Towards a Data Access Framework for Service-Oriented Rich Clients

Qi Zhao[1] XuanzheLiu[1]* XingrunChen[1] Jiyu Huang[1] Teng Teng[2] Yong Zhang[2]

[1]Key Laboratory of High Confidence Software Technologies, Ministry of Education, China
[1]School of Electronics Engineering and Computer Science, Peking University, Beijing, China
[2]Kingdee Middleware Company Ltd., Shenzhen, China

zhaoqi06@sei.pku.edu.cn liuxzh@sei.pku.edu.cn chenxr07@sei.pku.edu.cn huangjy07@sei.pku.edu.cn tengteng@apusic.com zhangyong@apusic.com

***Abstract*—Along with the proliferation of Web-delivered services and the wide adoption of popular Web technologies, it has been an emerging development style that composes services into Rich Internet Applications in the client-side runtimes, i.e. web browser. These service-oriented rich clients (SoRC) have to access and process client-side data for rich user experiences. Though server-side data access has become simple and effective facilitated by frameworks, e.g. Hibernate, client-side data access is yet challenging due to the heterogeneity and non-determinism derived from the local storage solutions (such as Flash LSO, HTML 5, etc.) and web browsers. In this paper, we present a web browser-based data access framework for service-oriented rich clients. The main efforts of this paper include: 1) an adapter for shielding heterogeneous data sources and a set of unified APIs for hiding incompatible access APIs provided by different local storage solutions; 2) a performance analysis of heterogeneous local storage solutions and an algorithm for selecting the most suitable local storage for current RIAs and browser; 3) a series of experiment evaluations for the feasibility and effectiveness of the framework.**

Keywords-data access; RIA;serivce-oriented rich client

I. INTRODUCTION

In the last years, browser-server architecture has evolved a lot. At server-side, Service-Oriented Architecture (SOA) has been widely adopted, and data and functionalities can be accessed in terms of Web services (SOAP and RESTful Web services, RSS/Atom feeds). At client-side, web browser is capable of providing rich user experiences facilitated by Web technologies (AJAX, Flash, etc). Therefore developers are now composing various services to create a plethora of Service-Oriented Rich Internet Applications (SoRIA) [1]. SoRIAs can well deal with several drawbacks of traditional web applications, such as poor user experience, unnecessary round-trip server access, and so on [6].

A major distinguished feature of SoRIAs is the capability to store and process data directly at SoRIAs' client-side, i.e. Service-Oriented Rich Client (SoRC) [1]. The client-side storage and computing capacities make SoRIAs capable of providing richer and more interactive user interfaces. It also significantly reduces network traffic using more intelligent asynchronous requests that send only small blocks of data [9]. Therefore, with the increasing of SoRIAs, the requirements of local storage increased rapidly. [7][8] indicate that SoRIAs had better to or even have to retrieve or persist data at client-side (i.e. local) in several cases. For example, if an entity represents content owned by the individual user or primarily originated and manipulated at client-side, client-side persistence might be achieve better performance since it gets rid of those unnecessary remote round-trips. Furthermore, if SoRIAs work under the offline mode, all data have to be retrieved locally, in which case remote data access will be failed and then cause bad user experience. To meet the demand, a number of local storage solutions appear, e.g. Flash Local Shared Objects (LSO)[1], HTML 5[2], Google Gears[3]. These solutions provide client-side data sources to make the SoRC capable of storing data at local. Until now, all popular web browsers, as "SoRC platforms", have local storage capacity more or less. For example, the old versions of browsers, such as IE 6, have such capacity via plug-ins; while the latest versions of the browsers, e.g. Safari 4, support local storage natively.

However, the local storage capacity still does not widely used in current SoRIAs. The fundamental reason is data access issues, which are mainly due to heterogeneity and non-determinism derived from the local storage solutions and the browsers: The heterogeneity is casused for the different solutions are incompatible and each solution is only supported by part of browsers (see TABLE I.), The nondeterminism means that a SoRIA cannot know which web browsers their users used and further which local storage solutions are supported until actual users arrive. As a result, to use local storage capacity, a SoRIA should be able to access several heterogeneous data sources in different web browsers. Unfortunately, it is not easy to produce optimal data access solutions for multiple web browsers. In fact, current SoRIAs are always based on one specific local data source. For example, Gmail used Google Gears as its local storage solution to implement its offline feature at first. Until recently, however, it changed to HTML5 database solution. The two solutions are supported by very different web browsers (Google Gears: IE 6+, Firefox 1.5+, Chrome;

* Corresponding Author.

