Web API Database Systems for Rapid Web Application Development

Takeru INOUE\textsuperscript{a)}, Member, Hiroshi ASAKURA\textsuperscript{a)†††}, Yukio UEMATSU\textsuperscript{a)}, Hiroshi SATO\textsuperscript{b)}, Nonmembers, and Noriyuki TAKAHASHI\textsuperscript{\dagger}\textsuperscript{b)}, Member

SUMMARY Web APIs are offered in many Web sites for Ajax and mashup, but they have been developed independently since no reusable database component has been specifically created for Web applications. In this paper, we propose WarDB, a distributed database management system for the rapid development of Web applications. WarDB is designed on Atom, a set of Web API standards, and provides several of the key features required for Web applications, including efficient access control, an easy extension mechanism, and search and statistics capabilities. By introducing WarDB, developers are freed from the need to implement these features as well as Web API processing. In addition, its design totally follows the REST architectural style, which gives uniformity and scalability to applications. We develop a proof-of-concept application with WarDB, and find that it offers great cost effectiveness with no significant impact on performance; in our experiments, the development cost is reduced to less than half with the overhead (in use) of response times of just a few msec.

\textit{key words:} Web API, database management system, REST

1. Introduction

Web APIs are a set of interfaces designed to support interoperable machine-to-machine interaction over HTTP. They have been developed for Asynchronous JavaScript and XML (Ajax)\textsuperscript{[12]} as well as mashup-based Web applications\textsuperscript{[37]}. Ajax improves the user’s perception of interactivity by enabling clients (typically, JavaScript codes) to retrieve data from a server’s API without blocking the user’s interaction. Today, many Web applications, like Gmail and A9.com, provide Web APIs for Ajax. Mashup-based applications combine data retrieved from multiple Web APIs to create a new service. Mashup has become popular in the last few years by virtue of its rapid integration. There are some tools (e.g. Yahoo! Pipes) that can help developers in creating mashups.

In the early 2000’s, developers had to design their own Web APIs. These APIs permitted similar Create, Read, Update, and Delete (CRUD) operations against Web resources, but they differed from each other because of the lack of standards. In the middle of the 2000’s, IETF released a set of standards called Atom\textsuperscript{[16], [29]}; it offers a unified approach to Web APIs for the CRUD operations. Recently, several Web APIs have been designed around Atom.

Unfortunately, developers are still being forced to develop their own data store. Some data stores equipped with Atom have been developed (e.g. mod_atom\textsuperscript{[5]} and Atom-Server\textsuperscript{[21]}), but they provide neither efficient access control nor easy extension methods. The lack of access control yields inefficient filtering of outside data stores against search results, and the lack of an extension method prevents developers from adding extra functionalities. As a result, Atom stores are rarely used, thereby increasing the development cost of applications that use Web APIs. We believe that this continued reinvention of the wheel should (and can) be avoided.

In this paper, we propose a novel database management system called WarDB for the rapid development of Web applications with APIs. Since applications are allowed to interact with WarDB directly via a Web API (an extended Atom protocol), there is no need to translate the API into SQL; developers are released from the need to re-implement the API processes. Moreover, WarDB has the features essential for Web application development including an efficient access control model, an easy extension mechanism, and search and statistics capabilities. It is designed as a distributed database system following the REST style\textsuperscript{[11]}, which gives uniformity and scalability to applications. We implement this new database management system and use it to develop proof-of-concept applications, a simple microblog and a photo-sharing service. Experiments reveal that it offers great cost effectiveness with no significant impact on performance.

The design concept of WarDB is largely oriented towards applications like social networking and photo-sharing services, since they often offer Web APIs. We, however, believe that it will be adopted by a wide variety of other services because of its generality. We focus on applications in which data is accessed mostly through Web APIs. Such an architecture brings design simplicity as well as better interactivity\textsuperscript{[22]}.

The remainder of this paper is organized as follows. Section 2 gives the necessary background, and Sect. 3 discusses requirements for WarDB. In Sect. 4, we propose WarDB for the rapid development of Web applications. In Sect. 5, we introduce the proof-of-concept application and evaluate its cost effectiveness. Section 6 summarizes related work, and finally Sect. 7 concludes with our contributions.

\textsuperscript{a)} E-mail: takeru.inoue@ieee.org
DOI: 10.1587/transinf.E93.D.3181
and future work.

2. Background

This section presents the background to our work. We first examine architectures for Web API applications as well as traditional Web sites. Well-known Web API technologies are also discussed.

