heterogeneous local storage solutions and browsers.

The local storage solutions can be categorized into two types. The first solution refers to "SQL local storage". It embeds lightweight relational databases (e.g. SQLite[7]) into browsers and provides SQL query APIs. The second solution is called "non-SQL" local storage. It offers table-based key-value data sources—each SORC has one table and each table only has a key column and a value column. The table supports simple CRUD (Create, Read, Update and Delete) operations rather than powerful query languages. We argue that the former one is more powerful, while the latter is easier to be used. In practice, the two solutions usually support different requirements. However, they are not completely compatible.

Furthermore, current SORCs are mainly based Object-Oriented (OO) technologies. Therefore, the classic "impedance mismatch" issues [11] still exist, between data objects and local data sources. The Object-Relational Mapping (ORM), between objects and SQL databases, has obtained a large body of research in Database Management Systems (DBMS) [12,13]. On the other hand, the mapping between objects and non-SQL tables should also be considered. Figure 2 indicates how to store OO data models with SQL and non-SQL local storages. Classic ORM maps an OO class to one SQL database table, and each property of the class to one column in the table. The relations among OO classes are realized by foreign keys among tables. When mapping objects to a non-SQL key-value table, each object is realized as a line of the table. The type and ID of an object refer to the key, and the object is serialized as a string and saved in the value column.

Each solution has several implementations. For example, the SQL local storage is supported by "Google Gears" and "HTML5 database" and "IE userData," and "Flash LSO" and "HTML5 local storage" solutions support non-SQL data sources. Each implementation has its own features and APIs. For example, in Google Gears, SQL queries execute in a synchronous model. HTML 5 database deals with every SQL query asynchronously, and the result handler should be registered as a callback function. The APIs of the two solutions are quite different as shown in Fig. 3. Therefore, the

[7] SQLLite: http://www.sqlite.org/.

```
//---Google Gears Sample---
var db = google.gears.factory.create('beta.database');
db.open('demo-app');
//Synchronous SQL execution
var user = User(db.execute('select * from User where id=5'));

//---HTML 5 Database Sample---
var db = openDatabase("demo-app", "1.0");
var user;
//Asynchronous SQL execution
db.transaction(function(tx) {
  tx.executeSql("select * from User where id=5", [], function (tx, result) {
    user = User(result);
  });
});
```

Fig. 3 Incompatible APIs for different SQL local storage solutions

Table 1 Local storages support in major Web browsers

	IE	Firefox	Chrome	Safari	IE (mobile)	Safari (iPhone)
IE userData	5.5+	N/A	N/A	N/A	N/A	N/A
Flash LSO	Plug-in	Plug-in	Plug-in	Plug-in	N/A	N/A
HTML 5 Local	8.0+	3.5+	3.0+	3.1+	N/A	3.1+ (OS 2+)
HTML 5 Database	N/A	N/A	3.0+	3.1+	N/A	3.1+ (OS 2+)
Google Gears	6.0+, plug-in	1.5+, plug-in	Default	N/A	N/A	N/A

SORCs suffer from the incompatible APIs when accessing the diverse local data sources.

In practice, the issues for accessing client-side data sources are even more complex, since the Web browsers plays as the runtime for the SORCs. Each browser supports different local storage solutions, as summarized in Table 1.[8,9,10,11]

Therefore, if a SORC only works with a specific local storage solution (e.g. Google Gears), it may not be able to execute in some browsers (e.g. Safari). Unfortunately, in the Internet, the SORCs cannot know the user browser before users actually use them. As a result, selecting an available local storage for current browser is a challenging issue for SORCs. Furthermore, the goal of local storages selection may be not only to find an available local storage but also try to determine which one is the most suitable, since one browser may support several local data sources with different characteristics, such as performance and size limit.

2.2 Challenging issues for accessing server-side data in SORCs

Traditionally, the server-side data sources, such as DBMS, are also heterogeneous [6]. However, SOA standardizes how to exchange data between the servers and the clients via standard Web-delivered service protocols (SOAP or RESTful) [14]. The standards can shield the heterogeneity of the server-side data sources. In this way, our SORCs data access framework does not have to care about the server-side heterogeneity issues.

In such conditions, the server-side data access issues mainly focus on connections between the SORCs and the server-side data sources, especially cache mechanism. Current SORCs already employ HTTP cache in order to improve network performance. HTTP cache strategies[12] are customized at the server-side and brought into effect by browsers. However, the HTTP cache is designed for the thin clients [15] in traditional Web applications and might be improper or even undesirable in the SORCs' context. The unsuited cache mechanism leads to unnecessary communications and user experience issues.

The SORCs obtain structured datasets from the servers rather than indivisible Web pages. The HTTP cache regards the datasets as the indivisible body, which cannot be cut separately. Nevertheless, the data objects in different

[8] IE userData supported platforms: http://msdn.microsoft.com/en-us/library/ms531424%28v=vs.85%29.aspx.

[9] Flash supported platforms: http://www.adobe.com/products/flashplayer/systemreqs/.

[10] HTML 5 supported platforms: http://en.wikipedia.org/wiki/Comparison_of_layout_engines_%28HTML_5%29#cite_note-177.

[11] Google Gears supported platforms: http://en.wikipedia.org/wiki/Google_Gears.

[12] HTTP/1.1, http://www.w3.org/Protocols/rfc2616/rfc2616.html.

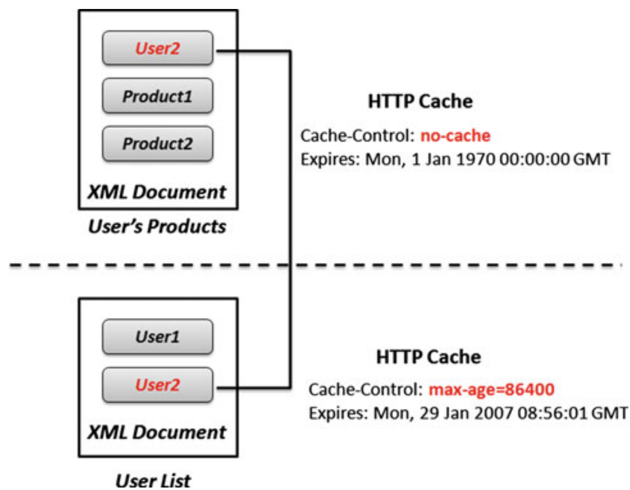

Fig. 4 HTTP cache issues

datasets may be duplicated in many cases. For example, a users list contains several users' detail, which can be cached for future use. However, the HTTP cache cannot cache these details separately (Fig. 4).

Furthermore, another potential problem of cache strategies at the service-side is that they might not satisfy the customized cache requirements of the SORCs. For instance, the SORCs access server-side data sources via both stable network connections (e.g. using cable) and instable network connections (e.g. WiFi or 3G). When in the instable network connections, the SORCs may lose requests and responses. In the offline status, the SORCs have to cache data until network connections recover. However, the HTTP cache cannot be aware of the network context and discard the "expired" data. Moreover, due to the increasing number of SORCs, a new type of Web applications, called Mashups [16], have become popular. Mashups combine several existing server-side data sources into a single Web application. Such SORCs often use server-side data in unexpected ways. The server-side cache strategies are therefore not suitable generally.

3 Adaptive access of client-side data

There are heterogeneous local storage solutions, and different Web browsers support different solutions. Therefore, when the SORCs access the local storage, the data access framework has two main functions: (1) to adapt heterogeneous client-side data sources and provide a set of unified data access APIs; (2) to select a proper local storage solution for current SORC and browser being used (Fig. 5).

3.1 Adaptation of heterogeneous data sources

Almost all data access frameworks face the impedance mismatch problem. Our data access framework should cope with the mismatch between objects and local data sources. Currently, there are two patterns, ActiveRecord[13] and DAO (Data Access Object),[14] to address the mismatch issues between Object-Oriented applications and data sources. In the DAO pattern, a data access object, e.g. *session*, takes charge of dealing with all data source–related operation, e.g. *session.save (user1)* in Hibernate. In the ActiveRecord pattern, the data objects (called ActiveRecords) perform persistent operations by themselves, e.g. *user1.save()* in RoR.[15] The ActiveRecord pattern provides a more intuitive and convenient persistent way [17]. However, the pattern realization depends on reflection mechanism and is hard to be implemented in most compiled languages, since it requires weaving persistent methods into data objects dynamically. Our data access framework is realized at the client-side, where the programming languages (e.g. JavaScript) are always dynamic, weak-typing and interpreted. Accordingly, our framework adopts ActiveRecord pattern.