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

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

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

HTML5 database: IE 8+, Chrome 3+, Safari 3.1+). Therefore, the offline feature always cannot serve all users. Nevertheless, even so, Gmail does not maintain the Gears and HTML5 solutions both. This proves the difficulty of realizing a data access solution supporting multiple local storages.

At server-side, web application vendors also suffer from data access issues when their application servers (e.g. Weblogic or JBoss) accessing heterogeneous databases (e.g. Oracle and MySQL). In this situation, the vendors turn to server-side data access frameworks (e.g. Hibernate[4]) rather than resolve the issues manually. Data access framework takes charge of connecting applications and data sources [4] and makes data access become simple and efficient [2][3]. Accordingly, to address the local data access issues for SoRIAs mentioned above, we believe that a data access framework for SoRC is required.

In this paper, we therefore propose a data access framework, which connects the client-side data model with data sources, resolves the data access issues for SoRC, and assists vendors to build SoRIAs with local storage capacity. Currently, there are several methods of rich clients, such as JavaScript rich clients in web browser, ActionScript rich clients in Flash player and so on. Our framework only targets at the JavaScript rich clients in browser currently, since it is the most widely used. However, the approach within this paper can apply to other types of rich clients.

The main capabilities of our framework include: 1) providing a unified means to store data objects and shielding the heterogeneous client-side data sources and incompatible local storage solution APIs; 2) selecting the proper local storage solution based on the characteristics of current browser and SoRIA in use. The evaluation results demonstrate the effects of this framework. Base on it, the SoRCs can work with diverse client-side data sources properly and effectively.

The rest of the paper is organized as follows. Section 2 discusses related work. In Section 3, we specify the data access issues in the SoRCs. Section 4 presents our framework and especially explains the solutions to resolve the issues mentioned above. Section 5 gives the evaluation. Finally, we give some discussions in Section 6 and conclude this paper in Section 7.

II. RELATED WORK

Some RIA Web engineering research works [7][8] pay attention to client-side data modeling. These works provide a set of guidelines to assist developers to determine store location (server-side or client-side) of specific content. They also offer modeling approaches to describe client-side data model. The models can be transformed into final RIA automatically. The works indicate the requirements and feasibility of local storages in the rich clients. However, the works do not specify how their rich clients to access local storages. And there is no sign that they concern the heterogeneous issues of local storage solutions and browsers.

Since the SoRC begin to adapot MVC pattern, a number of rich client MVC frameworks have arisen, such as JavascriptMVC[5] and SproutCore[6]. The frameworks provide MVC "templates" to developers. With the templates, developers can build MVC rich clients more easily. Since data models in rich clients generated by the frameworks also need connect with data sources, the model part of the frameworks pays attention to data access more or less. However, until now, these works focus on accessing server-side data sources rather than client-side data sources.

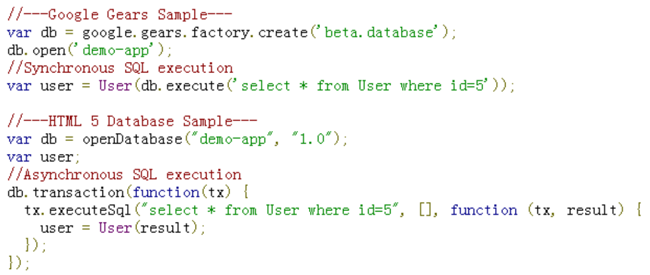
Many research works focus on server-side data access [11]. There are also plenty of mature server-side data access frameworks, such as Hibernate. These works inspire us to provide a data access framework for rich clients. We also borrow lots of ideas from the works. However, the rich client data access has its own issues, such as nondeterminism derived from web browsers, which are not suffered from by server-side data access framework. These special issues are the uppermost concern in our work.