2.1 Web Application Architectures

Figure 1 (a) depicts an example of the traditional Web site that follows the multi-tier architecture [10]. Client requests are processed by a logical tier component (e.g. an application server), which makes responses by retrieving resources from a data tier component (e.g. an RDBMS). While the logic tier component implements its own business logic that depends on service, the data tier component provides common CRUD operations through well-defined interfaces (e.g. SQL). The data tier component can be reused, thereby reducing the development cost.

Web applications that provide APIs have a different architecture, as shown in Fig. 1 (b). While the whole server-side works as a data tier, business logic is moved to the client-side. Clients (typically, JavaScript codes running on a Web browser) are allowed to dynamically update pages by requesting for resources through the API. Unfortunately, application servers are not reusable despite their similarity.

2.2 RESTful Web APIs

This subsection explains Atom. Atom is often called a RESTful Web API, since it follows the REST architectural style [11]. REpresentational State Transfer (REST) is the architectural style used to design key Web objects such as HTTP and URI. Its central characteristic is interface uniformity, which decouples implementations from the services and gives better interoperability. In RESTful APIs, operations are simply specified by pairs of an HTTP method and a URI. In REST style, servers must be stateless; they are not allowed to keep any state for any client. Requests can be processed by multiple servers in parallel, which improves scalability. RESTful Web APIs have been gaining popularity because of their simplicity, interoperability, and scalability [31]. We do not discuss non-RESTful technologies such as XML-RPC [23] and SOAP [2].

Atom is the only RESTful standard for which a protocol as well as a format has been published. The CRUD protocol is called Atom Publishing Protocol (AtomPub) [16], and the format is referred to as Atom XML [29]. Figure 2 (a) presents the abstract resource model in Atom; data called members are organized in containers called collections. They are linked as shown in Fig. 2 (b); the service document has links to collection feeds, which link to members in the same collection. Figure 3 shows examples of Atom XMLs. For member creation, only a media resource is transferred from a client to the server, which generates the associated media link entry automatically. It is not allowed to create, update, or delete multiple members at a time.

Since Atom is designed as a basic instruction set for building other Web APIs, its protocol and format are highly extensible; while the extensibility of the protocol depends on the structure of URI and HTTP headers, that of the format relies on the namespace mechanism of XML. Its extensions include Google Data APIs (GData), Open Data Protocol (OData), and Windows Azure REST API.

††Actually, typical Web sites include other types of components, such as a reverse proxy (often referred to as the presentation tier), a load balancer, and a cache, but we ignore them in this paper without loss of generality.

3. Requirements

This section discusses the requirements for WarDB. We first discuss reusability followed by the data model and ACID properties. We also investigate several of the advantages yielded by WarDB. The requirements discussed here originate from Web application development or the REST style, as shown in Table 2.

**Reusability.** We define reusability as the degree in which a part of the source code can be used again with slight or no modification. This property induces three desirable consequences; a reduction of the number in code lines newly written, a reduction in the number of bugs included in the software, and a reduction in the dependency between software components. Improving this property is the main force motivating this paper, since it reduces the development cost and enables rapid development. In order to reuse WarDB in actual services, their own requirements such as advertising, search, and statistics, are allowed to be incorporated. The API should be based on a standard that defines a protocol as well as a format.

**Hierarchical resource model.** Web resources accessed through the RESTful APIs follow a hierarchical model, not the relational model. As shown in Sect. 2.2, S3 and Atom follow a fixed hierarchical model (i.e., bucket/object and collection/member). The hierarchical model has an advantage; it is straightforward to partition data sets in the hierarchical model, while this is complicated in the relational model [30, Chap. 5.3]. This advantage is indispensable since Web resources can be voluminous.

**Relaxed ACID properties.** ACID (Atomicity, Consistency, Isolation, and Durability) [14] is a set of properties that guarantee reliable transactions. It is often relaxed in the REST style. Atomicity is limited to transactions of a single operation, because RESTful servers are not allowed to keep state between operations. Consistency is guaranteed by clients, not servers, except for referential consistency of the hierarchy. Each transaction is executed in an isolated manner by

