RC24770 (W0903-092) March 24, 2009
Computer Science

IBM Research Report

EDS: Data Service Middleware for Situational Applications

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

EDS: Data Service Middleware for Situational Applications

Avraham Leff and James T. Rayfield

IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, N.Y. 10598
{avraham,jtray}@us.ibm.com

Abstract. Developers of *situational applications* – applications created by a small group of users/developers to satisfy the specific needs of the group – require two things from their software stack. First, they require support for their rapidly changing designs; second, they require semantics that are close to their own domain of expertise. For example, developers of situational applications prefer to use scripting languages because the "duck typing" style of programming language allows them to ignore issues related to class inheritance or interface implementation. In comparison to strongly-typed languages, developers can begin programming more quickly, and can modify their program more rapidly in response to changing requirements.
In this paper we explore whether middleware services can similarly provide developers of situational applications with these desired software characteristics. Specifically, we present EDS, an **E**xtensible **D**ata-**S**ervice designed to support applications whose design changes rapidly and with semantics that are closer to the domain expertise of situational application developers. We present the features of EDS, contrast it to other data services and APIs, and discuss the EDS implementation.

1 Introduction

1.1 Situational Applications

Situational applications have been defined as software that is "designed in and for a particular social situation or context" [1]. The biggest difference between situational and traditional applications stems from the fact that the requirements of situational applications are driven by the needs of "a specific social group", in contrast to the requirements of traditional applications which are driven by the needs of "a generic set of 'users' ". Traditional applications tend to evolve fairly slowly because the (perceived) needs of "generic users" evolve fairly slowly, and the risk-management needs of large organizations imply that a broad consensus be reached before changes are made. The planning and design of traditional applications is therefore more formal and takes longer than situational software.

In contrast, situational applications evolve as rapidly as the requirements of the well-known, very concrete, set of users change, and developers of situational applications place a great premium on flexibility (typically operating

outside the usual I/T bureaucracy). Situational developers favor a client-centric programming model (perhaps, because of the lower entry-barrier or because of the proportionally greater focus on an application's view). These developers are much less risk-averse, and can reach consensus for change much more rapidly.

As a result of such differences, developers of situational applications prefer to use a software stack with somewhat different characteristics than that used for traditional applications. For example, developers of situational applications prefer to use scripting languages [2] because the "duck typing" style of programming language allows them to ignore issues related to class inheritance or interface implementation. In comparison to statically-typed languages, developers can begin programming more quickly, and can modify their program more rapidly in response to changing requirements. We contend that these preferences apply further down the software stack as well, and thus raise important issues for the middleware used by situational applications.

1.2 Middleware For Situational Applications

Some situational applications – e.g., a spread-sheet application used to manage departmental resources – do not actually require middleware. More typically, the situational applications used in enterprises do use middleware, although to a lesser degree than traditional software. For example, a situational application to manage departmental-sized purchase orders might use a database to store its data, a directory service to look up users, and an authentication service to validate users. The situational application is constructed by layering a primitive work-flow on top of this middleware, adding some business logic specific to the department's policies, and creating some GUI forms. For two reasons, existing middleware APIs and semantics may not be a good fit for developers of situational applications.

Middleware services have been distilled, over time, by observing commonly repeated patterns in business applications. Middleware abstracts and concentrates this expert knowledge into a powerful and sophisticated API that addresses a wide range of use cases. Often, however, this creates a gap between the messy business reality and the uncluttered abstraction required by the middleware model. For example, to use a relational database effectively, data modelers must typically transform an enterprise's raw data into 3rd-normal form. Following this modeling, database tables are created with the appropriate schema, and the enterprise data is stored in, and accessed from, the correct representation. However, as discussed above, situational application developers place a premium on software that provides the flexibility for rapidly changing requirements. Requiring a data modeling phase before writing the application implies either that the application will never get written, or (more typically) that messy hacks are coded by the developer because they don't properly exploit the middleware. In addition, while these developers have the domain-specific knowledge needed to code the business logic, they do not necessarily have the expert knowledge needed to use the middleware's framework. Their needs correspond to a small subset of the

use-cases supported by the middleware, and they have difficulty understanding and using the "complicated" middleware.

The mismatch between situational application developers and traditional middleware can be seen clearly in the context of Clay Shirky's observation [1]:

> Situated software isn't a technological strategy so much as an attitude about closeness of fit between software and its group of users, and a refusal to embrace scale, generality or completeness as unqualified virtues.