III. CLIENT-SIDE DATA ACCESS ISSUES FOR SERVICE-ORIENTED RICH CLIENTS

A. *The Issues due to Heterogeneous Storage Solutions*

As previously mentioned, with the requirements of local storage rapidly increasing, diverse local storage solutions have been arisen, such as Flash LSO, Google Gears and HTML5, which is the next version of HTML. However, the diverse local storages are heterogeneous.

Currently, the local storage solutions mainly include two sorts. The first type of solutions embeds lightweight relational databases (e.g. Sqlite) into browsers. They provide SQL query APIs for the SoRCs. The second type of solutions also provides table-based data sources. However, their data sources offer one table for each SoRC and each table only has two columns (a key and a value). The table supports simple CRUD functions rather than powerful query language. Comparing the two, the former has more powerful capabilities, while the latter is easier to be used for simple requirements. Accordingly, the two types are both necessary for SoRCs. Customarily, the first type of solutions is referred to as "SQL Local Storage", and the second is called "non-SQL Local Storage". The two types are not completely compatible obviously and the heterogeneous issue should be addressed.



```
//---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);
  });
});
```

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

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

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

Figure 1 Incompatible APIs for Different SQL Local Storage Solutions

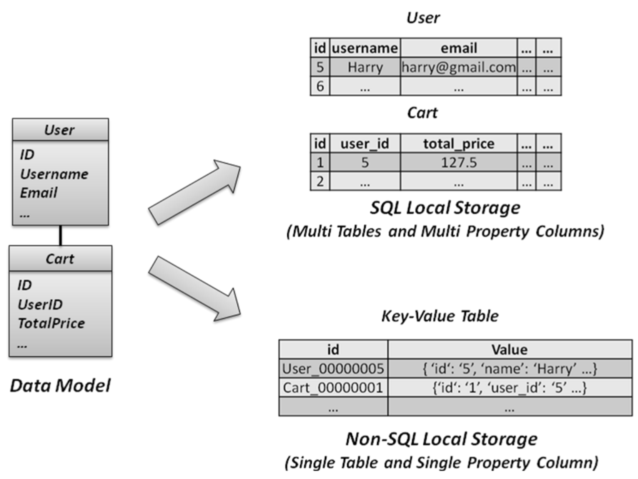

User
ID
Username
Email
...
Cart
ID
UserID
TotalPrice
...
Data Model
User
id | username | email | ... | ...
5 | Harry | harry@gmail.com | ... | ...
6 | ... | ... | ... | ...
Cart
id | user_id | total_price | ... | ...
1 | 5 | 127.5 | ... | ...
2 | ... | ... | ... | ...
SQL Local Storage
(Multi Tables and Multi Property Columns)
Key-Value Table
id | Value
User_00000005 | { 'id': '5', 'name': 'Harry' ...}
Cart_00000001 | {'id': '1', 'user_id': '5' ...}
... | ...
Non-SQL Local Storage
(Single Table and Single Property Column)

Figure 2 Heterogeneous Local Storages

Compounding this problem, each type of solutions has several different realizations. The SQL local storage is supported by Google Gears and HTML5 database. IE userData, Flash LSO, "HTML5 local storage" solutions provide non-SQL client-side data sources. Each implementation has its own characteristics and APIs. For example, in Google Gears, the SQL queries execute in synchronous model. While HTML 5 database has an individual background thread dealing with every SQL query asynchronously. As a result, the data result set handler should be registered as a callback functions in HTML5 database solution. The APIs of the two solutions are quite different thus, as shown in Figure 1. When rich clients access the diverse local data sources, these incompatible APIs are also an important issue that has to be resolved.

Furthermore, current data models in SoRCs are always object-oriented. Therefore, there are classic "impedance mismatch" issues [10] between data objects and the different client-side data sources, as shown in Figure 2.

B. The Issues due to Diverse Web Browsers

In practice, the issues for accessing client-side data sources are even more complex, since the possible web browsers, as the platforms for SoRCs, are wide varieties and each supports different local storage solutions. TABLE I. illustrates some major web browsers and the local storage solutions they support.

TABLE I. 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

A local storage solution cannot be used in the browsers that do not support them. Therefore, if a SoRC uses specific local storage solution (e.g. Google Gears), it may not be able to serve some browsers (e.g. Safari). Unfortunately, in Internet environment, the SoRCs have no idea about what browser may visit before the SoRIAs are deployed and actual users arrive. As a result, selecting an available local storage for current browser in use is a great issue.