The framework allows developers to define a metadata on each data model, as shown in Fig. 6. The metadata includes the name and type of each property that should be persisted. Then, when a SORC is initialized, the framework will read all metadata and weave a series of persistent methods into data objects, e.g. *User.find, user.save, user.update* and so on.

The ActiveRecords can be mapped to the heterogeneous data sources. As shown in Fig. 7, a data object is persisted as a row of a specific table in SQL database, or JSON string indexed by type and ID in the non-SQL key-value table. If a persistent method (e.g. *save*) is invoked, the framework will translate the request into the specific operation on current used data source (e.g. SQL *"insert"* in SQL database or *setItem* method in key-value table).

The SQL database adapter and key-value table adapter address the issue about the incompatible APIs of the same kinds of the solutions. The adapters encapsulate the widely different APIs of diverse storage solutions and expose a set of unified APIs. The unified APIs provide the general functionalities, such as CRUD and simple aggregate operations. However, if the SORCs need some specific capacities, e.g. complex SQL queries, they may still have to use solution-specific APIs.

3.2 Selection of proper data sources

Although our framework allows the SORCs to store data objects into different data sources via a unified way, a key issue still remains —how to determine which local storages are most proper in current context. We argue that the decision-making depends on two aspects.

[13] ActiveRecord Pattern, http://en.wikipedia.org/wiki/Active_record_patter.

[14] DAO Pattern, http://en.wikipedia.org/wiki/Data_access_object.

[15] Ruby on Rails, http://rubyonrails.org/.

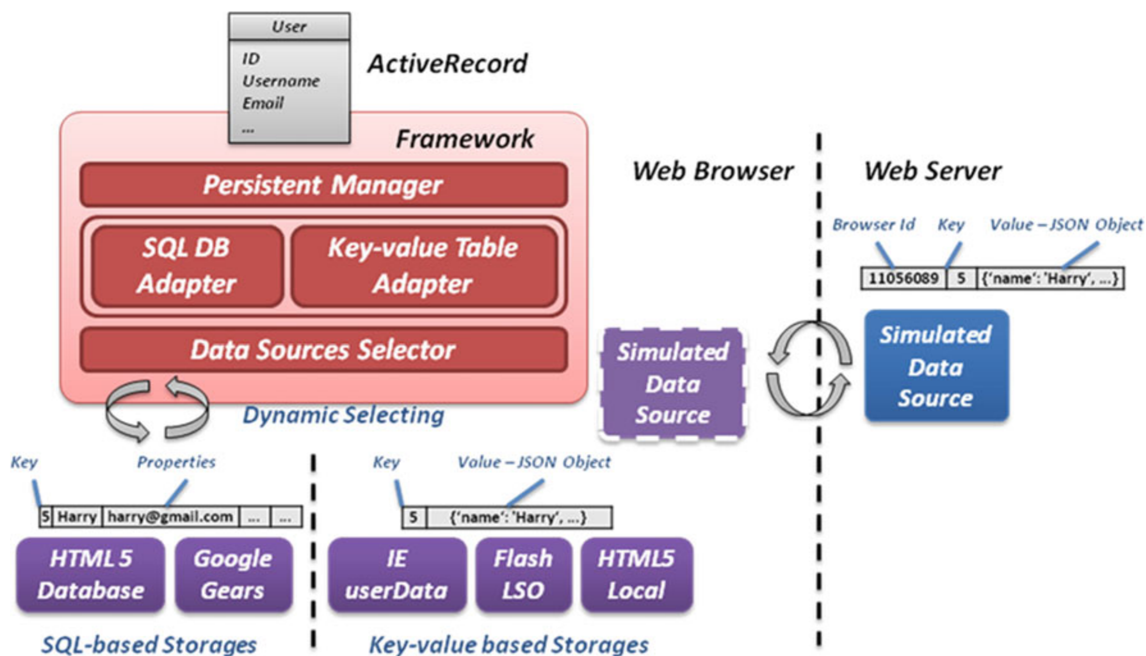

Fig. 5 Adaptation of client-side data sources

```
//User Class Metadata
{ properties:
    {
      'id':{type:'string', primaryKey:true},
      'username':{type:'string'},
      'email':{type:'string'},
      ...
    },
  ...
}
```

Fig. 6 Metadata of data model

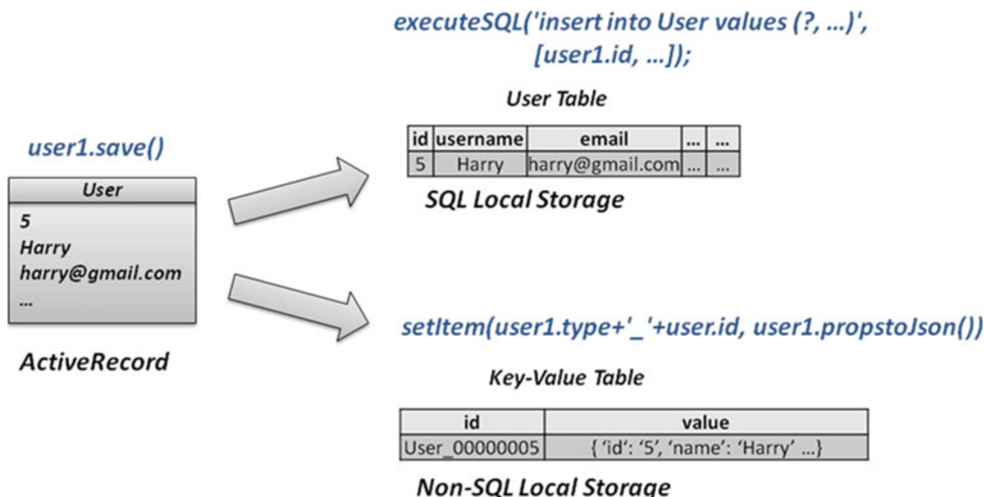

Fig. 7 Mapping objects to different data sources

The first factor is current browser that is used by the user. As mentioned previously, the different browsers support different solutions. The available local storages therefore depend on the current browser. The data source selection is made at runtime, since there is no way to prejudge what browser is used. The framework finds the available data sources via:

(1) The "user-agent" field, e.g. "*Mozilla/5.0 Gecko/20100401 Firefox/3.6.3*", which indicates the browsers' types and versions (e.g. *Firefox 3.6.3*);
(2) A series of conditional statements, e.g. "*if (window['google'] && window['google']['gears'])*", which determine whether a plug-in (e.g. *Google Gears*) is installed or not.

Unfortunately, a browser might have no client-side data source in the worst case. For example, Table 1 illustrates that mobile IE does not support any local storages. To address the problem, the framework offers a simulated data source to imitate a client-side data source at the server-side, as shown in Fig. 5. If a browser has no usable local storage, the simulated data sources will allocate a region for the browser. The region is identified by a unique ID in the browser's cookie or URL parameter. The way that the simulated data source works is similar to the server-side HTTP session. The simulated data source ensures that each browser has at least one available client-side data source.

Through the above steps, our framework can extract several available client-side data sources. However, a further problem is which solution is the most proper for current SORC. At this stage, the most important factors that affect the applicability are performances and size limits. The performances of different local storage differ significantly. Figure 8 illustrates the insertion performances of different local storage solutions in different browsers.

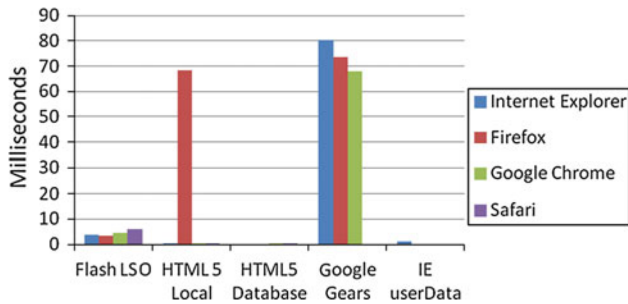

Fig. 8 Insertion performances of different local storages in different browser

Besides the difference of performance, the local data sources also have different storage size limitations, as shown in the Table 2.