Given that the strengths of middleware lie precisely in its "scale, generality, and completeness", it becomes obvious that situational application developers would prefer to use middleware that is a closer fit to their needs.

In Section 2 we take a closer look at the characteristics of data-service middleware that better fit the needs of situational applications. We also discuss how EDS relates to similar work in this area. In Section 3 we describe EDS in detail, based on a scenario that illustrates how a situational application might use EDS as a data service. The capabilities we describe (CRUD in Section 3.1, and support for relationships in Section 3.2) are fully supported in our current EDS implementation (Section 3.3). Finally, in Section 4 we point out certain limitations in EDS, and explain why those limitations may relate to intrinsic trade-offs in middleware for situational applications. The choices we made in the design of the EDS data-service middleware make it a good fit for developers of situational applications.

2 Situational Applications, Schema Constraints, and Data Services

A situational application developer prefers a minimal gap between her view of application data and the view required by the data service for:

- storing application data reliably (e.g., with "ACID" [3] properties).
- retrieving application data.

Constraints imposed by the data service before the application can store data are a hurdle to be overcome – even if the constraints are needed for a more powerful data model. In our view, *database schema* are the most important of these data service constraints. Depending on a the data service's persistence model, schema can constrain the developer to a greater or lesser extent.

Figure 1 shows a spectrum of database schema constraints, ranging from the least constrained (on the far left) to most constrained (on the far right). At the far left, the data service allows developers to store an array of bytes; no constraints are imposed on what the developer stores. This form of data service is typically provided by low-level operating-system drivers, or raw files, and is not usually considered to be a data service, because it provides no support for data retrieval. Clients of the data service are responsible for the task of organizing and interpreting the bytes.

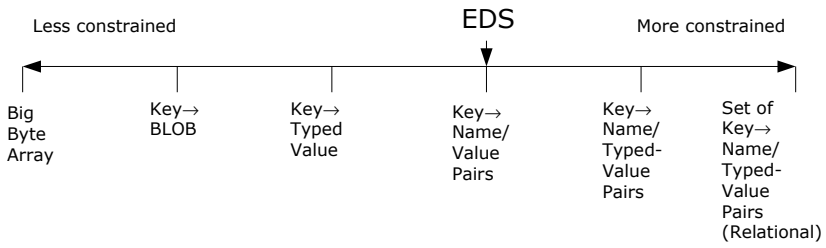

Fig. 1. Characterizing a Data-Service in Terms of Loose/Tight Schema Constraints

Moving to the right, the schema may constrain developers to associate a "key" with the raw data. This is more useful to the data service's clients because data retrieval can now easily specify the desired data by using one key rather than another. The application, however, is left to convert the data into its original "type" (whether integer, string, or more complex structure). More constrained schema require the developer to associate the data with a type that is also stored by the data service and used to convert the data when it's retrieved. The benefit of this is partially offset by the fact that developers must now decide – before they can use the data service – whether a given piece of data, for example, is "really" a string or a date. More constrained schema require developers to specify the structure of a datum by naming its properties (e.g., "phone number", "address"). This has the benefit of retrieving data in a form closer to the application level. Business logic rarely manipulates a single typed value: whether or not it manipulates "objects", an application usually groups a set of related properties (e.g., "employee", "purchase order") into a single structure. A data service can further constrain developers to also specify individual property types, rather than simply specifying property names. Finally, as in the case of the strict relational data model [4], developers are forbidden to insert duplicate data, and may be unable to insert data with property values that violate "primary-key" or "foreign-key" constraints.

In short, Figure 1 shows that the data modeling benefits (e.g., property name validation, type validation) provided by stronger schema constraints are offset by forcing the developer to do the data modeling before she can even use the data service. Equally important, every time that different sources of data are used by an application, and every time that business logic changes its use of the data, developers must ensure that the schema constraints have not been violated. It must be stressed that traditional applications are happy with a trade-off that lies towards the right side of this spectrum. It is only because situational applications require greater flexibility that developers evaluate this trade-off differently.

2.1 Related Work

We discuss related work with reference to the spectrum of database schema constraints shown in Figure 1. Hardware (e.g., memory and disk) and raw files provide the least amount of constraint, the "Big Byte Array". This level is primarily useful as a building block for the more constrained levels. The next level of organization includes models which are analogous to hash tables ("Key → BLOB" (Binary Large OBject)). Work in this space includes Project Voldemort [5], Dynamo [6], and Kai [7]. Similar projects that support only non-persistent data include Scalaris [8] and Memcached [9].