Furthermore, one browser may support several available local data sources, which have different performances. Therefore, the goal of local storages selection may be not only to find available local storages but also try to determine which one is the most suitable in current situation.

IV. DATA ACCESS FRAMEWORK FOR RICH CLIENTS

There are heterogeneous local storage solutions. On the other hand, different web browsers support different solutions. Therefore, the data access framework for SoRCs has two main functions: 1) to adapt heterogeneous client-side data sources and provide a set of unified APIs to store data objects; 2) to select a rational local storage solution for current SoRIA and web browser in use.

In this section, we illustrate the detail of our data access framework for rich clients and explain how it addresses the issues in above section. As mentioned previously, our framework target at the JavaScript rich clients in browser. Therefore, the framework is implemented by JavaScript and hosted in web browsers. Developers need import the frameworks' JavaScript files into their project. Then the framework will be initialized when SoRIAs loaded into web browsers.

A. Adapting Heterogeneous Data Sources

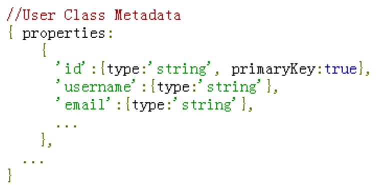
The impedance mismatch is a question which most every data access framework faces. Our data access framework also suffers from the model mismatch between object-oriented data model and local data sources.

Currently, there are mainly two patterns, ActiveRecord[7] and DAO (Data Access Object)[8] for data access framework to address the mismatch issues between object-oriented applications and data sources. ActiveRecord pattern makes developers be able to use persistent functions in more intuitive and convenient way [12]. However, since the pattern requires weaving persistent methods into data objects dynamically, its realization depends on reflection mechanism and it is hard to be implemented by compiled languages. Our data access framework is realized by JavaScript, which is a dynamic, weak-typing and interpreted language. Accordingly, the framework adopts ActiveRecord pattern.

To make the framework know which properties in an ActiveRecord should be persisted, developers need define the metadata on data model, as shown in Figure 3. The metadata includes the name and type of properties. And then,

[7] ActiveRecord Pattern, http://en.wikipedia.org/wiki/Active_record_patter
[8] DAO Pattern, http://en.wikipedia.org/wiki/Data_access_object

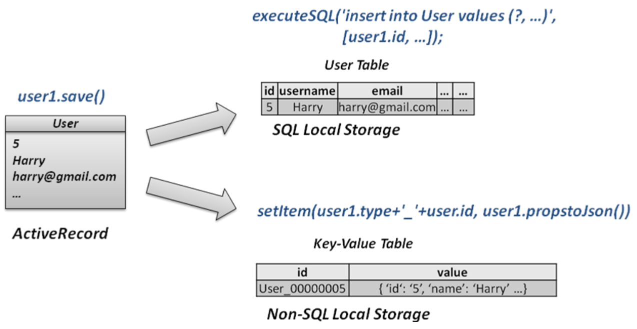
when a SoRC loaded, the framework will read all metadata definitions and weave a series of persistent methods into data classes and objects at runtime, such as *User.find*, *user.save*, *user.update* and so on.



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

Figure 3   Metadata of Data Model

The ActiveRecords can be mapped to the structures of heterogeneous data sources. As shown in Figure 4, a data object is persisted as a row of a specific table in SQL database or a JSON[9] string indexed by type and id in the non-SQL key-value table. If a persistent method (e.g. save) is invoked, the invocation will be translated into specific operations on current used data source (e.g. SQL "insert" on SQL database or setItem method on key-value table) and sent to related adapter.



```
executeSQL('insert into User values (?, ...)',
  [user1.id, ...]);
user1.save()
setItem(user1.type+'_'+user.id, user1.propstoJson())
```

Figure 4   Mapping Objects to Different Data Sources

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

B. Selecting Suitable Data Sources

Although the framework makes the SoRCs be able to store data objects into different data sources via a unified way, a crucial issue still remains to be resolved as mentioned above – how to determine which local storages is most suitable for a specific situation. The decision-making depends on two aspects.