Consequently, the framework considers the applicability of the data sources from their CRUD performances and size limitations. The CRUD operations' performance of each data source can be denoted as a vector $< \mathrm{P_c}, \mathrm{P_r}, \mathrm{P_u}, \mathrm{P_d} >$, while the size limitation of each data source can be expressed as $\mathrm{S_{max}}$. Developers can describe the characteristics of their SORCs through two variables: the vector $< \mathrm{W_c}, \mathrm{W_r}, \mathrm{W_u}, \mathrm{W_d} >$ presenting the weighting of each operation in the SORC, and the variable $\mathrm{S_{app-max}}$ expressing the max size of storage. Therefore, the evaluation function of each data source's applicability for a SORC can be denoted as:

$$E = \frac{1}{< \mathrm{P_c}, \mathrm{P_r}, \mathrm{P_u}, \mathrm{P_d} > \times < \mathrm{W_c}, \mathrm{W_r}, \mathrm{W_u}, \mathrm{W_d} >}, \quad if\,(\mathrm{S_{max}} \geq \mathrm{S_{app-max}})$$
$$E = 0, if\,(\mathrm{S_{max}} < \mathrm{S_{app-max}})$$

The most proper data source has maximum E value. The framework allows configuring the characteristic variables of a SORC in three ways:

- $< \mathrm{W_c}, \mathrm{W_r}, \mathrm{W_u}, \mathrm{W_d} >$ is assigned to $< 1, 1, 1, 1 >$ and $\mathrm{S_{app-max}}$ is assigned to infinite by default. This way guarantees that every SORCs can execute without errors;
- Developers can assign the characteristic variables manually. They determine the SORCs are read-intensive or write-intensive. The framework will assign a high value to the former, while a low value to the latter. Then, developers can assign $\mathrm{S_{app-max}}$;
- At last, the framework can assign the characteristic variables adaptively. In this way, the framework selects the best average performance data source at first and keeps a log of every history operations. $< \mathrm{W_c}, \mathrm{W_r}, \mathrm{W_u}, \mathrm{W_d} >$ is assigned to the quantity of history CRUD operations in a time window (e.g. the recent 100 operations). $\mathrm{S_{app-max}}$ is assigned to current size of saved data. The framework will calculate the evaluation functions for each data sources at intervals and migrate to new data source with the maximum E value.

4 Adaptive cache for server-side data

Our framework introduces a cache mechanism to address the cache issues for server-side data access. The cache mechanism supports client-side customization and adaptive cache

Table 2 Size limitations of different local storages

Data source	Flash LSO	IE userData	HTML 5 Local	HTML5 Database	Google Gears
Size limit	100 K	250 K	Depend on implementation.	Depend on implementation.	No limit

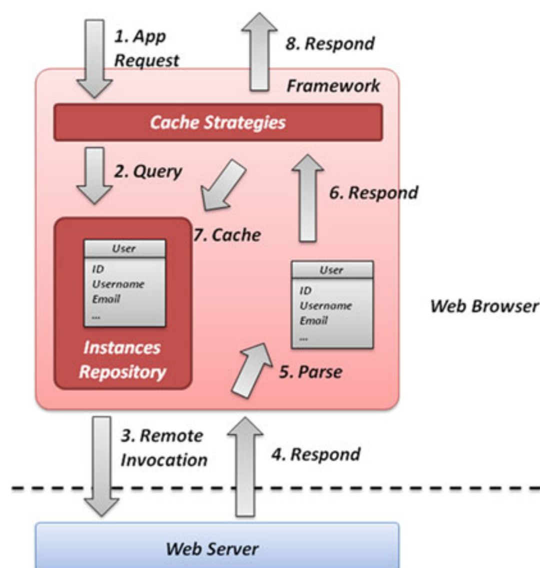

Fig. 9 Caching server-side data

strategies [18]. It copes with the data objects rather than the whole datasets, to further improve cache performance (Fig. 9).

4.1 Data cache scenarios in SORCs

Generally, for all server-side data sources that are delivered via service, the response comprises either one data object or a collection of objects. The collection type of responses could be denoted as $\{\text{object}_i | i = 1, 2, \ldots\}$ or LT (list type), and the object type of the response could be expressed as O (an object) or OT (object type). Based upon the above assumption, when the SORCs request two datasets sequentially, there are five cardinal cache cases where the cache mechanism could provide cached data from the first call to feed the second request:

- Identical. The second request has the exactly same URL and parameters with the first one. It is the classical case of cache;
- $\{\text{object}_i | i = 1, 2, \ldots\} \ni \text{object}_k$. This case comes when the OT response is contained by LT. Assume that the first service returns a list of 10 bestsellers, while the second one returns the most popular book. Then second response is included thoroughly in the cache from the first response;
- $\text{object}_i \ni \{\text{object}_k | k = 1, 2, \ldots\}$. If an OT response is a composite data structure with a list structure inside, then the following LT call might reuse the overlapped data. For instance, the OT is a description for a person. Within the description, there is a list of his/her friends. Therefore, after retrieving the person's description, all requests for his/her friends might benefit from cache;
- $\text{object}_i \ni \text{object}_k$. This is the case when users send OT requests for different integrities of the response. Suppose if the first call is for detailed information of a person with ID and description, while the second call is only to retrieve the person with his/her ID;
- $\{\text{object}_i | i = 1, 2, \ldots\} \ni \{\text{object}_k | k = 1, 2, \ldots\}$. This is the case when users send LT requests in different scopes. Suppose if the first call inquiries a collection of 10 bestsellers from Amazon, and the second call is for the top 5.

4.2 Realization of cache

The response type (LT/OT) and cache cases can be setup in the data model. Developers need to decide the response types, cache cases and primary keys for the model. When a request comes, the cache mechanism looks up its corresponding data model according to the URL pattern specified in the data model. After that, the mechanism needs to determine all related data models searched in the cached data objects. To this end, the response type and cache cases of the model are retrieved. The processing logic for different cases is discussed as follows:

- The first four cases discussed in the previous section would be handled automatically and uniquely: (1) to retrieve the data model of the other party in such relationships; (2) to search all data objects of the existing data models in the cached data;
- The last case (one list contains another) is more complex. All developers have to supply an extra query interface for such a LT model (only for models involved in such a cache case). In that query interface, the programmers should explicitly indicate how to query related models. For instance, if a LT model is to return a list of books, it should embody a query interface which takes the start and end indices as parameters to return the "slice" list. Therefore, after caching a list of "top 100 books", a subsequent inquiry for a list of "top 30 $\sim$ top 50" books could be handled by this inquiry interface with desired results.

When searching all related data models, judgment for a cache hit is realized through primary keys and the parser logics specified in the model. For instance, a request URL like "*proxy://url/q?name=Jeffrey*" is parsed to as an inquiry for a model whose name is Jeffrey. Then, the primary key in the related object is compared with this string. If not matched, or the instance is out of time, the cache mechanism would make the remote call and return to users after the server responds. Otherwise, a hit is found and the matched object is returned.

Validation is performed during the runtime. We implement validation with meta-property in the data model. The meta-property contains the following information:

Table 3 The cache strategy of the meta model

Policy name	Implementation
Time since last access	A timestamp for the last access
Entry time	A timestamp for instantiation
Frequency of access	A number to keep on counting the frequency
Expiration	A timer for validity of the model or data fields

- Data Type: The data type declares data type for each property in the model to facilitate the inquiries. It can be either simple data types (date & time, float, object, string etc.) or complex data types (such as other data models).
- Cache Strategy: The cache strategy is employed to achieve validation and replacement. Currently, we support the following policies in the strategy in Table 3.

In Table 3, the expiration field and entry time determine when a validation is necessary. Time since last access and frequency of access determine the target for replacement when cached data exceeds the size limit.

The cache mechanism sets up the above meta model for each data model. At runtime, after the instantiation of each data model, a time counter is triggered to record the expiration. When it diminishes to zero, the framework would invalidate the corresponding cache. When users request the data after invalidation, the framework would make the request to the target service and refresh the cache.

4.3 Adaptive cache

Users might access the cached data irregularly, and a fixed expiration would impact on the whole system's performance. On the one hand, if the expiration is too short, the hit ratio will be small and the system must send more query requests to the services, which might occupy and waste more resources. On the other hand, if the expiration is too long, users will have to use the out-of-time results, which might make the whole application not reliable.

Moreover, from the experiment, we believe that there are many factors affecting the changes in cache. These factors include both the user behaviors and service behaviors. In terms of the user factors, hit ratio and access frequency would refine the expiration time. In terms of the services factors, the update frequency of services might matter. For example, some of the services change their results by seconds, for example, the stock price fetching or friends-list fetching services, while others just return the same result every time when requested. Under such circumstances, the expiration time should change correspondingly. In this paper, we mainly ignore the service factors because our client-side framework cannot directly make constraints of the service on remote server.