A number of projects support the multi-dimensional key → BLOB storage model used by Bigtable [10], including Cassandra [11], HBase [12], and Hypertable [13]. Thrudb [14] supports a "Key → Typed Value" model, where values can be strings, JSON [15] objects, XML, or Thrift [16] objects.

CouchDB [17] supports a "Key → Name/Value Pairs" model similar to EDS.

Finally, the highly-constrained relational data model is supported by products such as DB2 [18] and MySQL [19]. The relational model is a very powerful paradigm, originally based on set theory [20]. The relational model is capable of describing virtually all the other common data models, including Key → value, hierarchical, network, and many more complicated models. Relational databases are typically accessed using ODBC [21] or one of its language-specific variants (e.g., JDBC [22] for Java and PDO [23] for PHP). Numerous projects have emerged to layer object-oriented technologies on top of relational databases [24], although the approach used by most of these is mathematically dubious [4]. These include Enterprise JavaBeans (EJBs [25]) and frameworks such as Hibernate [26].

Although the relational model is very powerful, and underlies many large websites, situational developers (and many others) have found it too difficult to master the use of relational technology. Relational databases are typically accessed via SQL [27], which is used to create, retrieve, update, and delete records, as well as to define and modify database schema. The retrieve (query) subset of SQL is very powerful, and includes selection predicates, joins, unions, sorts, etc.

Unfortunately, the trade-off for the high functionality of SQL is complexity. For example, the IBM DB2 v9.5 SQL reference manuals consist of 2064 pages in two volumes. Since many applications do not require the power of the relational model, most situational developers use (or invent) a simpler data model which is farther to the left in our spectrum. Unfortunately, the tools and insights provided by the relational model are then not available when needed.

Each point on the spectrum of Figure 1 offers a particular trade-off to developers. Loose schema constraints let developers to "get on with their work" and evolve applications rapidly. This is achieved at the cost of having the data-service provide less support to the developer. For example, loose schema constraints may imply that the data-service cannot validate at runtime that the correct property name is being used, nor can the data-service supply a typing system to validate that only data with the correct type is being stored. Loose schema constraints may imply that the data-service cannot perform certain optimizations because it

has no clues as to how disk storage should be organized or which columns should be indexed. Our point here is that data-services with characteristics such as EDS are especially suited to developers of situational applications. Just enough support is provided for data with finer internal structure than a single property, without constraining developers to stick with a defined set of properties or to stick with a defined set of types.

As mentioned above, CouchDB [17] resembles EDS in the way that it supports a "Key → Name/Value Pairs" persistence model. However, there are some notable differences between EDS and CouchDB:

- CouchDB is a document-oriented database, with REST access to documents consisting of JSON objects. EDS uses JSON as a storage format, but provides a Java API.
- EDS is built on top of an existing relational API and database, while CouchDB works directly with the file system.
- EDS supports multiple tables (*scopes*), while CouchDB supports only one scope.
- EDS leverages the query capabilities of the underlying relational database, while CouchDB requires developers to supply JavaScript functions to implement query. We discuss this issue more thoroughly in Section 3.2.
- EDS supports JOIN operations via the underlying relational database, while CouchDB explicitly does not support JOIN.

2.2 REST APIs

A number of database projects (e.g., CouchDB) make a point of being REST [28][29] accessible. However, in our view REST is a mechanism for providing remote access to a datastore, especially across the Internet and through firewalls, but is orthogonal to fundamental issues in database API design. REST defines a transport layer (HTTP), and maps CRUD on top of the HTTP verbs, but leaves most of the semantics unspecified. Specifically, knowing that a database supports REST will not provide enough information to interact with it. It does not tell the developer what data types are supported, what queries are available, or even whether PUT or POST is used for insert (see the comments on [30]). In fact, a REST API can be used to access relational data [31].

Finally, the implementation of REST APIs is quickly followed by the emergence of client libraries which provide a programming API in the client's native language. REST architecture is thus independent of where a data-service lies on the loose-constraint/tight-constraint spectrum of Figure 1. It is also independent of how a data-service can be designed to support extensibility (e.g., how to add relationships without prior data-modeling).