---

**Table 1**  Key elements in Atom XML.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>content</td>
<td>Contains or links to the content.</td>
</tr>
<tr>
<td>author</td>
<td>Indicates the author.</td>
</tr>
<tr>
<td>category</td>
<td>Conveys information about a category associated with.</td>
</tr>
<tr>
<td>link</td>
<td>Defines a reference to a Web resource (e.g. a media resource).</td>
</tr>
<tr>
<td>title</td>
<td>Conveys a human-readable title.</td>
</tr>
<tr>
<td>updated</td>
<td>Indicates the last modified time.</td>
</tr>
</tbody>
</table>

**Table 2** Requirements and their origin.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusability</td>
<td>Web application development</td>
</tr>
<tr>
<td>Hierarchical resource model</td>
<td>REST style (especially Atom)</td>
</tr>
<tr>
<td>Relaxed ACID properties</td>
<td>REST style</td>
</tr>
<tr>
<td>Efficient access control</td>
<td>Web application development</td>
</tr>
<tr>
<td>Easy extension</td>
<td>Web application development</td>
</tr>
<tr>
<td>Search and statistics</td>
<td>Web application development</td>
</tr>
</tbody>
</table>

---

*We discussed properties needed to support fragmentation here, but fragmentation algorithms are beyond the scope of this paper.*
utilizing concurrency control technologies (we ignore phantom read). Durability is required as in RDBMS.

**Efficient access control.** WarDB cannot be used for account management because of the limitations placed on transactions. This implies that user accounts are managed in a traditional RDBMS, while Web resources are stored in WarDB. Since access control requires account information as well as Web resources, distributed join operations are inevitable between the RDBMS and WarDB. Efficient join operations need a query execution plan that minimizes communication costs (e.g. traffic and round trips) [30, Chap. 7–9]. In addition, the access control model should take advantage of user relationships (e.g. friends).

**Easy extension.** WarDB is required to provide an easy function extension mechanism. Extension should be possible without source code. Moreover, no restrictions should be placed on the programming languages used to develop extensions. In this paper, we assume that WarDB would be extended mainly for resource modifications in CRUD operations, such as advertising and metadata addition.

**Search and statistics.** Search and statistics are essential functions in most Web applications. They should be built-in since they require additional indexes and tables, even though WarDB should have an extension mechanism. Furthermore, it is preferable that they be simple and suit Web resources.

### 4. Design

This section introduces and discusses the design of WarDB, a novel database management system with Web APIs. We first describe the architecture of WarDB. After briefly addressing resource names and ACID properties, we discuss access control model, extension method, and queries.

#### 4.1 Architecture for Reusability

WarDB ensures that application servers do not need to handle Web APIs, thereby reducing the development cost. As shown in Fig. 1 (c), WarDB is placed behind the application server. Upon receiving a request for Web resources in WarDB, the application server authenticates the request, if needed, and forwards it to WarDB with additional headers for access control as described in Sect. 4.4.

Figure 4 shows the configuration of WarDB. The controller is an HTTP server that executes access control and API processing. The full-text index (a search engine) and storage are used to manage Web resources. They can be a single component or distributed among multiple machines (fragmentation of the index and/or storage is beyond the scope of this paper). For a write query, the controller updates the index and saves the posted resource in the storage. For a search query, the controller searches the index and retrieves the corresponding resources from the storage.

The architecture of WarDB, shown in Fig. 1 (c), reduces the development cost of application servers, which results in the rapid development of Web applications. In this architecture, Web application developers are not required to implement resource management including access control. They implement only service-dependent functions such as user management in the application server. WarDB is a reusable component and needs no modification.

This architecture (Fig. 1 (c) with Fig. 4) degrades latency, since messages must pass through more machines. The impact of this is evaluated in the experiments in Sect. 5.

The design of WarDB is largely based on Atom, because it is the only standard that meets the requirements listed in Sect. 3.

#### 4.2 Resource Names

A resource name (URI) consists of base, collection, and member parts. The collection part can include a user name (e.g. joe). (b) Collections are configured by the administrator in advance. Their path can include a “:user” term, which is interpreted as actual user name.

<table>
<thead>
<tr>
<th>(a) URI structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://example.org/photo/joe/son.jpg">http://example.org/photo/joe/son.jpg</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) Collection Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>collections:</td>
</tr>
<tr>
<td>- title: user’s photo collection</td>
</tr>
<tr>
<td>path: photo/user</td>
</tr>
</tbody>
</table>

Fig. 5 Examples of a resource URI and collection settings. (a) A URI consists of base, collection, and member parts. The collection part can include a user name (e.g. joe). (b) Collections are configured by the administrator in advance. Their path can include a “:user” term, which is interpreted as actual user name.