Thus, we propose an improvement of cache strategy. In this strategy, our framework would find the desired expiration time expected by users based on the hit ratio, the access frequency and the data updated rate. The improved cache strategy tries to coordinate the following scenarios:

- Users access the cached data, which returns either a hit or a miss. In this way, the hit ratio is changed.
- Users access the cache either more "quickly" or "slowly", which in turn alters the access frequency.

The algorithm borrows the experience of the simulated annealing algorithm [19], to address the requirements of selecting the most appropriate expiration time for users. In this strategy, we adapt the expiration time dynamically according to the hit ratio and users' behaviors in the history (such as access frequency).

As shown in (Formula 1), the estimated value Vi denotes the expiration time for each cache item Ii. We denote Va as the threshold value, then Vi is calculated as below.

$$V_i = e^{\frac{\alpha \times \text{hit_ratio}}{\text{temperature}}}, \quad \text{if}(V_i > V_a)$$
$$V_i = NA, \quad \text{if}\ (V_i \leq V_a) \qquad (1)$$

The variables in formula (1) are explained as followings.

- Variable hit_ratio denotes the percentage of the hit times from all the visiting of cache item Ii. We enlarge the influence of hit_ratio by multiplying it by a constant α, which is assigned to 1,000 in our current implementation. The higher the hit_ratio's value is, the larger the expiration time value will be. If at most of time, we are able to find what we need in the cache, we just do not need to visit the services any longer. It is reasonable to simply enlarge the expiration time to make the cached data effective for a longer period of time. In contrast, if the hit_ratio is small, it means that the current cached data have little merit for the system to visit again, which has more potential to become out of time. In this case, we should assign shorter expiration time to cut down the data's lifespan. Since a too short expiration time makes the cache little use, we set up a lower bound threshold value and would turn off cache for the expiration that drops below it.
- Variables temperature denotes the sensitivity for the expiration time to change with hit_ratio. In addition, it essentially reflects the current frequency of accessing the services, in some degree representing the users' behavior. The user may use the same query to visit the same service many times in a very short time, causing hit_ratio increase remarkably. The expiration time will soon become very high if it is too sensitive to the hit_ratio. On the contrary, if user does not use the service quite frequently, we cannot get enough experience to adapt the expiration time.

As a result, the sensitivity should be adapted based on the frequency of accessing.

During the annealing process, temperature is adjusted as follows—Let T be the temperature and minT be its lower bound. Let Foa denotes frequency of accessing by users, then:

$$T = \text{Max}\left\{\min T,\ \text{Foa}^2\right\} \tag{2}$$

This formula represents that temperature is decreased when frequency of accessing grows up and increased when it falls down. However, the sensitive degree of changing the expiration time cannot be too small, which will cause a very large number. So we manually define a lower bound T to limit it. The annealing process ends when temperature drops to minT. Foa's increase means that all the services will have more chances to be visited in the period than in other ones. In this case, if the expiration time is seriously influenced by hit_ratio, it might get a very large value like 1 day or fall to a very small value like 0.1 s, both of which are unreasonable for updating services. So we must increase temperature to decline the influence of hit_ratio. On the contrary, if Foa is too small, it might cost the system a long time to collect enough information to adapt the expiration time to the best. In this case, we must decrease temperature to add the influence of hit_ratio. In a word, temperature should be positively affected by frequency of accessing.

5 Experimental evaluation

5.1 Example: online eStore

We implement a simple online eStore as our sample SORC for the experiments. In the eStore, a user can browse products' information (title, description, price, and so on). If the user finds a product interesting, he/she can put it into his/her shopping cart. After adding item into shopping cart, the user can choose to continue shopping or to check out. A new order will be created when the user checks it out.

Figure 10 gives the data model diagram of the SORC. The models of User, Prodcut, Order and OrderItem persist at the server-side. The OrderItems associate an Order with Product in it. The Cart (shopping cart) and CartItem are client-side data models stored in the local storage. The SORC caches User and Product to achieve better performance and user experience.

5.2 Performances analysis of different data sources

First, we measure the CRUD operations' performances of different data sources in different browsers. Each operation manipulates one CartItem data object (24 bytes) and is repeated one thousand times to make the final data distinct. Figures 11, 12, 13 and 14 illustrate the experiments' results. The unit of Y axis is millisecond.

Above results illustrate the characteristics of heterogeneous client-side data sources:

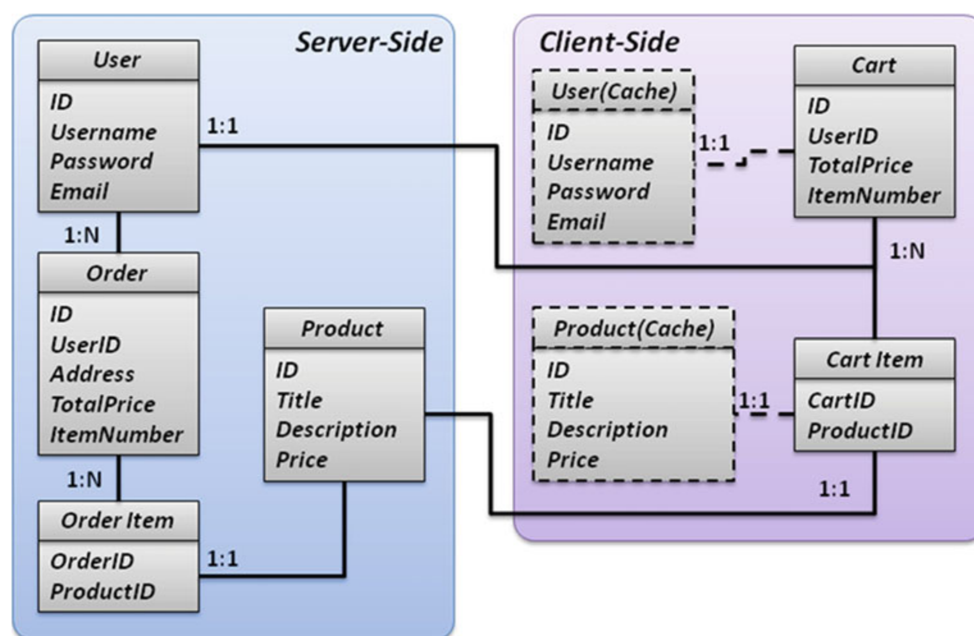

Fig. 10 Data models of online eStore SORC

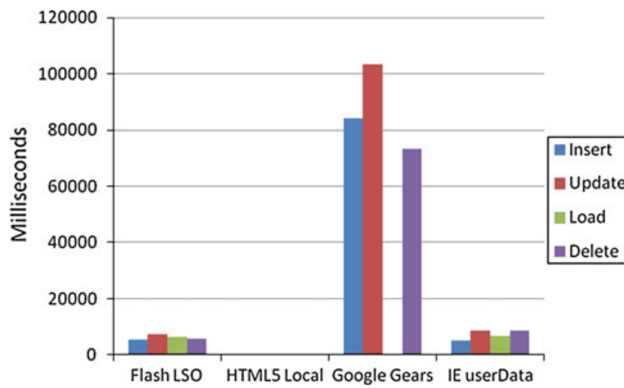

Fig. 11 CRUD performances in IE8

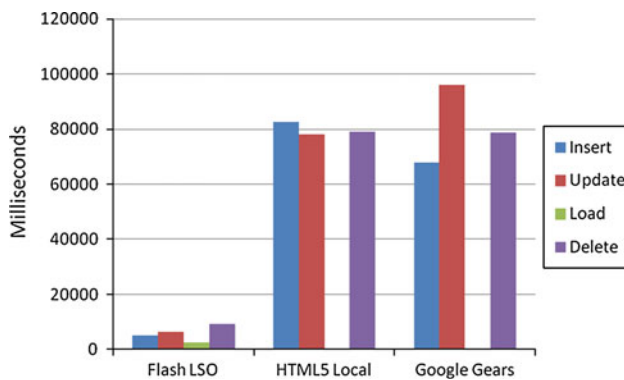

Fig. 12 CRUD performances in Firefox 3.6

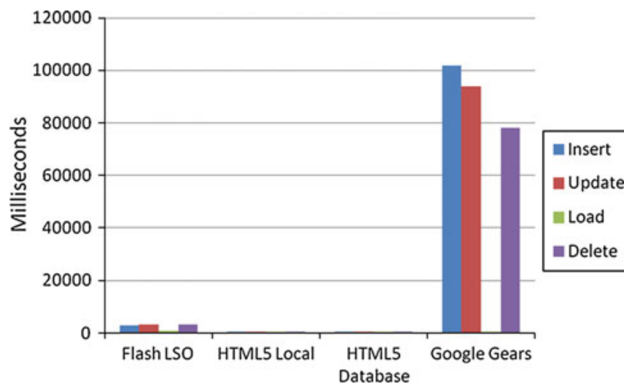

Fig. 13 CRUD performances in Chrome 5

- IE userData data source is the IE's private local storage solution. It is the only available data source in IE 7 or lower. However, the performance of HTML5 data sources is better than userData in IE 8;
- The performance of Flash LSO data source is very similar in different browsers. The possible reason is that Flash LSO is implemented by the unified plug-in and does not relied on browsers' built-in mechanism;
- The performance and size limit of HTML5 data sources differ greatly in different Web browsers, since HTML5 specifications are still in the draft stage and browser