3 EDS: Extensible Data Service

EDS is a middleware data-service that provides situational applications with an extensible persistence model. Developers don't define schema, so there is no

```
Tuple employeeFromJSON = Tuple.fromJSON(employeeJSON);

Tuple employeeFromMap = new Tuple
  ("employees", new HashMap(<String, Object> properties));

Tuple employeeFromProperties = new Tuple
  ("employees", Property ... properties);
```

Listing 1.1. Three APIs for Creating Tuples

friction in dealing with evolving schema. Instead, the EDS data-service persists data in the form of "key to name/value pairs" (see Figure 1) – and relieves developers from creating the key.

The EDS API defines two data-types: *tuples* and *result templates*. (Note that all of our code examples use EDS's Java API.) Tuples are a collection of name/-value pairs ("properties"). EDS uses the reserved *scope* property to isolate tuples in one scope from tuples in another scope. Clients PERSIST tuples to, and remove tuples from, EDS; EDS returns tuples to clients. Semantically, EDS tuples are interchangeable with JSON objects, and the tuple API therefore has a "tuple from JSON" factory method. Given an "employeeJSON" string whose value is:

```
{"scope": "employees", "age": 42, "name": "John Doe", "manager": false}
```

the code in Listing 1.1 creates a Tuple with integer, string, and boolean valued properties that will be associated with the EMPLOYEES scope. In this way, EDS provides native support for storing and retrieving JSON objects. Because of the interchangeability of JSON objects and Java Maps, tuples can also be created from Java Maps (using JSON encoding to store the Map values) and from one or more name/value pairs.

Result templates create tuples from other tuples, specifying that the created tuples should contain a subset of other tuples. Syntactically, a result template associates a scope property with an ordered collection of property names. These property names are easily represented as a JSON array of strings, and EDS therefore includes a "result template from JSON" API. Given a "selectName" string whose value is:

```
{"scope": "employees", ["name"]}
```

applying the result template created from "selectName" to the tuple created in Listing 1.1, returns this tuple:

```
{"scope": "employees", "name": "John Doe"}
```

To give a concrete description of EDS features, we offer a development scenario that illustrates the EDS API. This scenario exercises not only the familiar CRUD portion of the API, but also illustrates the more advanced features of the

```
1 // persist one or more tuples
  Tuple[] = EDS.persist(employee1, employee2);
```

Listing 1.2. Persisting Tuples

API such as the built-in support for maintaining and retrieving tuple relationships. Note that our current implementation (Section 3.3) supports every aspect of the scenario below.

In our scenario, a user/developer starts coding a small human-resources application (HRApp) to manage information about the users in her department. At this point, she certainly isn't interested in packaging behaviors with the data: all she wants to do is store a set of properties about each person. The set of properties is initially conceived to be: *name*, *phone*, *house number*, *street* and *department*.

3.1 Support For crud

Using one of the APIs shown in Listing 1.1 to create an *employee1* and *employee2* Tuples, the tuples are persisted as shown in Listing 1.2.

Note the following points:

1. The developer did not ask EDS to create a database.
2. The developer did not ask EDS to create an *Employees* table within that database.
3. The developer did not ask EDS to create "name", "phone", "address" or "department" column in the *Employees* database table.

Before persisting a tuple, EDS detects whether or not a database table and the corresponding table columns exist. If they don't, EDS will silently create them before persisting the tuple. Less obviously, the developer did not have to define EMPLOYEES identity, nor did she specify a primary key for the *Employees* table. Instead, EDS silently added a UUID property to the persisted tuple: the value of this property uniquely identifies the tuple, and can be used to trivially retrieve the tuple. The developer is free to leverage the existence of the UUID if she so desires, or ignore it.

Tuples are retrieved using the `esearch` (exact search) API which returns a set of tuples (the result set) and which has two input parameters:

1. a tuple with properties that are matched against the persisted tuples: tuples that match are added to the result set.
2. a result template that specifies which properties should be included in the result set tuples.

The semantics of ESEARCH are "search by example": return all tuples that match the example tuple, and format tuples in the result set as specified by the result template.

```
ResultTemplate resultTemplate =
  new ResultTemplate ("employees");
Tuple searchTuple =
  new Tuple("employees", employee.getProperty(Tuple._UUID));
Set<Tuple> retrievedTuples =
  EDS.esearch(searchTuple, resultTemplate);
```

Listing 1.3. Using a UUID to Retrieve a Tuple

```
departmentValue = form.get("department");
final ResultTemplate resultTemplate =
  new ResultTemplate ("employees");
final Tuple searchTuple = new Tuple
  ("employees", property("department", departmentValue));
final Set<Tuple> retrievedTuples =
  EDS.esearch(searchTuple, resultTemplate);
```