1Since WarDB does not manage user accounts, it does not know whether the user actually exists. However, this does not matter, because write queries require authentication at the application server.
Member. Member parts are determined by WarDB by using a title element and/or the current time. Clients are allowed to suggest them by the Slug header in the request.

4.3 Implementing Relaxed ACID Properties

WarDB implements the relaxed ACID properties. We assume that the index and the storage provide atomicity for single operations and durability by themselves. Referential consistency is guaranteed by the controller. To execute transactions between the index and the storage in an isolated manner, a distributed transaction algorithm (two/three-phase commit or minitransaction in Sinfonia [1]) or optimistic concurrency control is used.

4.4 Efficient Access Control Model

Assuming that users are allowed to access the resources of their friends, we first discuss a query execution plan for distributed join operations. Account information is managed in an RDBMS, while Web resources are stored in WarDB. A write query poses no significant difficulty, since it involves just a single resource. On the other hand, a search query can involve a large amount of resources, and so an efficient plan is imperative for minimizing the cost of the distributed join operation. Figure 6 shows three possible plans.

1. The first plan sends the entire set of matched resources to the application server from WarDB, and filters them at the server. Unfortunately, this creates excessive traffic, R’, between the server and WarDB.

2. The second plan utilizes semi-join; it returns only the primary key and friend columns; filtering is done at the server followed by retrieval of the body of the desired resources from WarDB. This plan reduces the traffic compared to the first plan, but does require an extra round trip.

3. The third plan sends user’s account information (e.g. friend list) to WarDB and resources filtering is performed in the search process. The account information, U’, is much smaller than search results, R’, and no extra round trip is required. Moreover, complicated joins do not need to be implemented in the application server.

We choose the third plan, since it yields low communication cost for most queries. The existing Atom stores, such as modAtom and AtomServer, cannot execute our plan since it requires the join operation shown in Fig. 6 (3).

Next, we discuss the detailed behavior of our access control model. In the model, the user has the privilege of setting the visibility of all his/her resources. Here visibility has a three-grade scale: “only me”, “friends”, or “everyone”. If Joe sets “friends” visibility to his resources, only he and his friends are allowed to access them. If Joe sets “friends” visibility to resource X, while “only me” visibility to resource Y, his friends are unable to access Y.

Privileges for user’s resources. In this case (Fig. 7 (a)), users are forced to set the visibility of their resources in advance. The visibility is stored in the RDBMS with account information. For a write query, the application

<table>
<thead>
<tr>
<th>User table @ RDBMS</th>
<th>Resource table @ WarDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>user</td>
<td>friend</td>
</tr>
<tr>
<td>joe</td>
<td>[kate]</td>
</tr>
<tr>
<td>kate</td>
<td>[joe,meg,tom]</td>
</tr>
</tbody>
</table>

Fig. 6 Three possible execution plans for distributed join operations for access control in the relational algebra; the example is Kate finding her friends’ resources. The operations are managed by the application server between the user table in the RDBMS and the resource table in WarDB (the tables are not normalized due to space limitations). (1)st plan creates excessive traffic R’. (2)nd plan requires an extra round trip. (3)rd plan reduces the traffic and requires no extra round trip. (This figure shows that resources are stored in an RDB, but this is just for readability. Of course, WarDB is not an RDB. Section 5.1 describes how to store resources in WarDB.)
Fig. 7 Examples of our access control model, which efficiently executes distributed join operations between the user table in the RDBMS (not shown in the figure) and the resource table in WapDB. (a) Joe sets his resource visibility to “friends”, which allows Kate (Joe’s friend) to access them. (b) When Joe sets “friends” visibility to a resource, Kate is allowed to access only the resource. (Though it seems that resources are stored in an RDB and accessed with SQL in this figure, this is just for easy understanding. As discussed in Sect. 5.1, resources can be stored in non-relational database systems to offer full-text search and for better performance and scalability.)

Privileges for every member resource. In this case (Fig. 7 (b)), the visibility is specified whenever a resource is created or updated. For a write query, the application server authenticates the query, and forwards it with the X-WapDB-Visibility header containing the specified visibility. WapDB stores the resource into the resource table. The owner column includes the user name. The friend column contains null for “only me”, the user name for “friends”, or the reserved term for “everyone”. Finally, WapDB executes a search over resources whose owner column is the user name, or the friend column of one of his/her friends or the reserved term for “every-
Fig. 8 Trigger-like extension mechanism. (a) Trigger events and action servers are specified in the RESTful way. (b) The posted resource is transferred to the action server that provides extensions if the request matches the corresponding event. (c) The media resource and the associated media link entry are packaged into a multipart/related representation before being transferred to the action server.

one”, by adding a filtering predicate to the original query (owner=kate OR friend IN (...) in Fig. 7(b)).