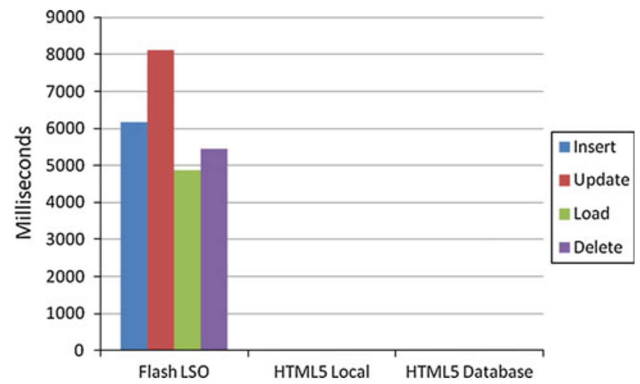

Fig. 14 CRUD performances in Safari 4

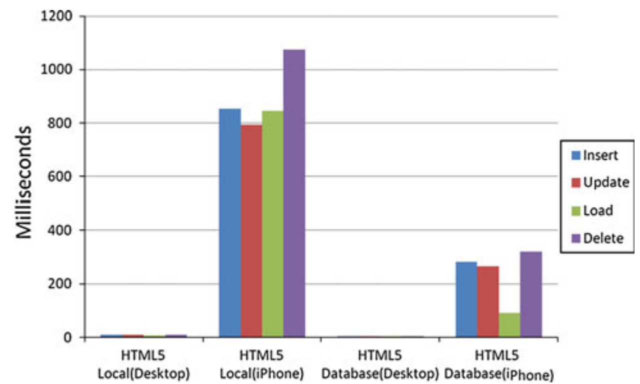

Fig. 15 Performance comparison of HTML5 in Safari desktop and mobile

vendors implement HTML5 data sources according to their own understanding. However, HTML5 data sources have the best average performance in most modern browsers;
- Google Gears data source has fast read and slow write operations. The average performance of Google Gears data source is worst in all client-side data sources. However, it is the only data source without storage size limit. So it is suited to server large-scale data storage requirements.

In the second experiment, we compare the performances of the same local storage solution in desktop and mobile browser. The experiment runs on Safari 4, which has both desktop and iPhone version. In this test, each operation also repeats one thousand times. The result is shown in Fig. 15. Since Flash LSO is not supported by iPhone Safari, the Flash LSO–related data are not displayed in the figure.

In iPhone Safari, HTML5 local storage is about one thousand times slower than its desktop version, while HTML5 database is about two hundred times slower. Since the two versions of Safari use the same browser core, called Webkit,[16] the gap of performances should be mainly due to the

[16] Webkit, http://webkit.org/.

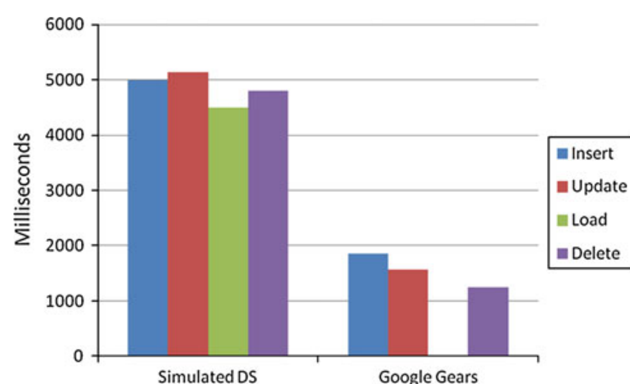

Fig. 16 Performance comparison of local data source and simulated data source

performance difference between PC and mobile phone. However, if compared with other local storages in the desktop browsers, such as Flash LSO and Google Gears, iPhone HTML5 storage has much better performance. Since the latest mobile browsers are mostly based on Webkit, we can draw the conclusion that the local data sources in the latest mobile browsers are usable well.

Finally, we also compare the performance of server-side simulated data source with local data source. Figure 16 displays the comparison results between the simulated data source with Google Gears data source, whose average performance is the slowest.

The performance of simulated data source is much slower than the slowest local data source. Such a big gap is due to that each operation at the simulated data source performs a remote invocation. The parameters marshaling and network round-trip are extremely time-consuming actions. In our experimental environment, the server and the SORC are in the same LAN with low network latency. The performance of simulated data source will be worse in production environment. Accordingly, the simulated data source is only the last resort—when none of the other client-side data sources are available.

5.3 Evaluation of data sources selection

We evaluate the effect of our local storage selection approach though a script, which simulates users' actions and invocation requests. We make the cache store data in the local storage rather than the memory to obtain more local data access. In this case, the eStore firstly writes large amount of data into local storage since the cache is empty. When the hit ratio of cache rises, the read operations will predominate. The demonstration is run in Google Chrome 5 and Firefox 3.6.

Chrome 5 supports four local storage solutions: Flash LSO, HTML 5 Local, HTML 5 Database and Google Gears, while Firefox 3.6 supports: Flash LSO, HTML 5 Local and Google Gears. We use the CRUD operations' performance and size limitation measured in the last section, refer Tables 4, and 5.

Then, we perform the test in the three ways of assigning the characteristics variable of the SORCs, as we mentioned in Sect. 4.2. In the manual way, $S_{app-max}$ is assigned to 100 K due to no need for mass local storage in this test. Both "read-intensive" and "read-write-balancing (RW-balancing)" strategies are put to the test—$< W_c, W_r, W_u, W_d >$ is assigned to $< 0, 1, 0, 0 >$ in the former, while $< 1, 1, 1, 1 >$ in the latter – to find how different manual strategies affect the final result.

Figures 17, 18, 19 and 20 illustrate the evaluation results. The two left figures present the total data access processing time with different selection strategies in the two browsers. The right parts figure the time consumption of every 50 operations to display the change in performance.

The above results reveal the following observations:

- The eStore costs the longest time by default. The default strategy has no idea about the size requirement of

Table 4 Performances and size limits of different local storages in Google Chrome 5

	Read	Insert (ms)	Update (ms)	Delete (ms)	Size limit (K)
Flash LSO	0.70	2.87	3.37	3.30	100
HTML 5 Local	0.09	0.32	0.31	0.38	5,000
HTML 5 Database	0.001	0.002	0.002	0.002	5,000
Google Gears	0.001	101.96	93.80	78.00	No limit

Table 5 Performances and size limits of different local storages in Firefox 3.6

	Read	Insert (ms)	Update (ms)	Delete (ms)	Size limit (K)
Flash LSO	2.45	4.99	6.50	9.15	100
HTML 5 Local	0.10	82.57	78.03	78.89	2500
Google Gears	0.003	67.80	96.14	78.65	No limit

