Embedding Legacy Keyword Search into Queries for the Ubiquitous ID Database

Tetsuo Kamina, Noboru Koshizuka, and Ken Sakamura

The University of Tokyo,
7-3-1, Hongo, Bunkyo-ku, Tokyo, 113-0033, Japan
{kamina,koshizuka,ken}@sakamura-lab.org

Abstract. Ubiquitous ID is a general purpose framework for implementing context-aware ubiquitous computing applications, where identifiers (called ucode numbers) and their relations are maintained in a large scale distributed database called UCRDB. Since we have to maintain a huge amount of data in UCRDB, it is sometimes desirable to delegate some subqueries to the "legacy" text-base search engines. In this paper, we propose a new query language construct for UCRDB (but can be applied to any other similar technologies such as RDF databases) that enables such a dynamic linking. Using this approach, context of the real world and legacy contents exist in the digital space can be seamlessly combined, and we can view UCRDB and legacy search engines as a single hybrid database so that no programming to "hard-wire" existing services is required. Furthermore, in our system, configurations of dynamic linking are described as a rule base stored in UCRDB itself, thus the resulting system is very simple but highly flexible and extensible.

1 Introduction

In a ubiquitous computing environment, it is important to enable computing devices to *identify* objects and places in the real world; one of the promising ways to satisfy this requirement is to assign an identifier to each object and place that we want to identify, and store this identifier into some digital formats (such as RFID, barcode, and so on) that computing devices can read. Furthermore, to realize the property of *context awareness*[2], we can define the relations among objects and places that describe the context of the real world by relating each identifier.

Ubiquitous ID[1] is a general purpose framework for implementing such context-aware ubiquitous computing applications. In this framework, each object and place is identified by a *ucode number*, a unique identifier that is actually a 128-bit length integer carrying no semantic information and thus it can be assigned to *everything*. Even logical entities such as relations of ucode numbers may be identified by ucode numbers. All the ucode numbers and their relations are stored in a large scale distributed database called UCRDB (UCode Relation database). These relations form a very large directed graph like RDF[15], where each node is a ucode number or a string literal (that is an attribute of the ucode number),

and each edge is a relation between a ucode number and another ucode number or a string literal. Queries to UCRDB is performed by adapting the technology of graph pattern matching. A query graph pattern is constructed by a mobile terminal acquiring ucode numbers from its surrounding environment and combining them with some contextual information (such as access histories); this pattern is then sent to the UCRDB and all the matched results are returned.

To implement many information services based on UCRDB, we have to prepare a huge amount of digital contents, which can easily be a very time-consuming task. On the other hand, there already exists a huge amount of information in World Wide Web, and its amount is explosively increasing. Therefore, to effectively reuse these contents, it is sometimes desirable to link UCRDB with services provided by the "legacy" text-base search engines (regardless to say that they serve documents on the Internet or a local document repository).

In this paper, we propose a new query language construct that enables such a dynamic linking. Even though this proposal is based on UCRDB, our approach can be applied to any similar database technologies such as RDF databases. Actually, our query language is implemented as a simple extension of SPARQL[16], a standard query language for RDF and thus details of the underlying implementation of database systems are out of scope of this paper. However, to clarify our motivations, in the rest of this paper we argue our approach based on UCRDB.

The proposed query language has a new feature of interfacing and querying external text-base search engines *on behalf of the query execution of UCRDB*. In general, the scope of a query language is closed inside its targeting database system, and its only way to communicate with external systems is using standard interfaces such as JDBC and ODBC, through which execution of query is *passively* invoked from general purpose host languages such as Java and C++. Since many query languages do not provide interfaces where they can *actively* communicate with the external systems, the aforementioned dynamic linking have had to be implemented in the level of host languages. Our approach is useful in that this dynamic linking is performed in the level of query languages.

In our approach, not only a linking between UCRDB and legacy contents but also which external search engines to be accessed is determined at run-time. To implement this feature, we use UCRDB itself as a key technology; using UCRDB, we define which external engines to be accessed and how to communicate with these engines as a rule base. Our query system understands this rule base and automatically delegates search request to the appropriate external engines. Thus, in our approach a query is very concise and simple but its behavior is flexibly configurable and extensible.

So far, our contributions can be summarized as follows:

- In our approach, context of the real world and legacy contents exist in the digital space are seamlessly combined so that application developers can easily make use of legacy contents in new context-aware ubiquitous information services.
- A simple query language enabling aforementioned dynamic linking is implemented as a simple extension of SPARQL.