We will evaluate the efficiency of this access control model in the experiments.

Our model does not violate the REST style, since it introduces no state management.

4.5 Trigger-Like Easy Extension

WapDB provides easy extension through a trigger-like mechanism. Triggers[26] are commonly supported in RDBMSs; SQL actions are executed in response to certain events if given conditions are met. They are often used to log changes and execute business rules. Since WapDB follows the REST style, our triggers are provided in the RESTful way; events are specified by a pair of HTTP method and URI, and actions are implemented as HTTP servers†, as shown in Fig. 8(a). For simplicity, conditions are not specified. Triggers of WapDB are row triggers, not statement triggers, since it is not allowed to edit multiple resources (rows) at a time. Triggers are evaluated in the order of setting.

Figure 8(b) is an example of trigger processing upon receipt of a request for creating a new resource. WapDB controller evaluates the posted resource, and generates an associated media link entry if needed. The resources are forwarded to all action servers whose events are matched to the request (if a media link entry is generated, the media link entry as well as the posted media resource is packaged as a single multipart/related representation[15], as shown in Fig. 8(c)). The action servers process the resources, which are returned to the controller. The controller finally saves the returned resources into the index and storage‡.

Our trigger system takes advantage of Atom’s extensibility, because action servers can evaluate Atom’s extensible parts: URI, HTTP headers, and XML (entry). Since WapDB and action servers are loosely coupled through HTTP, the programming languages used are not restricted. Trigger overheads, extra round-trips and other XML parsing operations, are evaluated in the experiments described later.

4.6 Search Queries and Frequency Distribution

WapDB uses the GData query model because it has a proven track record, has been operational for several years in Google, as well as being simple and Atom-friendly. Table 3 presents some of the GData query parameters (they are expressed as HTTP URIs). Clients are allowed to select resources with full-text search and range queries for key Atom elements. Results are sorted in descending order of last modified time. If the number of results per page is limited, a link to the next page is given with the start-index.

††HTTP server development is now tractable due to recent advances in Web application frameworks like Ruby on Rails and Catalyst[34].

†1We briefly discuss status codes returned by action servers. On an error (4xx or 5xx), the controller stops the processing and returns the received error status. On a redirection (3xx), the controller follows the new location. On success status (2xx), the controller continues to process the returned resources. Even if the controller finds some error with the resources, it should return the server error (500) to the client, not the client error (400). Continue status (100) is forbidden; it should be treated as server error.
Table 3  Key query parameters in GData.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>Full-text query string. e.g. ?q=term1 term2</td>
</tr>
<tr>
<td>/-/category</td>
<td>Category filter. e.g. /photo/joe/-/family</td>
</tr>
<tr>
<td>author</td>
<td>Entry author. e.g. ?author=joe</td>
</tr>
<tr>
<td>updated-min, updated-max</td>
<td>Bounds on the entry update date. e.g. ?updated-min=2009-01-01T00:00:00Z</td>
</tr>
<tr>
<td>start-index</td>
<td>1-based index of the first result to be retrieved. e.g. ?start-index=1</td>
</tr>
<tr>
<td>max-results</td>
<td>Maximum number of results to be retrieved. e.g. ?max-results=10</td>
</tr>
</tbody>
</table>

(a) Pseudo schema for full-text index

create table resource_index (  
# resource name (/prefix/user/id)  
member_id char,  
collection_user char,  
collection_prefix char,  
  
# access control  
friend char,  
  
# synchronization between index and storage  
timestamp integer,  
  
# GData  
title fulltext,  
body fulltext,  
category fulltext,  
# used as vector  
author char,  
updated_datetime  
);  

(b) Schema for storage

create table resource_storage (  
id integer primary key auto_increment,  
  
resource_name char unique,  
# (/prefix/user/id)  
  
# synchronization between index and storage  
timestamp integer,  
  
entry_resource blob,  
media_resource blob,  
content_type char  
);  

Fig. 9 Frequency distribution shown in the service document (indicated by dashed square).

