A Database-Based Service for Handling Logical Variable Bindings

Wolfgang May
Institut für Informatik, Universität Göttingen, Germany
may@informatik.uni-goettingen.de

Abstract: The paper discusses the use of a *database as a service* that supports applications whose data flow is based on sets of tuples of variable bindings, as e.g., in logic-based frameworks like Datalog and similar rule-based approaches. The service also supports cooperative work of different services on the bindings, thus replacing the need for actual exchange of potentially large sets of data during workflows. The concrete usage scenario of the service is in a distributed environment for processing ECA (Event-Condition-Action) rules and processes over relational states.

1 Introduction

Several applications and languages use logic programming style variables for their data flow. The actual state of such an application is represented as a set of tuples of variable bindings, i.e., a (potentially large) relation. The common functionality for manipulating this information can be seen as an *abstract datatype* that provides constructors and the usual operations of the relational algebra. As applications can handle large quantities of data, a realization of the datatype as a *database service* is appropriate. Such a service can not only handle bindings for being processed by an individual application, but also serve as a central storage that replaces *data exchange* between applications.

The functionality of the abstract datatype for logical variable bindings does not only consist of the manipulation/combination of the actual tuples (in terms of applying SQL commands), but also has to consider the format of the relations to translate the required operations into SQL commands. For instance, the join of two sets of tuples is not only an SQL query over two relations, but first requires to go through the variable lists of both, and identify the join variables to *construct* the appropriate SQL statement.

Structure of the paper. We first give a formal definition of the concept of *variable bindings* and discuss possible implementation alternatives as a starting point. Section 3 introduces the actual usage scenario in a distributed Web Service environment. Section 4 discusses the design and usage of the service. Some specific SQL issues are picked up in Section 5; a short conclusion follows.

2 Variable Bindings – the Concept

Sets of tuples of variable bindings are a common concept in logic-based approaches, like Datalog. Consider a 5-ary predicate country/5 that holds for the name, the country code, the area, population, and name of the capital of countries, e.g., for Germany country("Germany", "D", 356910,83536115, "Berlin"). Then, the simple Datalog query

```
?- country(N, C, A, P, Cap).
```

binds the variables N, C, A, P, and Cap and results in a set of tuples of *answer bindings* (one tuple for each country)

$\{\{$N/"Germany", C/"D", A/356910, P/"83526115", Cap/"Berlin" $\}$
$\{$N/"France", C/"F", A/547030, P/"58317450", Cap/"Paris" $\}, \ldots\}$

which can be interpreted as a relation with attributes N, C, A, P, and Cap. Consider another predicate city/3 that holds for a city name, the country code where it is located, and its population, e.g., city("Berlin", "D", 3472009). Then, the evaluation of conjunctive queries, like

```
?- country(N, C, A, P, Cap), city(CityN, C, CityP).
```

consists of a *join* of two such relations, where the common variable C acts as join variable. As known from Datalog, (safe) negation is implemented by set difference (together with a projection), and built-in predicates are implemented by selections. Furthermore, grouping, ordering (and top-k), and some other operators are common.

Definition 1 *A set of tuples of variable bindings consists of (i) a format* $\{\mathit{Var}_1 : D_1, \ldots, \mathit{Var}_n : D_n\}$ *(the variable names* Var_i *with datatypes* D_i*), and (ii) a set of tuples* $\tau_1, \ldots, \tau_m$ *where each* τ_j *is a (possibly partial) mapping* $\tau_j : (\mathit{Var}_1, \ldots, \mathit{Var}_n) \rightarrow (D_1 \cup \{null\}) \times \ldots \times (D_n \cup \{null\})$*.*
A set of tuples can thus be seen as a relation R *with attribute names* $\mathit{Var}_1, \ldots, \mathit{Var}_n$*.*

The size of variable bindings needed in a workflow vary from a single tuple over a small number of variables up to potentially millions of tuples, although usually still over a relatively small number of variables (often even less than 10). Nevertheless, the concept should be able to support also bindings of a large number of variables.