Listing 1.4. Retrieving all Tuples in a Department

```
newDepartment = form.get("new department");
employeeName = form.get("name");
// retrieve employee that matches "employeeName"
employee = EDS.esearch(...).iterator().next();
employee.add(property("department", newDepartment));
EDS.persist(employee);
```

Listing 1.5. Updating an Employee

By saving a reference to the "employee", HRApp can use its UUID to retrieve the employee later (Listing 1.3). Here, the code asks EDS to return all employees tuples whose UUID property matches the property of the previously persisted employee. Since only one tuple can have that UUID, the result set will contain exactly one tuple. Because the developer did not explicitly specify result template properties, EDS returns values for all properties in the EMPLOYEES scope.

The developer now adds a GUI through which employees can be retrieved. The GUI form collects search criteria from users, and then uses the criteria to retrieve tuples from EDS. Because other departments are expressing interest in HRApp, she decides to implement "search by department" (Listing 1.4). Here, the result set will contain all Tuples in the specified department.

What started as a "store-and-retrieve" application is now getting used more broadly, and the developer is therefore asked to accommodate the fact that the department has experienced a "reorg". She adds a GUI with input fields for the employee name and new department, and writes new code to update the specified employee (Listing 1.5).

Note the following points:

```
String department = form.get("department");
String manager = form.get("manager");
Set<Tuple> employeesInDepartment =
  EDS.esearch(...);
for (Tuple employee : employeesInDepartment) {
  employee.add(property("manager", manager));
  EDS.persist(employee);
}
```

Listing 1.6. Add a Manager Property

```
Tuple layedOffEmployee = EDS.esearch(...);
EDS.remove(layedOffEmployee);
```

Listing 1.7. Removing an Employee

1. ESEARCH returns a Set of tuples, so we use an iterator to get the first tuple.
2. The PERSIST API is used both for inserting new tuples and for updating existing tuples. EDS uses the (non-)existence of the UUID property to determine whether the properties overwrite the tuple uniquely identified by the UUID or whether to insert a new tuple.
3. The *add* API is used both to add new properties to a tuple and to update the values of existing tuples.

As HRApp becomes more visible in the company, people begin to notice that employees are not associated with their managers. That situation being untenable, our developer adds an input field to the GUI, and adds the "manager" property using the code in Listing 1.6. Here again, EDS supports the situational application since schema changes are made implicitly rather than explicitly. Similar flexibility is shown with respect to changing the type of the *house number* property value. Originally, the example set that drove the requirements only contained entries such as "23 Maple Lane". Now, HRApp must deal with input such as "39B Leibzig Ave", and the developer realizes that *house number* is string-valued, not integer-valued. The EDS-related code, however, does not change despite the data-modeling change. The Tuple API is type-agnostic: although the EDS runtime will detect that the property type has changed, this information will be silently maintained in the JSON representation.

To complete the canonical set of CRUD operations that a data-service should support, Listing 1.7 shows how a tuple is removed. As with the ESEARCH API, the REMOVE API is applied to all tuples that match the input tuple parameter. Since a tuple trivially matches itself, passing it to REMOVE, removes it from the data-service.

```
janeTuple = new Tuple("employees", ... );
homePhoneTuple = new Tuple("phones", ...);
cellPhoneTuple = new Tuple("phones", ...);
EDS.persist(janeTuple, homePhoneTuple, cellPhoneTuple);
```

Listing 1.8. Creating a Relationship Between an Employee and her Phones

3.2 Support For Tuple Relationships

An important EDS feature is its support for creating and retrieving relationships between data items in a way that is consistent with development of situational applications. To motivate the usefulness of this approach, consider the structure of the "employee" tuple used in HRApp. The tuple groups related properties such as *name*, *address*, and *department*. It seems natural – and we'd actually expect – that our developer would initially include the *phone* property in the "employee" tuple. But, what should the developer do when a new requirement emerges to support *multiple* phone numbers: e.g., "work", "cell", "fax", "home"? A schema-less persistence model makes it very easy for the developer to simply add a separate property for each new type of phone.

However, as shown by Codd [20], there are many advantages to "normalizing" the EMPLOYEES and PHONES information. The most intuitive reason for normalizing data relates to the DRY (Don't Repeat Yourself) principle [32]: conglomerating EMPLOYEES and PHONES information will result in duplicate information. This should be avoided because it makes the current system harder to understand, makes it harder to change the system, and increases the possibility that the system will be inconsistent. The problem becomes even more serious if the phone tuple includes information such as an office location, making it difficult to update HRApp when an employee moves to a new office.