Fig. 10 Schemas used in the experiments for full-text indexing and storage. Though the index has its own query language, we describe the schema of (a) in SQL for readability (this is a pseudo schema that has the same semantics).

5. Experiments

In this section, we evaluate the cost effectiveness and performance overhead of WapDB.

5.1 Implementation

We implemented a prototype of WapDB on the LAMP (Linux, Apache, MySQL, Perl) stack [24], as shown in Table 4. infobee/evangelist[19], which is a search engine utilizing an inverted index, was used for GData full-text queries. The controller translates AtomPub and GData queries into a query language implemented for the full-text index. This translation is tractable since no join operation is required (join in Fig. 6 is replaced by the IN predicate as shown in Fig. 7). We implemented this manually without any difficulty. The controller implemented data validation logics based on the Atom’s specification†. Service-dependent validation, e.g., validation against media resource format, would be implemented in the action servers.

Figure 10 shows the schemas used in the experiments. We created a complex index for each part of a resource name (prefix, user, and id in Fig. 10(a)) to search for resources that follow the hierarchical resource model. This schema enables WapDB to find an individual resource as well as all resources in the same collection quickly. Storage keeps structured data by name, as shown in Fig. 10(b). We used MySQL as the storage system in our experiments, but the storage requirements are just storing data and finding them by name, like a file system. Distributed file systems or so-

†We implemented most of Atom’s validation logics, but some of them were left unimplemented due to its complexity. This is our future work.
Fig. 11 Experimental setup. Server-side (the dashed square) consists of three machines (the thick squares); each of them runs the application server with the RDBMS, three components of WapDB, and two action servers. These machines have a quad-core processor (Intel Xeon 2.13 GHz), and are connected by Gigabit Ethernet.

Fig. 12 An image of the simple microblog showing friends’ posts. This application was developed with WapDB; resources are stored in WapDB with access control, but triggers, search, and statistics features were not implemented.

called key-value stores could be used as the storage.

By adding a timestamp to each record, the controller can find any inconsistency between the index and storage. This simple optimistic concurrency control keeps the index and storage consistent (synchronized), so atomicity and isolation are guaranteed in transactions of a single operation (we assume that the index and storage provide atomicity for a single operation). The current implementation does not roll back transactions even if conflicts are found, since our expected Web applications do not need strict conflict resolution (simply, the record with newer timestamp wins). Data consistency should be guaranteed by clients, as mentioned in Sect. 3. We assume that durability is guaranteed by the index and storage.

We also developed proof-of-concept applications, a simple microblog and a photo-sharing service. Figure 11 shows the experimental setup, and Figs. 12 and 13 are images of the applications. In the experiments, users are given a privilege to set the visibility of all his/her resources (the case of (a) in 7), and so visibilities are stored in the RDBMS with the normal account information. Web resources are managed in WapDB; resources are users posts in the microblog, and photos (media resources) and their metadata (media link entries) in the photo-sharing service. In the photo-sharing service, two triggers are fired on creating and updating resources. One is for advertising; it inserts an advertisement matched to the photo’s category into the media link entry. The other trigger extracts the location from the photo’s Exif data, and adds it to the associated media link entry.

We confirmed that the application could be developed based on the hierarchical resource model and the relaxed ACID properties without any difficulty. It works well with the Atom client named Windows Live Writer [36], whose conformance to Atom was verified in Atom interop experiments1.

5.2 Evaluation

We developed the applications with and without WapDB, which are based on architectures of Figs. 1 (c) and (b)††, to evaluate the cost effectiveness and performance overhead of

†The results are found at http://www.intertwingly.net/wiki/pie/November2007Interop.

††To support GData full-text queries, a full-text index as well as an RDBMS was used even in the case of Fig. 1 (b).
our proposal.

We evaluate cost effectiveness based on reusability. Reusability leads to three advantages as we noted in Sect. 3, but we evaluate just the first one, the number of code lines newly written, because it is difficult to accurately evaluate the others. Figure 14 shows the lines of code needed to create the applications. In our experiments, the use of WarDB cut the number of lines needed by more than half in both applications. This is mainly because WarDB eliminates the need to implement Atom processing, access control, search, and statistics. We implemented only business logics at the client, account management, and header manipulations for access control at the application server. Actual development period was reduced by roughly half. We found no difficulty in developing the action servers thanks to recent advances in Web application frameworks (e.g. Ruby on Rails and Catalyst); it required only 168 lines of code. In addition, the development cost of action servers can be shared, because common action servers (e.g. advertising) will be reused by other services.