So far, there are several straightforward implementation alternatives:

1. Provide and import an appropriate Java class VariableBindings that uses an in-memory data structure. This choice is sufficient for small amounts of data.
2. Another possibility is that the VariableBindings class uses a local database via JDBC.
3. The third possibility is to use a (remote) Web Service that provides the interface of the abstract datatype and uses *its* own database. The VariableBindings class is then only a stub that forwards its methods to the Web Service.

While (1) is actually restricted to small amounts of data, (2) has the disadvantage that it would require each service that uses the class to be equipped with a local database installation. A disadvantage of (3) is that even for small amounts of data (extreme, but frequent case: only one tuple is propagated through the application) the database service is contacted.

Further drawbacks of (3) are less obvious, but showed up when considering what the actual implementation would look like. As described above, apart from the actual relational operations that can be done in SQL, the abstract datatype has some functionality for maintaining and manipulating the format, i.e., the columns/attributes list and the datatypes, and constructing the SQL queries as strings, which requires actual programming and persistent storage of the metadata. Realizing this inside the database by PL/SQL is cumbersome. Alternatively it can be implemented in the Java layer that implements the service interface and then communicates by JDBC with the actual database. The latter alternative already suggests to consider to move this functionality away from the Web Service, into the stub VariableBindings class that is directly imported in the client services – and communicate from there with the database via JDBC.

The communication interfaces of Web Services are usually based on XML. The variable bindings can thus either be communicated in XML markup (that was already defined for the given use case), or an instance of the VariableBindings Java class is exchanged via SOAP. As the latter is inefficient, only the plain XML choice remains. XML data is untyped, so when exchanging variable bindings in explicit XML format, a datatype conversion has to take place twice. Thus, it is preferable not to exchange the actual variable bindings at all, but just to exchange a reference to a table where they are stored in a database.

Next, we discuss the environment that lead to the development of the service, its specific requirements, and its actual design decisions.

3 MARS

The service has been developed for use in the *Modular Active Rules in the Semantic Web (MARS)* framework. MARS is an open infrastructure for ECA rules and processes that involve multiple services and languages. For each component of a rule or a process, the rule designer can choose amongst an (open) set of appropriate languages. For instance, the event specification can be a composite event expression in the Snoop event algebra [CKAK94], which in turn embeds specifications of the relevant atomic events in –again– a different language; and the action component can be expressed in the *Calculus of Communicating Processes (CCS)* Process Algebra [Mil83]. The markup of rules is in XML, with the components embedded as XML subtrees. An ECA rule engine has been implemented [BFMS06]; every component language is also implemented by some processor as a Web Service. The communication between the Web services for executing rules is done by HTTP messages that contain the fragment to be executed and the variable bindings. The HTTP URIs and other metadata for finding an addressing processors for given language fragments are managed by *Language and Service Registries (LSRs)*.

Figure 1 (from [MAA05]) illustrates the structure of the rules and the corresponding types of languages.

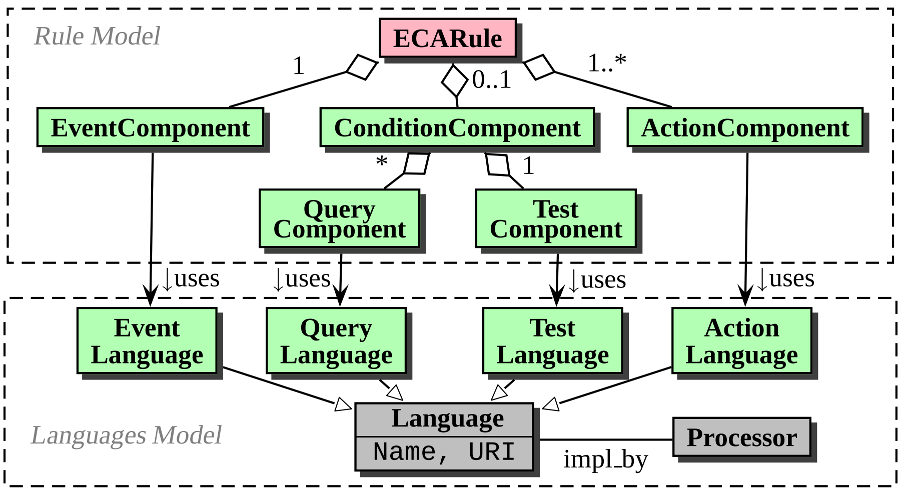


Figure 1: ECA Rule Components and Corresponding Languages (from [MAA05])