The first influencing factor is current web browser in use. As previously mentioned, the different browsers support different solutions. The available client-side data sources therefore depend on the current browser in use. Since nobody can prejudge what browser may be used, the data source selection has to be made at runtime. The local data source adapter finds the available data sources via: 1) the "user-agent" field, e.g. "*Mozilla/5.0 Gecko/20100401 Firefox/3.6.3*", which marks each browser's type and version (e.g. *Firefox 3.6.3*), and; 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, there will be no available client-side data source in the worst case. For example, TABLE I. illustrates mobile IE does not support any local storage solution. In order to address this situation, the data access framework offers a simulated data source to imitate a client-side data source in the server-side. When a browser without usable local storage arrives, the simulated data sources will allocate a region for the browser. The region is identified by a unique id saved in the browser's cookie or URL parameter. The way of simulated data source working is similar with the server-side HTTP session. The simulated data source ensures that each browser has at least one available "client-side" data source, even if the browser does not support any local storage solution.

Through the above step, several available client-side data sources have been picked out. However, a further problem is which storage solution is the most suitable for current SoRIA. At this stage, the most important factors that affect the applicability of data sources are performances and size limits. The performances of different local storage differ significantly. Figure 5 illustrates CRUD performances of different local storage solutions in different web browsers.

Besides the difference of performance, the local data sources also have different storage size limitations, refer to the following table.

TABLE II.   SIZE LIMITATIONS OF DIFFERENT LOCAL STORAGES

Data Source	**Flash LSO**	**IE userData**	**HTML 5 Local**	**HTML5 Database**	**Google Gears**
Size Limit	100K	250K	Depend on Impl.	Depend on Impl.	No Limit

Accordingly, we consider the applicability of data sources from their CRUD performances and size limitations. The CRUD operations' performance of each data source could be denoted as a vector $< P_c, P_r, P_u, P_d >$, while the size limitation of each data source could be expressed as S_{max}. And then we describe the characteristics of a SoRIA through two variables: a vector $< W_c, W_r, W_u, W_d >$ presenting the weighting of each operation in the rich clients, and a variable $S_{app-max}$ expressing the rich clients required max size of storage. Therefore, the evaluation function of each data source's applicability for a SoRIA can be denoted as:

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

[9] JSON, http://www.json.org

The most suitable data source has maximum E value. The data access framework supports setting the characteristic variables of a SoRIA in three ways:

- There is no especial characteristic description by default. Therefore, $< W_c, W_r, W_u, W_d >$ are assigned to <1,1,1,1> and $S_{app-max}$ is assigned to infinite. This way guarantees that every rich clients can execute without errors;
- The RIAs vendors can assign the characteristic variables manually. First, they can determine that their rich clients are read-intensive or write-intensive. In the former a high value, while in the latter a low value, will be given to W_r. And then the vendors can also assign $S_{app-max}$;
- At last, the local data source adapter can select 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. $< W_c, W_r, W_u, W_d >$ is assigned to the quantity of history CRUD operations in a time window (e.g. the recent 100 operations). $S_{app-max}$ 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 if necessary.

V. EVALUATION

We implement a simple online e-Store SoRIA as our sample application. In this SoRIA, a user can browse products' information (title, description, price …) in his/her browsers. If the user finds a product interesting, he/she can put it into his/her shopping cart. After added item into shopping cart, he/she can choose to continue shopping or proceed to check out. A new order will be created when checkout.

The main functions of the application, such as CRUD of product, are implemented as RESTful web services. The SoRC provides a rich UI to use the services. The Cart (shopping cart) and CartItem are client-side data models that are stored in the local data sources. The CartItems associate a Cart. We also implement a simple cache mechanism to cache User and Product in execution to achieve better performance and user experience. The cache mechanism saves data into the local storages and therefore makes more persistent operations in the client-side data sources.