- By using this query language, we can view UCRDB and legacy search engines as a single hybrid database and thus no programming at the level of host programing language is required.
- The resulting system is very simple but highly flexible and extensible.

2 Motivating Applications

2.1 Ubiquitous ID Framework

In this section, we introduce the core technologies of ubiquitous ID as a background. Ubiquitous ID is a general purpose framework that is designed to implement context-aware ubiquitous computing applications. To realize the property of context-awareness, in this framework every object, place, and even concept that we want to identify is assigned a unique identifier called a ucode number. A ucode number can be stored in many kinds of tags such as passive RFID tags, active RFID tags, infrared markers, barcodes, 2-D codes, and so on. How to embed ucode numbers to these tags and access protocols to these tags are standardized so that many kinds of computing devices can read ucode numbers attached to the real world.

The context of the real world is described by relating each ucode number. Each relation is also identified by using a ucode number, thus this relation is described as a triple of a subject ucode number, a relation ucode number (i.e. predicate), and an object ucode number, which is exactly the same format of RDF[15] triples except that each element of a triple is not an URI but a ucode number[1]. Actually, each ucode number can be represented as an URI by using a namespace; furthermore, in UCRDB we may assign an *alias name* (in the form of URI) for each ucode number. Therefore, we can query over UCRDB by using SPARQL, a standard query language for RDF databases[2].

2.2 Applications

Ubiquitous ID is a general purpose framework in that each ucode number does not carry any semantic information, and UCRDB is semi-structured so that we can freely define application specific schemata. Since we have to maintain a huge amount of data in UCRDB, when we implement an information service using UCRDB it is sometimes desirable to delegate some subqueries to the legacy text-base search engines. In the following subsections, we describe such scenarios.

---

[1] As in RDF, the object part of a triple may also be a string literal. Note that this string literal may be a URI string. For example, we may relate a ucode number with a URL that stores further information of the ucode number

[2] In our implementation, each alias name is reduced to the corresponding ucode number in the preprocessing phase of SPARQL.

Site-specific information systems. By using ucode numbers, we can identify places or sites. Furthermore, by adapting this identification technique, we can also define the relations among places such as "that is the 4th building past the intersection," "these two intersections are connected," and so on. By maintaining these relations in UCRDB, we can construct a pedestrian navigation system[4]; in this system, an active RFID tag announces the ucode number of a place, and the mobile terminal reads the ucode number and queries the route information to the UCRDB. This mechanism enables more fine-grained pedestrian navigations than that are built using GPS technologies. Another advantage of this approach is we can provide a pedestrian navigation for interior regions such as museums and shopping malls, where GPS technologies cannot be applied.

In a pedestrian navigation system, it is also convenient to link the navigation system with other information services; for example, some users may want to be notified the nearest shop information or sightseeing information from where they are. Since creating such contents from scratch is a very time-consuming task, such information should be retrieved from the Web search engines using the name of the place (and other auxiliary information) as keywords. Therefore, some linking mechanism between UCRDB and Web search engines will be useful.

Equipment management. Preserving digital archives of records of the equipments in the real world is a persistent requirement. By applying the Ubiquitous ID framework, we can also construct such an equipment management system. In this system, an RFID tag containing a ucode number is attached to each equipment, and we can get the records of equipments on site using a mobile terminal that reads the RFID tags and queries the database storing the information.

Such equipment management database can be constructed by relating each ucode number to their information. However, there are huge amount of records of equipments, thus constructing a database of such records is a very time-consuming task. On the other hand, to construct a digital archive, we may also use a full-text search engine. In this case, we do not have to construct a database; we just index each document stored in the file system so that each document is searched using keywords. To make use of such a full-text search engine from UCRDB, we have to implement some communication mechanisms between them.

3 Design and Implementation

In the aforementioned applications, we *conceptually* view the UCRDB and text-base search engines serving World Wide Web or file systems as a single hybrid database. We introduce a new feature of interfacing and querying external engines *on behalf of the query execution* into the SPARQL query language. In this extension, we impose the following requirements:

Extensibility: There are many kinds of text-base search engines. The number of search engines is still growing, and each search engine may evolve so that its interface changes. Thus, our extension should be able to incorporate with new kinds of external engines.

```
@prefix ex: ... .
ex:pointA ex:latitude "..." .
ex:pointA ex:longitude "..." .
ex:linkAB ex:node ex:pointA .
ex:linkAB ex:node ex:pointB .
ex:linkAB ex:length "34.5" .
ex:pointA ex:nearestShop ex:abcDepartmentStore .
ex:pointA ex:nearestShop ex:xyzMusic .
ex:abcDepartmentStore ex:name "ABC Department Store" .
ex:xyzMusic ex:name "XYZ Music" .
```

Fig. 1. A simple RDF data for pedestrian navigation system

Lightweight language: SPARQL is a very simple query language and thus adding unnecessary complexity is undesirable. Therefore, the new language constructs that are added to SPARQL should be very simple.