In our experiments, 2,186 lines were needed to create the controller in WarDB. This development cost can be amortized with just two Web applications as large as the photo-sharing service, since WarDB reduced the code lines by 1,327, which is about half of 2,186.

Next, we observed the response time for evaluating performance overhead in the photo-sharing service. The response time was measured at the client for creating a resource and searching for the resources of a specified category (i.e. /-/<category> in Table 3). While the posted resources were photos with average size of 120 KB, the associated media link entries were 4.0 KB in size on average. The client received, at most, the top 10 entries for each category query. We repeated the measurements 1,000 times and determined the 1st, 50th, and 99th percentiles.

In addition to implementing the application twice (i.e. with and without WarDB), we introduced another implementation named WarDB* in order to distinguish the overhead of the new architecture from that of triggers. In this implementation, the application was developed with WarDB based on the architecture of Fig. 1 (c), but triggers were not used (advertising and Exif extractor were implemented in the application server). For resource creation, new architecture overhead is the difference between response time of “without WarDB” and that of “with WarDB*”, while trigger overhead is the difference between “with WarDB*” and “with WarDB”. In terms of searching, the overhead of the new architecture is the difference between “without WarDB” and “with WarDB”.

Figure 15 depicts the response time for each implementation. The figure shows that the overhead of WarDB is not significant; on average, the new architecture overhead is 7.5 and 6.7 msec. for creation and search, respectively, while trigger overhead is 22.0 msec. The impact of searching can be considered even smaller in practice, because some results would be retrieved from cache servers. Although trigger overhead is slightly larger, it does not matter because write queries are much less common than read ones. Long bars in creating a resource indicate a large fluctuation in the response time. It depends on the size of the posted photo, not WarDB, since they are seen with or without WarDB.

We do not show the impact of huge access loads, since they mainly depend on performance of the full-text index and storage systems, which is beyond the scope of this paper. Here, we briefly discuss performance and parallelism under huge access loads.

We observed CPU utilization of the controller machine accessed by multiple clients; the controller was configured to run up to 24 processes, and 16 clients continued to send requests in parallel. The CPU utilization was nearly 100% throughout the experiments for resource writing and searching (response time was roughly doubled). The results showed that the controller had no heavy lock contention. This is because our simplified optimistic concurrency control relies on timestamps, not locking. A process does not need to wait for other processes to complete their transactions provided they are manipulating other rows (we assume that the full-text index and storage provide row-level locking). Moreover, the timestamp manipulation is very fast; it needed less than 1.0 msec. in our experiments.

Our access control process acquires locks on multiple rows in the user account database, but the period was very short, only 1.2 msec. in our experiments (the database maintained 1,000 users and about 5,000 friendships). More-
over, this access can be accelerated by cache technologies like memcached. Though these would introduce some update delay, this would not cause significant problems in most Web applications.

6. Related Work

We found a couple of existing Atom storage schemes other than WarDB. mod_atom [5] is an Apache module that allows clients to access file systems through AtomPub. While its deployment is very easy, it provides no search query, statistics, extension mechanism, or access control mechanism based on user relationships. AtomServer [21] is a data store providing some part of the GData queries as well as AtomPub. However, it provides neither access control nor full-text query. Its extensibility is limited to XML validation.

There has been some work on equipping RDBMSs with HTTP interfaces. Shadgar et al. mapped SQL statements to the WebDAV methods [35]. Munk-Stander re-designed Atom to handle the relational model in his master thesis [27]. While their goal is to manage relational data through HTTP interface, our goal is to manage Web resources that follow the hierarchical resource model.

Geambasu et al. proposed Menagerie, a FUSE (Filesystem in USErspace) proxy to access Web APIs [13]. Unfortunately, Menagerie provides no search query and statistics, since it is based on a file system like mod_atom.

Wiki [25] and Content Management System (CMS) [18] allow Web sites to be constructed without programming effort. However, they have a few restrictions on cache deployment and resource model, because they are provided as an all-in-one package. WarDB has no such limitation, as WarDB is just a data tier component.

Klaim, a kernel programming language designed to yield distributed systems consisting of several mobile components, provides mechanisms to customize access control policies in distributed environments [8], but efficient execution is beyond their scope. The access control method for XML documents has been studied well [7]. Their target is a fine-grained access control method that controls access to an element or an attribute of XML documents, not a coarse-grained control with high efficiency as we discussed.