Data modelers therefore recommend that data such as EMPLOYEES and PHONES be normalized, and that relationships between employees and their phones be specified separately. EJBs, for example, identify *seven* types of relationships between entities, depending on the relationship cardinality and its direction. EDS recognizes that this type of explicit relationship modeling is unsuited for situational programmers since it assumes that the developer took the time to realize that employees may have multiple phones or that phone numbers may be assigned to offices. Other schema-less data-services (e.g., CouchDB) specify that the data-service does not provide support for relationships in the base service. Instead, developers are responsible for defining relationships at a higher level than the base data-service. In CouchDB this is done with server-side JavaScript functions, called "views" which CouchDB executes as data are inserted in order to keep the relationships consistent.

Situational application developers *do* prefer an API with semantics that are closer to their application view, and therefore avoid formal approaches such as EJBs. In our view, these developers nevertheless prefer to delegate as much function as possible to the middleware. While the existence of data relationships

is a business modeling decision, middleware ideally should assume the task of implementing the relationship. In the EDS approach, therefore, developers do not supply the relationship management code. Instead (see Listing 1.8), EDS clients may implicitly specify that relationships exist between tuples from different scopes as part of the PERSIST operation. The code in Listing 1.8 corresponds to Figure 2; it shows that "Jane Galt" has two phones – her cell-phone and her home-phone.

EDS uses a link-table approach in which relationships between two scopes are persisted in a separate database table. Link tables store UUID pairs where each UUID in a given pair are the two ends of a single relationship. (Recall that EDS uses a UUID as the primary key of the base scope table.) As discussed before (Listing 1.5), EDS uses the same PERSIST API both for creating new, and updating existing, tuples. This is true with respect to relationships as well. When EDS detects that the tuples being persisted are from two scopes, it checks whether a relationship between them currently exists. If not, EDS will create a new relationship.

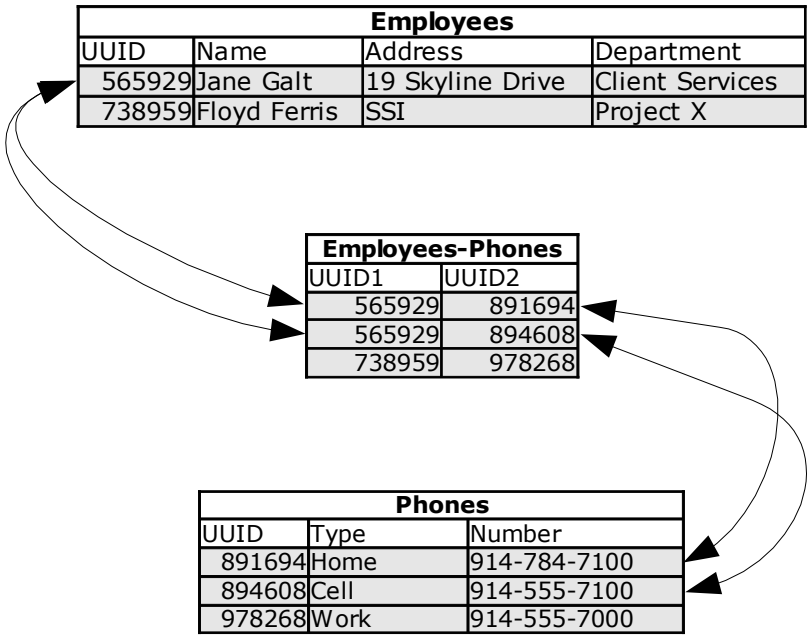

Fig. 2. Maintaining Relationships Between Employee and Phone Scopes

```
// result set will contain two List elements: each List
// contains the ``Jane'' employee tuple followed by one of
// her phones
final Set<List<Tuple>> janeResults =
  EDS.esearch
    (Arrays.asList(janeTuple),
        Arrays.asList
        (new ResultTemplate("employees"), new ResultTemplate(
            "phones")));
```

Listing 1.9. Use an Employee to Retrieve Her Phones

```
final Set<List<Tuple>> departmentPhones =
  EDS.esearch
     (Arrays.asList
        (new Tuple("employees",
                   Tuple.property("department", "Client
                      Services"))),
      Arrays.asList(new ResultTemplate(phones)));
```

Listing 1.10. Retrieve All Phones in a Department

EDS allows clients to retrieve related tuples across two scopes using an extension of the ESEARCH API.