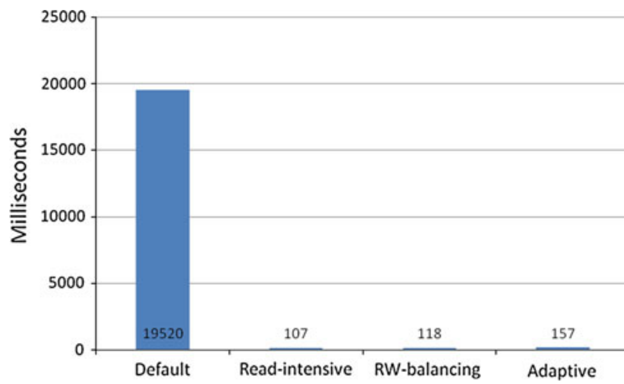

Fig. 17 Time consumption of different strategies in Chrome 5

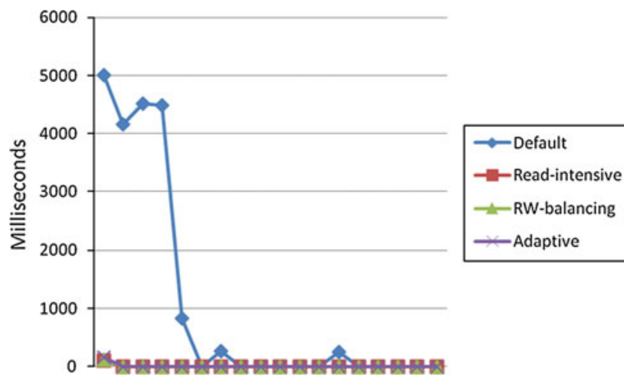

Fig. 18 Time consumption of every 50 operations in Chrome 5

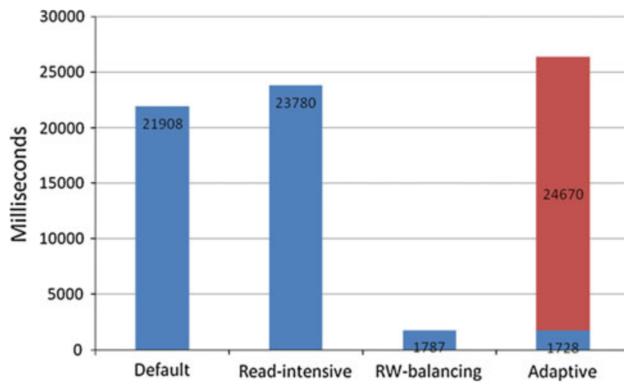

Fig. 19 Time consumption of different strategies in Firefox 3.6.3

application. The framework therefore always tends to select Google Gears data source, which is the only currently known "infinite" local storage, to ensure that data will not cause overflow. However, the write operations in the Gears data source work extremely slowly. As the blue lines show in Figs. 18 and 20, the time consumption of operations is high at first due to frequent write operations, and therefore increases the total processing time.

- With the manual strategies, the result is a bit complex. Counts afterward demonstrated that the ratio between read and write operations in the test is three to one.

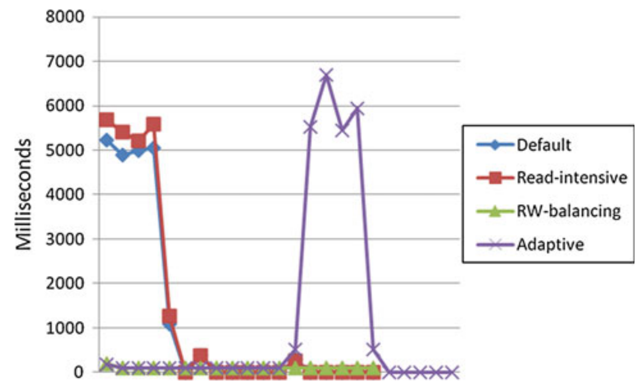

Fig. 20 Time consumption of every 50 operations in Firefox 3.6.3

However, read-intensive strategy does not always select the most suitable data source.

- ■ In Google Chrome, both read-intensive and RW-balancing strategies achieve great performances. It is due to Chrome's HTML5 database executing in an independent background thread, and therefore all CRUD operations run extremely fast. Accordingly, no matter read-intensive or RW-balancing strategy, HTML5 database is selected and then the best performance is achieved.
- ■ In Firefox, however, the read-intensive strategy takes poor performance, similar to the default situation. But the RW-balancing strategy achieves a better result. The reason is that, although Gears data source occupies 2 ms (millisecond) with each read operation, it slow about 70–80 ms with each write (CUD) operation compared with Flash LSO. Therefore, even though the read operations in the test is three times more than write operations, the read-intensive strategy, which selects Gears data source, is much slower than RW-balancing strategy (Flash LSO).

The result above indicates that, due to the exact speed differences among local storage solution, to find the most proper data source, it is better to fine-grained assign $< W_c, W_r, W_u, W_d >$ based on the test results (e.g. $< 0.7, 3, 0.01, 0.29$) rather than coarse-grained "strategy".

- The adaptive selection based on fine-gained sets $< W_c, W_r, W_u, W_d >$ value depend on historical data.
    - ■ In Chrome, HTML5 database is always selected, since HTML5 database always works fastest no matter which kind of operation is in the majority. The total time of adaptive selection is a bit longer than manual strategies (also use HTML5 database). It is probably due to the framework records each data access operation. Each record action spends a very

short time. However, the total time cost of all record actions can be viewed in the final result.

- In Firefox, Flash LSO is selected at first, since it has the best average performance. When cache hit ratio rises, the read operations grow in number. In this situation, Google Gears, which read faster, is more suitable. The right blue bar in Fig. 19 illustrates that the operations' time consumption with the adaptive selection is shorter than RW-balancing strategy, which only uses Flash LSO data source. Unfortunately, the red part of right bar in Fig. 19 shows that the cost of data source migration needs to delete all data in the old source and insert them to the new one. The cost makes the adaptive selection manner be even slower than the default way.

The result using adaptive selection proves that data source migration at runtime is not cost effective. However, the test with manual strategies shows that the fine-grained assigned $< W_c, W_r, W_u, W_d >$ is necessary to find suitable data source. Therefore, there is a more reasonable way for selecting data source. First, the framework records each data access operation. But it does not change to the new data source when an application is running. The data are migrated just before the application closes, and the new data source is available at next use.

The right blue bar in Fig. 21 illustrates that the total processing time of re-run the test after re-selecting data source before the SORC closed. This way neither depends on unreliable coarse-grained manual strategy nor affects the total time consumption and user experience seriously.

5.4 Evaluation of basic data cache

The cache mechanisms evaluation is done by writing a script, which simulates users' actions and invokes requests on behalf of a user. To better imitate the user behavior and to ponder the validation, our script would pause 5 s between every two requests. The demonstration is performed on Firefox 3.6.3.

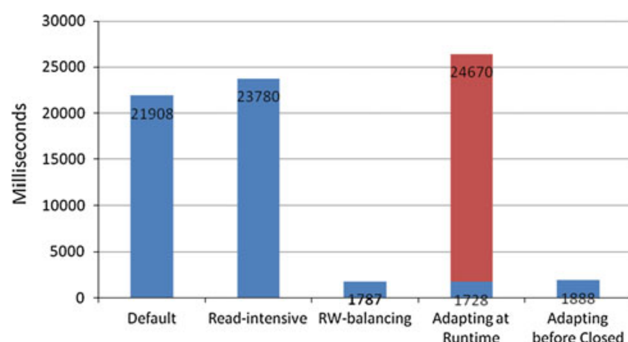

Fig. 21 Time consumption of different strategies in Firefox 3.6.3

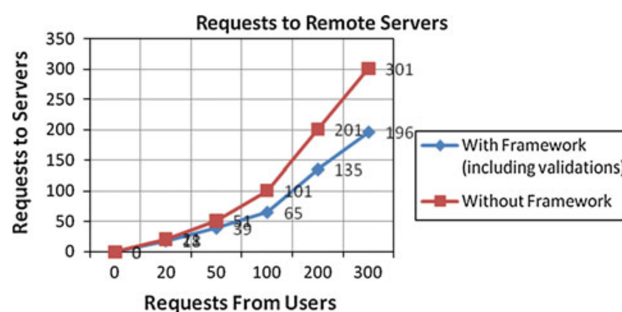

Fig. 22 Requests to remote servers comparison

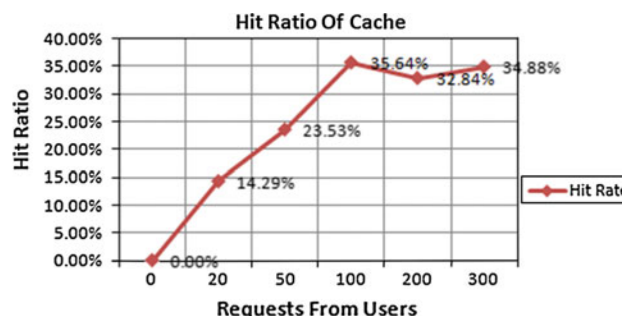

Fig. 23 Hit ratio of cache of the framework

We record the outgoing requests versus the increase in requests with and without the cache mechanism. It is worth noting that the memory consumption of the cache mechanism is dependent upon the data in application context other than the cache mechanism, so it is not included in the evaluation.