To achieve these requirements, our new extension of SPARQL has the following features:

Defining the mapping to external engines in UCRDB: In our extension, how queries are executed on each external engine is user-definable using UCRDB. To implement this feature, we store attributes of external engines those are written as UCRDB relations mapping each external server's ucode number to their attribute values. Each of these relations is assigned a ucode number and also given an alias name in the form of URI for human readability.

Developing the interpreter: We introduce a new pattern `SEARCH EXTERNAL` into SPARQL. A `SEARCH EXTERNAL` pattern interprets the relations mapping each external server to its attribute values and sends a query to the appropriate external engine. This pattern may contain variables appear in other triple patterns.

In the following sections, we assume that every ucode number appears in our example is assigned an alias name. Thus, in the following examples every triple of UCRDB appears in the form of an RDF triple. For simplicity, we also use a convention to write a URI by using a namespace prefix such as `ex:`.

An example. To explain the feature of our proposal, we use a simple example that describes UCRDB data of a pedestrian navigation system. Fig.1 shows a piece of the UCRDB database in the form of N3[3]. The values `ex:pointA` and `ex:pointB` are points in the pedestrian network that we want to identify to represent routes (e.g. intersections); these points are connected by the link `ex:linkAB` whose length is 34.5m. Besides the route for the goal, this pedestrian navigation system also shows some useful information around the point, if necessary. For example, the descriptions in Fig.1 show that `ex:pointA` is close to a department store and a music shop.

Consider that a pedestrian reads a ucode of `ex:pointA` using a mobile terminal. The following query acquires the name of shops close to the variable `?this` (whose value is implicitly given as `ex:pointA`) and sends a request to the external search engine by using the acquired shop name as a keyword:

```
SELECT ?y ?ext WHERE {
  ?this ex:nearestShop ?x .
  ?x ex:name ?y .
  SEARCH EXTERNAL ?ext { ?y -> extern:myServer } }       (1)
```

The `SEARCH EXTERNAL` pattern is the new syntax we add to SPARQL. This pattern may appear in any places where a SPARQL pattern (such as a triple, a `FILTER` pattern, and so on) may appear, but inside the block of this pattern has a special syntax; we may not write any SPARQL patterns there. Instead, we put a search request descriptor (just a variable `?y` in the above example) and a search engine descriptor (a resource `extern:myServer` in the above example), which means that the search request `?y` is sent to the resource `extern:myServer`. The response of this request (a list of pair of found resource's URL and title) is stored in the variable `?ext`[3]. As in the original SPARQL, the result of this query is a list of tuples whose columns are indexed as `?y` and `?ext`, respectively. A request is sent to `extern:myServer` for each matched value with `?y`[4].

We can use multiple variables in a search request descriptor by combining them by logical operators such as `&&` (and) and `||` (or). We can also restrict the search results by specifying filters such as the content's language (written `lang="jp"`, for example), maximal counts of the results (written `max=10`, for example), and so on. Note that filters can only be combined using the `&&` operator.

Behavior mapping in UCRDB. In the above example, how the query keywords are sent to the external engine is not specified. In our system, this mapping is not hard-wired, to support a dozen of external engines and to incorporate with a new kind of external engines that is not taken into account at the first time of development. For this purpose, we introduce a new UCRDB vocabulary (i.e., a set of ucode numbers assigned to relations specifying attributes of external engines). This vocabulary is used to map a resource identifying an external engine (`extern:myServer` in the above example) to its attributes such as its base URL, a HTTP request parameter name that captures query keywords, and so on.

For example, a specification of the external engine `extern:myServer` can be written as shown in Fig. 2. It shows that the base URL of the location of `extern:myServer` is `http://search.yahoo.co.jp/search`, and the HTTP request parameter name used for queries to this search engine is `p`. Thus, the query `(1)` results in the following sequence of HTTP requests, when the variable `?this` matches `ex:pointA`:

[3] Note that the variable `?ext` does not match any of subject, predicate, and object of triples, since it does not have a type of URI or string literal.