A. Performances Analysis of Different Data Sources

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

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

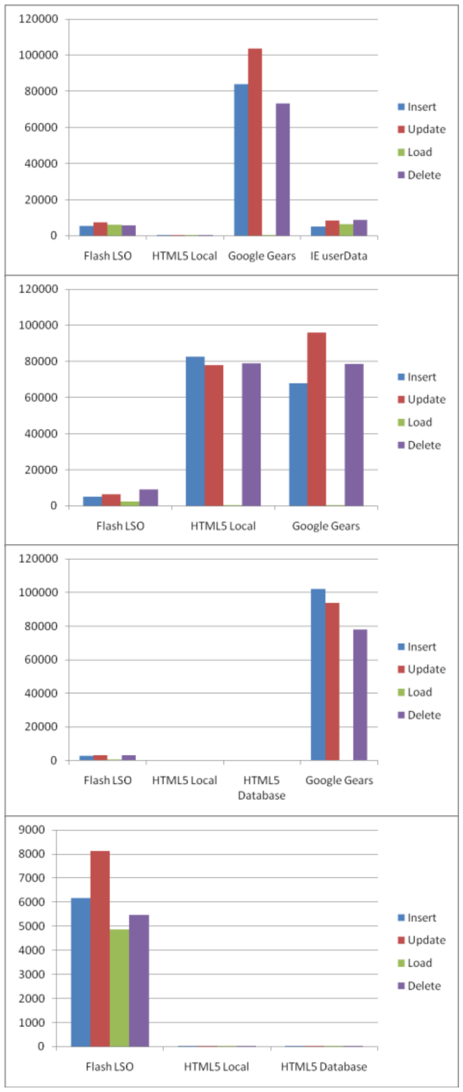


Figure 5 CRUD performances in IE8, Firefox 3.6, Chrome 5 and Safari 4

- IE userData data source is the IE's private local storage solution. It is the only available data source in IE 7 and lower;
- The performance of Flash LSO data source is very similar in different web 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 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 requirement.

In the second test, we compare the performances of a same local storage solution in desktop and mobile browser. The test runs in Safari 4, which has desktop and iPhone version. In this test, each operation also repeats one thousand times. The result is shown in Figure 6. Since Flash LSO is not supported by iPhone Safari, the Flash LSO related data is not displayed in the figure.

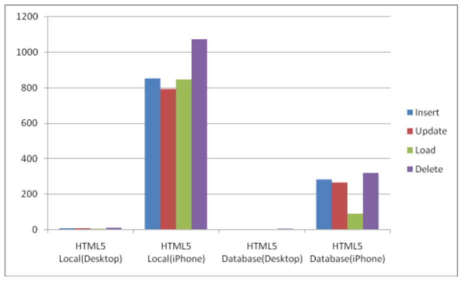


Figure 6 Performance Comparison of HTML5 in Safari Desktop and Mobile

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 implementation, Webkit10, the gap of performance should be mainly due to the performance difference between PC and mobile phone. However, if comparing the HTML5 performance in iPhone Safari with some other data sources in desktop browser, such as Flash LSO and Google Gears, we can find the iPhone Safari's HTML5 has excellent 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 7 displays the comparison results between the simulated data source with Google Gears data source, which has the slowest average performance in all local storage solutions.

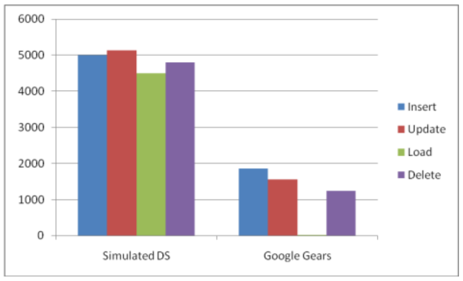


Figure 7 Performance Comparison of Local Data Source and Simulated Data Source

The performance of simulated data source is much slower than the slowest local data source. Such big performance gap is due to that each operation in the simulated data source is a remote invocation. Moreover, in our testing environment, the server and browser are deployed in a same LAN, where network latency is low. The performance of simulated data source will be worse in production environment – the performance will decrease as network latency increased. Therefore, the simulated data source is only the last resort – when none of the others client-side data sources are available.

B. *Evaluation of Data Sources Selection*

We evaluate the effect of our local storage selection approach though a script which simulates users' actions and invokes requests on behalf of a user. We make the cache store data in the local storage rather than the memory to obtain more local data sources visits. In this case, the rich client 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 to the following table.