The first evaluation is for the efficiency of our cache mechanism. In Fig. 22, the X axis refers to the total number of requests sent by users, and the Y axis refers to the requests that are actually processed by the servers. More requests are replied from caches when the requests rise. The reason is that both the cache and data response reuse the OT models of shopping. The same cases happen in reality when users prefer to quickly browsing all available providers while going back and forth periodically. Moreover, if the strategy conducts the validation less frequently, requests to servers would further decrease.

The second evaluation is to check the performance of the cache manager. In Fig. 23, the X axis takes the same meaning as above. The Y axis refers to the number of hits. The above results reveal the following observations.

- The hit rate starts highly for even small scale of requests. This is due to the concrete behavior of the script and the specific scope of this application. Since the descriptions of the providers seldom change during a long period of time, the cache mechanism incorporates obvious improvement;
- The hit rate rises slightly as the size of requests increases, while sliding down a bit around 32%. The increase in hit

rate comes from the increase in available instances. The decrease is caused by some instances that are out-of-time, which would diminish the hit ratio by missing requests.

From the evaluation above, we demonstrate that the cache manager has a sound performance in exerting the cache strategies from developers, and under appropriate strategies which balance the traffic and validity well, the corresponding composite applications would benefit from the framework without burdening too much on traffic.

5.5 Evaluation of adaptive cache

Besides the above evaluation, we have conducted an experiment on the adaptive feature of the cache. Our purpose is to find out how our algorithms in Sect. 4.3 change the expiration time, so it would adapt to users' behavior.

We let the SORC request only one dataset of the server-side data source. There is another script requesting the dataset on behalf of the users with irregular access frequencies and different parameters (from a parameter collection that is large enough). We have timed the process and recorded the following factors: frequency of access (Foa), hit ratio (hit times divided by total times), temperature and Vi. The records have the unit of 10 times, and the time interval is 2 min (Figs. 24, 25).

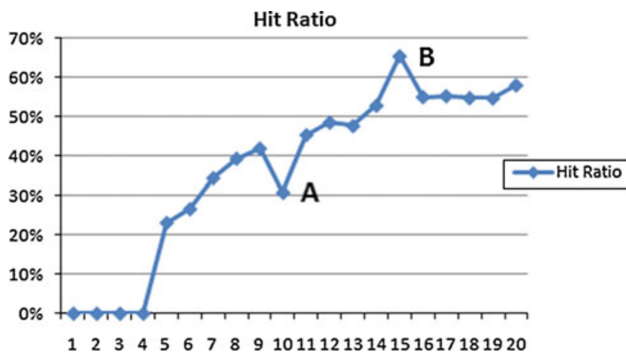

Fig. 24 Hit ratio against time without adaptive cache

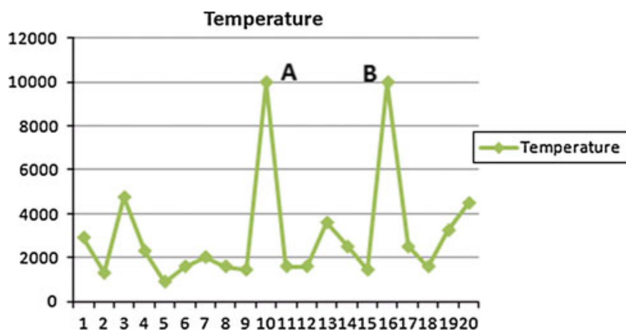

Fig. 25 Temperature against time

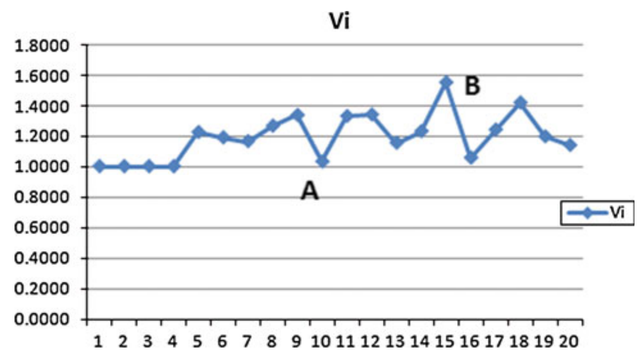

Fig. 26 Vi against time with adaptive cache

The first step of the experiment is done without the adaptive cache. We generate a series of requests (denoted as R0). These requests are sent irregularly, which means the frequency of invocations would change in some period of time.

From Fig. 26, we could find out that there is a dramatic decrease in hit ratio at Point A and also an increase at Point B. The reason is around the period of A, users send too many requests with different parameters, which lead to significant cache miss. While in the period of Point B, users just send identical requests which increase the hit ratio. As discussed previously, our approach should alleviate the dramatic expiration change at both Point A and B. We should also allow expiration to change much more quickly around other time.

Then, we apply the adaptive method, send the exactly same requests R0 in the same order and frequencies and obtain the following data about temperature.

According to Formula 2, temperature is affected by Foa. From the graph, we can observe that around the period of A and B, temperature rises significantly due to the large quantity of requests. According to Formula 1, since high-frequency results in high temperature, which "slows down" the process of "annealing", we expect the Vi to drop smoothly at the Point A and increase smoothly at Point B. At other points, temperatures are rather low so Vi should change quickly.

Finally, we check the resulting expiration time (which has a linear conversion from Vi) from the impact of temperature.

From Fig. 26 and Formula 2, we are generally more concerned over how Vi performs at A and B among all the dots. The relatively small decline at A shows a desirable result, which means our approach controls the expiration time to be relatively stable when too many cache misses occur. In this way, if we set up the threshold value of Va as level of A, the framework could turn off the cache to reduce the unnecessary memory occupation. The same effect could be observed on the smooth increase at Point B. On the other hand, the fluctuation on the left segment of the curve shows our annealing algorithm changes the expiration time very quickly at a lower temperature. This makes sense in the sense that the

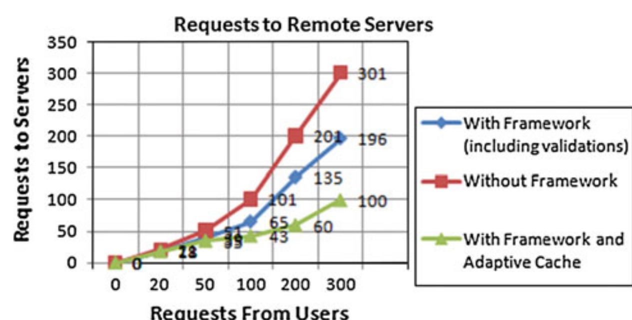

Fig. 27 Requests to remote servers comparison (2)

framework would find the desired expiration time quickly without too much history data.

Consequently, we run the beginning eStore test script again with the adaptive cache on. The comparison of the requests is as shown in Fig. 27. From the experiment above, we show that the adaptive cache and the annealing algorithm suit some scenarios rather well. It guarantees the stable changes in expiration time, and encourages quick adaption at the inception of scenarios.

6 Discussion

In practice, there are many mature and powerful ORM functions, such as inheritance mapping, object-oriented query language (e.g. HQL) and so on. However, the SORCs are still in early stage and do not have so powerful data models yet. Our framework does not support these ORM functions since they are yet not necessary for current SORCs. On the other hand, since our framework should take non-SQL local storages into consideration, it is hard or even impossible to realize some of the powerful functions. The advanced topics therefore are beyond the scope of this paper. So far, our framework only adopts the simple ORM strategies. In our future work, we will try to introduce more powerful ORM functions and investigate how the functions affect the SORCs' data access.

Currently, our local data sources selection approach only considers the performance of CRUD operations. However, the aggregate operations also have great influence on the performance of local storages and further greatly affect the data sources selection. For example, the SQL solutions, such as Google Gears, support multi-conditions SQL queries. In the non-SQL local storages, such inquiries have to be implemented by consulting the whole table, which occupies more time cost. Accordingly, considering the performance of the aggregate operations, the SQL local storages can be more feasible for some SORCs and some browsers.

Our cache mechanism is implemented on the client-side. The use of our cache mechanism might be restricted for server-side applications, since many developers compose services on the server-side and provide the final Web pages to users. To enable caching for these applications, the cache mechanism should accommodate the server-side implementation. Since the mechanism and the runtime environment are loosely coupled in our framework, it might not be a tough task to realize such an environment.

There are several other important data access issues to a data access framework, such as transaction, concurrency control and so on. However, the client-side data sources neither suffer from these issues nor support particular advanced features to deal with them. For example, no local storages support transaction until now, and they do not need concurrency control since JavaScript always runs in a single thread in a SORC. At the server-side, we find that most of the issues are not quite different and can be partly handled by existing server-side data access frameworks. Therefore, within this paper, we do not discuss the problems. Such issues may be more considered in our future work.