[4] However, we may restrict the total number of requests to the same external engine in one query, whose default value is 100.

```
extern:myServer exvocab:url "http://search.yahoo.ac.jp/search" .
extern:myServer exvocab:query_param "p" .
... (other specifications for HTTP request parameters)
```

Fig. 2. A specification of an external engine written in RDF

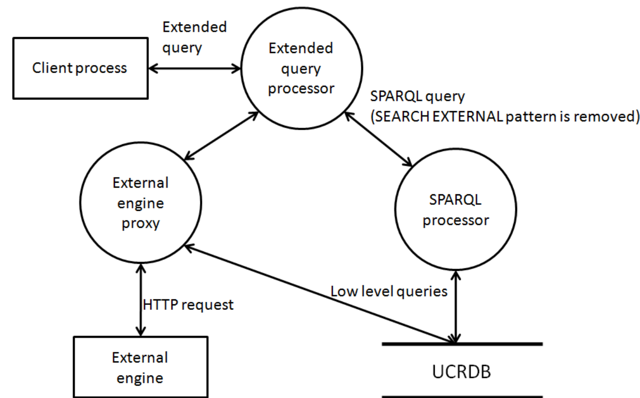

Fig. 3. Data flow of our query processor

```
http://search.yahoo.co.jp/search?p=%22ABC+Department+Store%22
http://search.yahoo.co.jp/search?p=%22XYZ+Music%22
```

We can also specify other external engines in the similar way. Furthermore, we can change the specification of `extern:myServer` to make use of the other external engine without modifying the query. Our query engine understands the UCRDB vocabularies and delegates queries for the appropriate external engine.

Note that the format of results from each external engine (in general written in HTML) also differs from other external engines. This means that how to extract the list of pairs of found document's URL and title from the results of each external engine is also differs from that of other external engines. Therefore, we also have to specify the rules for such extraction using UCRDB vocabularies. For example, which anchor tag contains URL of the query result can be determined by observing its `class` attributes' value, its enclosing tag and attributes, depth from the root `html` tag, and so on.

Implementation. A data flow of our query processor is shown in Fig.3. Firstly, the extended query processor parses the query and constructs a SPARQL statement from which `SEARCH EXTERNAL` patterns are removed. Then, it passes this statement to the original SPARQL processor. The SPARQL processor executes the SPARQL statement and returns the results to the extended query processor. By inspecting the results, the extended query processor then assigns values

```
String query = ...; /* SPARQL statement shown in section 3.1 */
DatabaseResult rs = ucode.execSparqlQuery(query);
while (rs.hasNext()) {
  ExternalResultList lst = (ExternalResultList)rs.get("?ext");
  while (lst.hasNext()) {
    ExternalResult ext = lst.next();
    String url = ext.getURL();
    String title = ext.getTitle(); ... } }
```

Fig. 4. An example code using our API

to the variables appear in `SEARCH EXTERNAL` patterns to construct a query for each external engine. This process iterates over all the combinations of variable assignments. The external engine proxy constructs HTTP request parameters for the external engine whose location is acquired by inspecting the UCRDB, and sends them to the external engine. The response from the external engine is also inspected by looking up the UCRDB rule base to extract the search results (i.e. found document's URLs and titles). Finally, the extended query processor merges the results from the original SPARQL processor and external engines, and returns them to the client.

Our current implementation is based on Java API for UCRDB. This API is object-oriented, in that every query is executed via message passing. A receiver of messages is a ucode object; besides simple lookup methods such as "getting all the triples whose subject is the receiver ucode," a SPARQL execution method is also implemented as a member of `Ucode` class. The method signature of `execSparqlQuery` is as follows:

```
DatabaseResult Ucode.execSparqlQuery(String query);
```

The formal parameter `query` is an (extended) SPARQL statement. This statement may include a special reserved variable `?this`, which refers to the ucode number of its receiver.

A result of SPARQL statement is a list of tuples, whose columns are indexed by the names of variables. The class `DatabaseResult` encapsulates the internal structure of tuples and provides access methods. In general, a value of each column has a type of ucode or string literal (or blank node), but some values have a type of "lists of pairs of found document's URL and title." Consider the code shown in Fig.4. In this code, the SPARQL statement `(1)` is executed, and its result is assigned to the variable `rs`. For each row of `rs`, we get a value of column indexed as `?ext`. Since this value is a result of `SEARCH EXTERNAL`, it contains a list of found document's URL and title. The types of `?ext` and its elements are represented as classes `ExternalResultList` and `ExternalResult`, respectively.

So far, the aforementioned requirements are met; owing to UCRDB's ability of self-description, we may set each parameter describing external engines in UCRDB itself, thus the resulting system is highly flexible and extensible.