There are several query languages that can be adopted to Atom, but they are too rich and complicated to handle Web resources because of their expressiveness. The Feed Item Query Language (FIQL) [28] was designed for Atom, but it has rich predicates not limited to Atom. XPath [3] and XQuery [4] are query languages for arbitrary XML documents. SPARQL [33], an RDF query language, can be used for Atom by using Atom/RDF mapping [32].

Database triggers [26] have been investigated in detail for RDBMSs, while we discussed a trigger design that follows the hierarchical resource model in the RESTful way. Here we briefly discuss Web Application Description Language (WADL) [17], which can be used for triggers. WADL is an interface definition language that describes the RESTful Web APIs. It provides a machine-readable description for easy interactions between RESTful components. However, it is rarely used because of its complexity, and so our trigger design does not take advantage of WADL.

7. Conclusions and Future Work

This paper introduced WarDB, a novel database management system with Web APIs for the rapid development of Web applications. While Web APIs are offered in many Web sites, they are implemented independently and so fail to offer reusability. We designed WarDB as a highly reusable database component for reducing the development cost. WarDB removes from developers the need to reinvent Web API processes. In addition, WarDB provides common features in Web applications, such as efficient access control, an extension mechanism, and search and statistics capabilities. We developed a simple microblog and a photo-sharing service by using WarDB. Experiments revealed the great cost effectiveness provided by WarDB: the number of lines was reduced to less than half. In addition, the performance overhead was no more than ten msec. for searching, and several tens of msec. for creating a resource even if triggers were used.

We hope that this paper makes other contributions for ongoing movements related to Web and database systems. Distributed join operations like our access control model are expected to become more important, since the use of specialized database systems including BigTable [6] and Dynamo [9] will be common. The trigger system in WarDB is an incarnation of a RESTful Web service, which is deemed to be a promising way of realizing practical Web services. Our future work also includes role-based access control, efficient bulk insertion, and operational issues.

Acknowledgements

We would like to thank Dr. Onizuka for his valuable comments on the Introduction. We are also grateful to Mr. Okabe for his suggestion on triggers. We would like to thank Mr. Sawamura for the use of his lovely photos.

References


Takeru Inoue received the B.E., M.E., and Ph.D. degrees from Kyoto University, Kyoto, Japan, in 1998, 2000, and 2006, respectively. Since joining NTT Network Innovation Laboratories in 2000, he has been engaged in R&D on mobile networking, multicasting, and Web architecture. His research interests are in large-scale communication networks. He received the best paper award from Asia-Pacific Conference on Communications in 2005. He also received the research awards of the IEICE Information Network Group in 2002 and 2005. He is a member of IEEE.

Hiroshi Asakura received the M.E. degrees in information science from Nara Institute of Science and Technology in 1999. He joined NTT in 1999. He has been in NTT Communications Corp. since 1999. He engaged in large-scale ISP operation, business development and product R&D. His current interests are Web architecture, Web-based protocol and data management systems for the Web. He is a member of IPSJ.
Yukio Uematsu received the B.E., M.E., and Ph.D. degrees from Tokyo University of Science, Chiba, Japan, in 2001, 2003, and 2008, respectively. Since joining NTT Cyber Solutions Laboratories in 2003, he has been engaged in R&D on information retrieval, web mining and database technologies. After he moved to NTT Communications corporation on 2009, he has been engaged in R&D on recommender systems. He also joined Tokyo University of Science as an adjunct lecturer on 2010. His research interests are in data mining and large-scale data processing. He is a member of JSAI and DBSJ.

Hiroshi Sato received the B.S. and M.S. degrees in mathematics from Tohoku University, Sendai, Japan, in 1994 and 1996, respectively. In 1996, he joined NTT Communication Science Laboratories, where he engaged in research on natural language processing. He moved to NTT Network Innovation Laboratories in 2007, and has been researching on ubiquitous data management. He won the best paper award of IPSJ National Convention in 1999. He is a member of Information Processing Society of Japan (IPSJ) and Japan Society for Fuzzy Theory and Intelligent Informatics (SOFT).

Noriyuki Takahashi received the B.E. and M.E. degrees in information engineering from Kyoto University, in 1990 and 1992, respectively. He joined NTT laboratories in 1992. He has been in the NTT Network Innovation Labs since 2001. His current interests include routing mechanisms, adaptive networking systems, and architecture design of future networks. He is a member of IPSJ, ACM, and IEEE.