As shown in Listing 1.9, the developer specifies that EDS should search for all tuples that match the "Jane" tuple, and *also* asks that the result set should contain all EMPLOYEES properties as well as all PHONES properties. EDS interprets this as a request to perform a JOIN over the two scopes, and using the information stored in the link-table, returns the employee and her phones. The result of the ESEARCH is a Set containing two Lists, each of which is comprised of an EMPLOYEES tuple (the "Jane Galt" tuple) and a PHONES tuple (either the "home" or "cell" tuple).

Listing 1.10 shows how the API is used to return all phones in the "Client Services" department. Here EDS looks only for EMPLOYEES tuples whose *department* property matches the specified value. Unlike Listing 1.9, the client asks that EDS only include PHONES tuples in the result set. Because the ESEARCH specifies two scopes (an EMPLOYEES tuple in the search constraint, and a PHONES tuple in the result template), EDS silently performs a JOIN to retrieve *related* tuples in the two scopes. The result of the ESEARCH is a Set containing two Lists, each of which contains a single PHONES tuple: one tuple is Jane's "home phone", the other is Jane's "cell phone". (Recall that Sets are unordered, so the order of the tuples is not specified by the API.)

The EDS relationship model is an undirected graph, whose nodes are tuples and in which two nodes are connected iff a relationship exists between the two tuples. The API thus does not require developers to distinguish between many

```
final Set<List<Tuple>> janeEmployee =
  EDS.esearch(Arrays.asList
              (new Tuple("phones",
                         Tuple.property
                            ("phone", "914-784-7100"))),
              Arrays.asList(new ResultTemplate("employees")))
                  ;
```

Listing 1.11. Using a Phone Number to Retrieve an Employee

different types of relationships in order to create or retrieve relationships. Listing 1.11 illustrates how EDS's relationship model allows developers to ignore issues related to both relationship direction and relationship cardinality. Here, the developers does a "reverse lookup" of EMPLOYEES using one of her phone numbers. This ESEARCH is the mirror-image of the previous one: the client specifies that a PHONES tuple be used to as the search criterion, but that the result set contain only the related EMPLOYEES tuples. The single List of the ESEARCH result set contains the "Jane Galt" EMPLOYEES tuple since she is the employee who is associated with the specified phone number.

3.3 Implementation

We have implemented EDS in Java, delegating responsibility for persistence to a relational database (Apache Derby [33]). The persistence model that EDS presents to clients thus has much looser schema constraints than the one EDS uses internally. EDS itself is standard middleware – *not* a situational application – and therefore benefits from the more rigorous constraints of the relational data model. Internally, all EDS database tables use a UUID column as the primary key: as described in Section 3.1, this makes it possible for EDS to use the identical API for both creating and updating tuples. As described in Section 3.2, the UUID approach allows EDS to get good performance on queries by benefiting from the superior indexing capabilities of relational databases. All database columns used internally by EDS are of type VARCHAR: incoming property values are encoded as JSON strings, and outgoing properties are decoded before returning them to clients.

EDS uses the JDBC [22] meta-data API to determine whether the tables and columns that correspond to the client's scopes and property names exist. Tables and columns are created on-demand using the JDBC API to execute DDL statements such as CREATE TABLE and ALTER TABLE. As clients specify a relationship between tuples in different scopes, link-tables are created with the structure shown in Figure 2. When a client performs an ESEARCH on two scopes, EDS retrieves related tuples by doing a three-way JOIN between the specified two scopes and the associated link table.

Our current EDS implementation supports all of the scenarios discussed above, including the CRUD API and retrieval of related tuples. At this point, we provide only a Java API and do not yet provide a REST API to web-clients.

4 Analysis

We have focused in this paper on the programming API of EDS, and deliberately ignored REST issues, for reasons discussed in 2.2. Since the programming APIs are compatible with JSON (the Tuple and ResultTemplate Java objects are directly serializable to and from JSON), it would be straight-forward to provide a REST transport and client library for EDS. This would enable this use of EDS in a Web 2.0 (business logic on the client) environment [34].

In order to simplify the programming API and semantics, EDS provides less functionality than a relational database. For example, since the schema is unconstrained, there is no way to detect incorrect property names or types on PERSIST operations. Also, the types of queries available in EDS are very limited: only exact matches on subsets of properties are supported. We have not (yet) conceived of a more general-purpose query mechanism that does not head down the slippery slope of creating our own query language (or adopting an existing one). Since many of these complex query languages have been rejected by situational developers, this does not seem like a promising direction. For now, more complicated queries must be implemented in application space.