The only language construct we add to SPARQL is `SEARCH EXTERNAL` pattern, whose semantics is also straightforward. The only burden we add to the underlying SPARQL implementations is the process time of external engine proxy, which includes response time of HTTP request that likely be a bottleneck. This response time varies depending on the external engines and network conditions, but in most of the cases it should be acceptable.

4 Related Work

Extensive research efforts have been made for exploiting how to combine classical search techniques with semantic model described as metadata or ontology[11, 7, 12, 6]. For example, Rocha et al.[11] show an approach of using semantic model of a given domain to calculate weights of links that measure the strength of the relation. Spread activation techniques are used to find related concepts in the ontology given an initial set of concepts and corresponding initial activation values, which are obtained from the results of classical search. In general, these approaches are useful when there is rich metadata associated with web pages.

Our approach, on the other hand, aims to combine classical search techniques with semantic search, regardless to say that there are metadata associated with web pages or not. In this sense, our approach is more similar with business process execution languages such as BPEL4WS[5], which is a result of merging previously developed WSFL[9] and XLANG[13]. In these approaches, however, the description of service partners is done via WSDL portType definitions[14], which prevents solely matching of WSDL messaging interfaces[5]. Furthermore, semantically the same services cannot be combined unless the import and export interfaces are exactly matched. Therefore, these approaches do not provide a flexible mechanism to combine each service as presented in our approach.

Mandell and McIlraith [10] propose a bottom-up approach to integrate Semantic Web technologies into Web services. Based on BPEL4WS, they present integrated Semantic Web technology for automating customized, dynamic binding of Web services together with interoperation through semantic translation. Our approach, on the other hand, is based on SPARQL and provides much closer looking at database queries. In our approach, combination of UCRDB and legacy search engines is performed in a declarative way.

5 Concluding Remarks

This paper presents a new query language construct that enables dynamic linking of UCRDB and legacy text-base search engines. This feature is designed and implemented at the top of UCRDB, but the same approach may be applicable to any other similar technologies such as RDF databases. Using this approach, context of the real world and legacy contents exist in the digital space can be

[5] Kamina and Tamai [8] propose a method of structural matching of WSDL messaging interfaces.

seamlessly combined, and we can view UCRDB and legacy search engines as a single hybrid database so that no programming to hard-wire existing services are required. Furthermore, in our system, configurations of dynamic linking are described as a rule base stored in UCRDB itself, thus the resulting system is very simple but highly flexible and extensible.

References

1. Ubiquitous ID Center. http://www.uidcenter.org/.
2. Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, volume 1707 of *LNCS*, pages 304–307, 1999.
3. Tim Berners-Lee. Notation 3. http://www.w3.org/DesignIssues/Notation3.html, 1998.
4. Masahiro Bessho, Shinsuke Kobayashi, Noboru Koshizuka, and Ken Sakamura. A space-identifying ubiquitous infrastructure and its application for tour-guiding service. In *SAC 2008*, pages 1616–1622, 2008.
5. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services. http://www.ibm.com/developerworks/library/ws-bpel/.
6. John Davies and Richard Weeks. QuizRDF: Search technology for the Semantic Web. In *Proceedings of the 37th Hawaii International Conference on System Sciences*, page 40112, 2004.
7. Li Ding, Tim Finin, Anupam Joshi, Rong Pan, R.S̃cott Cost, Yun Peng, Pavan Reddivari, Vishal Doshi, and Joel Sachs. Swoogle: A search and metadata engine for the Semantic Web. In *CIKM'04*, pages 652–659, 2004.
8. Tetsuo Kamina and Tetsuo Tamai. Loosely Connected RPC: An approach for extendable interface of Web Services. In *Proceedings of the 1st International Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMAI-2003)*, pages 62–73, 2003.
9. F. Leymann. Web services flow language. http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
10. Daniel J. Mandell and Sheila A. Mcllaith. Adapting BPEL4WS for the Semantic Web: The bottom-up approach to web service interoperation. In *The Semantic Web – ISWC 2003*, volume 2870 of *LNCS*, pages 227–241, 2003.
11. Cristiano Rocha, Daniel Schwabe, and Marcus Poggi de Aragao. A hybrid approach for searching in the semantic web. In *WWW 2004*, pages 374–383, 2004.
12. Ljiljana Stojanovic, Nenad Stojanovic, and Raphael Volz. Migrating data-intensive Web Sites into the Semantic Web. In *ACM SAC'02*, pages 1100–1107, 2002.
13. S. Thatte. XLANG: Web services for business process design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.html.
14. W3C. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.
15. W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/, 2004.
16. W3C. SPARQL Query Language for RDF. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, 2008.