There are several types of SORCs, such as AJAX-based SORCs, Flash-based SORCs, Silverlight-based SORCs and so on. Our data access framework mainly targets at the AJAX-based SORCs, since it is the most common and popular fashion. However, the different types of SORCs address the same requirements and suffer from the same data access issues as well. Furthermore, all types share the same application structure or pattern. They are all implemented by the structural markup language (AJAX-HTML, Flash-MXML, Silverlight-XAML), the variant of ECMAScript (AJAX-JavaScript, Flash-ActionScript, Silverlight-JavaScript) and the CSS (Cascading Style Sheets). Accordingly, the functions that can be realized are similar in the different types. Therefore, we note that most of the mechanisms proposed in this paper can be also extended for other types.

7 Related work

There have been some Web engineering research efforts [9, 10] paying attention to SORC data modeling. They provide a set of guidelines to help developers determine storage locations (server-side or client-side) under specific contexts. Some of the works also offer client-side data modeling approaches. The result data models can be transformed into a part of final SORC automatically. However, most of current works do not specify how the SORCs to access local storages, and there is no sign that they concern the issues of heterogeneous local storages and unsuitable HTTP cache mechanism.

A number of AJAX-based SORC development frameworks have emerged, such as ActiveJS,[17] JavaScriptMVC[18]

[17] ActiveJS, http://activerecordjs.org/.

[18] JavaScriptMVC, http://javascriptmvc.com/.

and SproutCore.[19] These frameworks all provide "Model-View-Controller templates" to developers. With the templates, developers can build SORCs more easily and quickly. The model part of the frameworks is responsible of connecting the SORCs with the data sources. However, until now, these development frameworks mainly focus on how to rapidly generate data models based on existing Web-delivered services, especially RESTful Web-delivered services [20]. As a result, the MVC frameworks offer very few facilities in addressing the issues mentioned in this paper. However, since the frameworks adopt diverse data access patterns, such as DAO (Data Access Object) and ActiveRecord, they provide some valuable lessons in the patterns selection and implementation of our framework.

The concept of semantic cache is proposed to reuse the overlapped cache data with the help of the semantic knowledge of the data [21]. The semantic cache focuses on how to exploit the semantic information within the query and how to use this information for the future reuse [22]. We believe that semantic cache is a very desirable reference for our cache mechanism. Because data models with semantics are similar with result sets in semantic cache, and the algorithms for classifying, reusing and replacing [23] could supplement our relatively simple models. However, it is also worth noting that semantics in datasets from Web-delivered services are more variable and complex than the query-based responses (usually expressed as name-value pair) of semantic cache. Thus, our framework differs from traditional semantic cache technologies.

Depending on the hierarchy differences, there are two kinds of browser-side cache. The first one is to cache on the data tier, such as the browsers' built-in cache or cache modules in the browser add-ins (e.g. Runtime Shared Libraries in Flash[20]). This sort of cache would map URLs to content according to HTTP/1.1. A number of elements in the HTTP HEADER are included to direct cache behaviors, which are specified by service providers. The second kind resides on the application tier, which are usually embedded in client-side JavaScript frameworks [24]. Nevertheless, popular JavaScript frameworks such as Dojo[21] and DWR[22]only cache the data on basis of the entire Web pages or responses, providing inadequate flexibility for developers.

Many research works [12,13,25] focus on data access between the application servers and the databases at the server-side. There are also some server-side data access frameworks, such as Hibernate, RoR ActiveRecord. These works inspire us to provide a data access framework for SORCs. We also borrow lots of experiences from the works. However, the SORCs have their own data access issues, such as heterogonous data access clients, which have not considered in the server-side data access frameworks. The special issues are the uppermost concern in our work.

[19] SproutCore, http://www.sproutcore.com/.

[20] RSL, http://www.adobe.com/devnet/flex/articles/rsl.html.

[21] Dojo, http://dojotoolkits.org.

[22] DWR, http://directwebremoting.org/dwr/index.html.

8 Conclusion

Service-Oriented Rich Clients combine the benefits of the Web distribution thin clients model with the highly interactive desktop clients. The SORCs provide the advantages in terms of client-side data storage and manipulation. The SORCs suffer from data access issues. In this paper, we propose a data access framework for SORCs. This paper makes the following contributions: (1) an adapter for uniformly accessing heterogeneous local storages and a local storage selection technique; (2) a client-side cache customization mechanism and an adaptive technique for cache strategies.

As we discussed in the Sect. 6, there are still some open issues for the SORC data access framework. We will try to support relationships among data models and provide more powerful query languages in the future work.

Acknowledgments This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60821003, 60873060, 61003010; and the Program for New Century Excellent Talents in University.

References

1. Al-Masri E, Mahmoud QH (2008) Investigating web-delivered services on the world wide web. In: Proceedings of 28th WWW conference. Beijing, China, pp 1–8
2. Driver M, Valdes R, Phifer G (2005) Rich internet applications are the next evolution of the web. Technical report, Gartner
3. Zhao Q, Liu X, Huang J, Huang G (2010) A browser-based middleware for service-oriented rich client. In: Proceedings of 2010 international conference on service science (ICSS). Hangzhou, China, pp 22–27
4. Duhl J (2003) White paper: rich internet applications. Technical report, IDC
5. Meliá S, Gómez J, Pérez S, Díaz O (2008) A model-driven development for GWT-based rich internet applications with OOH4RIA. ICWE
6. Teng T (2007) Research on pragmatics-based persistence. Doctoral dissertation, Peking University
7. Haas LM, Miller RJ, Niswonger HM, Tork M, Schwarz RPM, Wimmers EL (1999) Transforming heterogeneous data with database middleware: beyond integration. IEEE Data Eng Bull 22(1):32–38
8. Atkinson M, Morrison R (1995) Orthogonally persistent object systems. VLDB J 4(3):319–401
9. Bozzon A, Comai S, Fraternali P, Carughi GT (2006) Conceptual modeling and code generation for rich internet applications. In: Proceedings of 2006 international conference on web engineering (ICWE). Palo Alto, California, USA, pp 353–360

10. Preciadol JC, Linajel M, Comai S, Sanchez-Figueroal F (2007) Designing rich internet applications with web engineering methodologies. In: Symposium of 9th web site evolution (WSE). Paris, France, pp 23–30
11. Zani GP (1992) Expert database systems: state of the art. Tutorial Documents of the First World Congress in Expert Systems, USA
12. Klettke M, Meyer H (2001) XML and object-relational database systems: enhancing structural mappings based on statistics. WebDB 2000, LNCS 1997
13. Ambler SW (2006) Mapping objects to relational databases: O/R mapping in detail. http://www.agiledata.org/essays/mappingObjects.html
14. Huang G, Zhao Q, Huang J, Liu X (2009) Towards service composition middleware embedded in web browser. International conference on cyber-enabled distributed computing and knowledge discovery
15. Fielding RT (2000) Architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California, Irvine
16. Mashups MD (2006) The new breed of Web app. http://ibm.com/developerworks/library/x-mashups.html
17. Ruby S, Thomas D, Hansson DH (2009) Agile web development with rails, 3rd edn. Pragmatic Bookshelf, USA
18. Rabinovich M, Xiao Z, Douglis F, Kalmanek C (2003) Moving edge side includes to the real edge: the clients. In: Proceedings of the 4th USENIX symposium on internet technologies and systems
19. Russsel SJ, Norvig P (1995) Artificial intelligence: a modern approach. Prentice-Hall, Englewood Cliffs, NJ
20. Richardson L, Ruby S (2007) RESTful web-delivered services. O'Reilly
21. Dar S, Franklin MJ et al (1996) Semantic data caching and replacement. In: Proceedings of 22nd VLDB conference. Bombay, India, pp 330–341
22. Chidlovskii B et al (1999) Semantic cache mechanism for heterogeneous web querying. Int J Comput Telecommun Netw 31: 1347–1360
23. Arens Y, Knoblock CA (1994) Intelligent caching: selecting, representing, and reusing data in an information server. In: Proceedings CIKM'94 conference. Gaithersburg, MA, pp 433–438
24. Barish G, Obraczka K (2000) World wide web caching: trends and techniques. IEEE Commun Mag Internet Technol Ser 38(5):178–185
25. Kounev S, Buchmann A (2002) Improving data access of J2EE applications by exploiting asynchronous messaging and caching services. In: Proceedings of 28th VLDB conference, pp 574–585