TABLE III. PERFORMANCES AND SIZE LIMITS OF DIFFERENT LOCAL STORAGES IN GOOGLE CHROME 5 AND FIREFOX 3.6

Google Chrome	**Read**	**Insert**	**Update**	**Delete**	**Size Limit**
Flash LSO	0.70ms	2.87ms	3.37ms	3.30ms	100K
HTML 5 Local	0.09ms	0.32ms	0.31ms	0.38ms	5000K
HTML 5 Database	0.001ms	0.002ms	0.002ms	0.002ms	5000K
Google Gears	0.001ms	101.96ms	93.80ms	78.00ms	No Limit

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

Then, we perform the test with the three ways of assigning the characteristics variable of rich clients, as we mentioned in section 4.2. In the manual way, $S_{app-max}$ is assigned to 100K due to no need for mass local storage in this test. And 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.

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

---

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

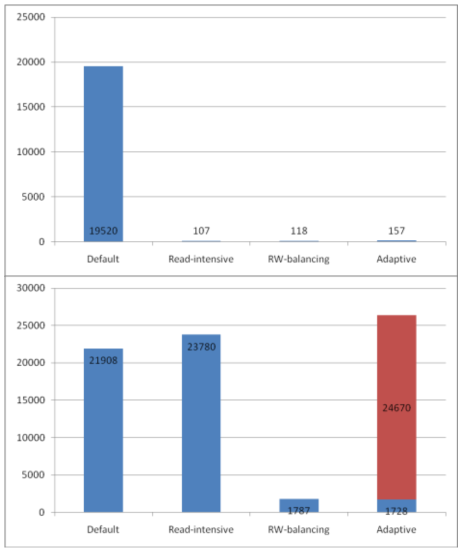


Figure 8 Time Consumption of Different Strategies in Chrome 5 and Firefox 3.6.3

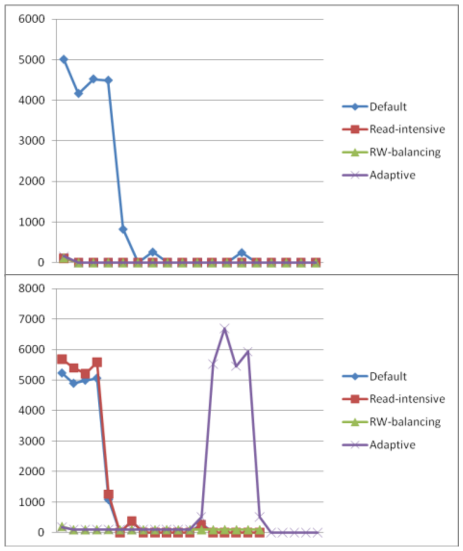


Figure 9 Time Consumption of Every 50 Operations in Chrome and Firefox 3.6.3

The above results reveal the following observations:

- The rich client cost the longest time by default. The default strategy has to no idea about the size requirement of application. The strategy therefore always tend to select Google Gears data source, which is the only currently known "infinite" local storage, to ensure that data will not overflow. However, the write operations in the Gears data source work extremely slowly. As the blue lines shown in Figure 9, 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 complicated. Counts afterward demonstrated that the ratio between read and write operations in the test is three to one. 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 because that Chrome's HTML5 database executes 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 gets poor performance, similarly with the default situation. But the RW-balancing strategy achieves a better result. The reason is that although Gears data source gains 2ms (millisecond) with each read operation, it slow about 70-80ms with each write (CUD) operation compared to 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 solutions are complex, to find the most suitable 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 way fine-gained sets $< W_c, W_r, W_u, W_d >$ value based 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 manually strategies (also use HTML5 database). It is possibly due to the operation logger spends a litter time.
  - ✧ 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 Figure 8 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 Figure 8 demonstrates the cost of data source migration, which need 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, we believe a more rational way to select data source is that recording historical operations but migrate data source before application closed rather than during application execution.