EDS usefully supports situational application development in two ways. First, because of the close fit between the dynamically typed languages commonly used in such development and the dynamically typed data-service provided by EDS. Scopes are transparently persisted for the client; properties are transparently persisted or removed for the client; and changes in a property's type are transparently provided to the client. Second, EDS infers relationships between tuples when they are persisted together. Queries which span two scopes use the relationship information from the PERSIST operations to do a JOIN across the scopes.

We think that implementing EDS on top of a relational database provides important advantages. First, the experimental aspects of EDS involve the API and semantics, not the database implementation. Rewriting the database manager, logging, transaction support, utilities, etc., will not teach us anything further about the usefulness of the API and semantics, although they could improve performance. In addition, the relational model provides a very strong base for the implementation of EDS.

5 Future Work

Currently we are building an implementation of an EDS API on top of an LDAP [35] API. This presents some interesting challenges: for instance, LDAP properties may have multiple values. Also, LDAP is tree structured, rather than table structured (or multi-table structured). Addressing these issues should provide interesting insights into the design of the EDS API.

In addition, we are working with situational developers to validate our intuition regarding the usefulness of the EDS API. An open question is what kind of benchmark is appropriate to assess this question.

References

1. Clay Shirky. Situated software. http://www.shirky.com/writings/situated_software.html, 2004.
2. J.K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3):23–30, Mar 1998.
3. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, USA, 1993.
4. C.J. Date and H. Darwen. *Databases, Types and the Relational Model (3rd Edition)*. Addison-Wesley, Boston, MA, 2006.
5. Project voldemort. http://project-voldemort.com/, 2009.
6. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, 2007.
7. kai. http://sourceforge.net/projects/kai/, 2009.
8. scalaris. http://code.google.com/p/scalaris/, 2009.
9. memcached. http://www.danga.com/memcached/, 2009.
10. Bigtable: A distributed storage system for structured data. http://labs.google.com/papers/bigtable.html, 2009.
11. Cassandra: A structured storage system on a p2p network. http://code.google.com/p/the-cassandra-project/, 2009.
12. Hbase. http://hadoop.apache.org/hbase/, 2009.
13. Hypertable. http://hypertable.org/, 2009.
14. Thrudb. http://code.google.com/p/thrudb/, 2009.
15. Json in javascript. http://json.org/js.html, 2007.
16. Thrift. http://incubator.apache.org/thrift/, 2009.
17. Couchdb. http://couchdb.apache.org/, 2009.
18. Db2 product family. http://www-01.ibm.com/software/data/db2/, 2009.
19. Mysql. http://www.mysql.com/, 2009.
20. E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
21. Kyle Geiger. *Inside ODBC*. Microsoft Press, 1995.
22. Maydene Fisher, Jon Ellis, and Jonathan Bruce. *JDBC API Tutorial and Reference*. Prentice Hall PTR, 3rd edition, 2003.
23. Php data objects. http://www.php.net/pdo, 2009.
24. Object-relational mapping. http://en.wikipedia.org/wiki/Object-relational_mapping, 2009.
25. Enterprise Javabeans Technology. http://java.sun.com/products/ejb/, 2009.
26. Relational Persistence for Java and .NET. http://www.hibernate.org/, 2007.
27. C. J. Date and Hugh Darwen. *A Guide to SQL Standard*. Addison-Wesley, 4rth edition, 1996. ISBN: 0201964260.
28. Roy Fielding. Representational state transfer (rest). http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 2007.

29. Wikipedia. Representational state transfer. http://en.wikipedia.org/w/index.php?title=Representational_State_Transfer&oldid=109299419, 2007.
30. Joe Gregorio. How to create a rest protocol. http://www.xml.com/pub/a/2004/12/01/restful-web.html, 2004.
31. Avraham Leff and James T. Rayfield. Zazen: A mediating soa between ajax applications and enterprise data. *Services Computing, IEEE International Conference on*, 1:85–92, 2008.
32. Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
33. Apache derby. db.apache.org/derby, 2009.
34. A. Leff and J. T. Rayfield. Issues and approaches for web 2.0 client access to enterprise data. *accepted for publication, Advances in Computers*, 2009.
35. Lightweight directory access protocol. http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol, 2009.