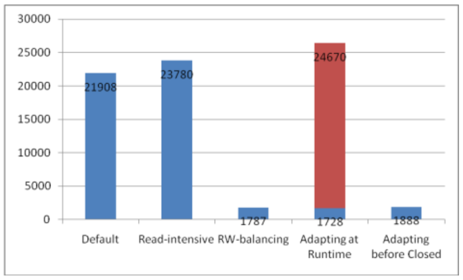


Figure 10 Time Consumption of Different Strategies in Firefox 3.6.3

The right blue bar in Figure 10 illustrates that the total processing time of re-run the test after re-selecting data source before the RIA closed. This way selects the most suitable data source (Flash LSO) neither depending on unreliable coarse-grained manual strategy nor affecting the total time consumption and user experience seriously.

VI. DISCUSSION

Object-relational mapping (ORM) is a mature topic [11]. In practice, there are many complicated 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 complex data models yet, most of the functions are not necessary characteristics thus. On the other hand, since the data access 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 not uppermost concern within this paper. So far, our framework only adopts the simple ORM strategies, and developers should still set up complex mappings manually. In our future work, we will try to introduce more powerful ORM mechanisms into the framework and investigate how the mechanisms affect rich client data access.

Currently, our local data sources selection approach only consider the performance of CRUD operations. However, the powerful but complicated aggregate operations also have great influence on the performance of local storages and further greatly affect the data sources selections. For example, SQL solutions, such as Google Gears, support single and much condition inquiry based on SQL. In non-SQL local storages, such inquiries have to be implemented by traversing the whole table, and therefore must be much slower. Accordingly, considering the performance of aggregate operations, the SQL local storage solutions can be applicable for more RIAs in some browsers (e.g. Firefox). It is able to deduce that the selection approach in this paper still can be used to deal with this condition with extended performance vector and weighting vector. Nevertheless, related evaluation is still required.

VII. CONCLUSION

Service-Oriented Rich Internet Applications combine the benefits of the Web distribution model with the highly interactive desktop applications. SoRIAs move amount of data and application logics from server-side to SoRCs. Thererfore, the SoRCs suffer from client-side data access issues. In this paper, we propose a data access framework for SoRCs. This paper makes the following contributions: 1) An adapter for shielding heterogeneous data sources (SQL or Non-SQL) and a set of unified APIs for hiding incompatible access APIs provided by different solutions; 2) An algorithm for selecting the most suitable local storage for current SoRIA and browser in use; 3) A performance analysis of heterogeneous local storage solutions and some experiments for evaluating the feasibility and effectiveness of the framework.

As we discussed, there are still some open issues for our data access framework. We will try to address these issues in the future work.

VIII. ACKNOWLEDGEMENT

This work is supported by the National Basic Research Program of China (973) under Grant No. 2011CB302604; the National Natural Science Foundation of China under Grant No. 60821003, 60873060, 60933003; the High-Tech Research and Development Program of China under Grant No. 2009AA01Z116; and the National S&T Major Project under Grant No. 2009ZX01043-002-002.

REFERENCES

[1] Qi Zhao et al., A Browser-based Middleware for Service-Oriented Rich Client. International Conference on Service Science (ICSS), 2010.

[2] L. M. Haas et al., Transforming Heterogeneous Data with Database Middleware: Beyond Integration. IBM, 2001.

[3] Malcolm Atkinson, and Ronald Morrison, Orthogonally Persistent Object Systems. VLDB Journal, 1995.

[4] Teng teng, Research on Pragmatics-based Persistence. Doctoral dissertation, Peking University, 2007.

[5] M. Driver et al., Rich Internet Applications Are the Next Evolution of the Web. Technical report, Gartner, May 2005.

[6] J. Duhl, White paper: Rich Internet Applications. Technical report, IDC, November 2003.

[7] J. C. Preciadol et al., Designing Rich Internet Applications with Web Engineering Methodologies. WSE, 2007.

[8] Alessandro Bozzon, and Sara Comai, Conceptual Modeling and Code Generation for Rich Internet Applications. ICWE, 2006.

[9] Santiago Meliá et al., A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. ICWE, 2008.

[10] Zani, G.P., "Expert Database Systems: State of The Art", Tutorial Documents of the First World Congress in Expert Systems, USA, 1992.

[11] Scott W. Ambler, Mapping Objects to Relational Databases: O/R Mapping In Detail. http://www.agiledata.org/essays/mappingObjects.html, 2006.

[12] Sam Ruby et al., Agile Web Development with Rails (3rd Edition). Pragmatic Bookshelf, 2009.