Variable Bindings in MARS ECA Rules. Coming back to the issue of the current paper, the data flow is managed in terms of logical variables in the style of production rules. The state of evaluation consists of a set of tuples of variable bindings (cf. Figure 2):

$$\begin{aligned}&\mathit{action}(X_1,\ldots,X_n,\ldots,X_k) \leftarrow\\ &\qquad \mathit{event}(X_1,\ldots,X_n),\ \mathit{query}(X_1,\ldots,X_n,\ldots,X_k),\ \mathit{test}(X_1,\ldots,X_n,\ldots,X_k)\ .\end{aligned}$$

The evaluation of the event component (i.e., the detection of a (composite) event) results in a set of tuples of variable bindings that is then extended in the query component, possibly constrained in the test component, and propagated to the action component. The SNOOP and CCS languages have also been extended with internal relational data flow [BFMS08].

Example 1 (ECA Rules and Variable Bindings) *Consider a simple rule like (the syntax is not the full MARS one, but just for sketching the main ideas)*

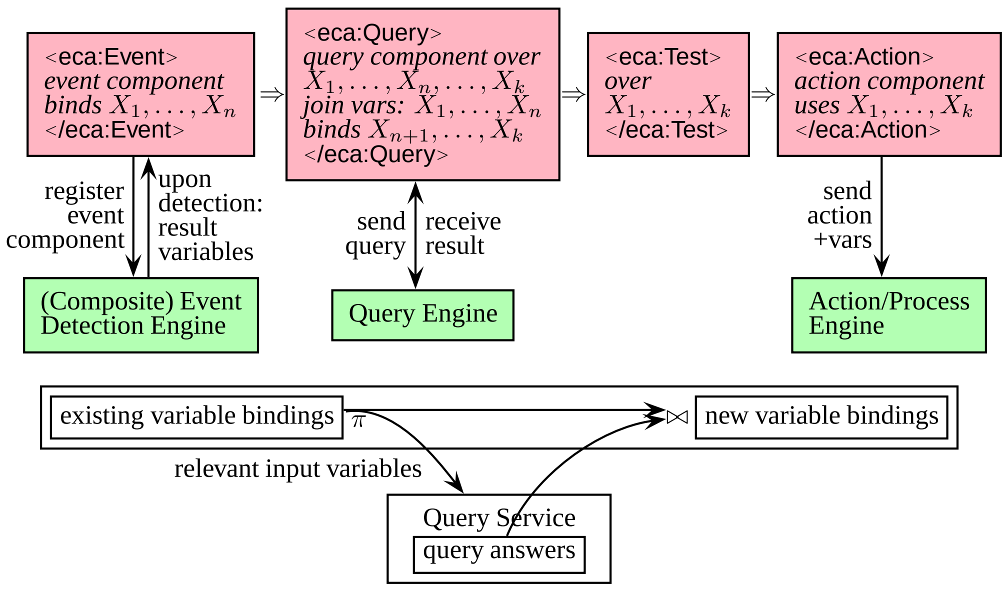
ON <travel:FlightDelayed travel:flight="F" travel:minutes="X"/>
WHERE F = "LH123"
DO send_sms(0049-0815,"your flight is X minutes delayed")

When a matching event <travel:FlightDelayed travel:flight="LH123" travel:minutes="30"/> occurs, one tuple of variable bindings is generated by the event detection service, i.e., $(F/$"LH123"$, X/30)$. It is propagated through the rule engine, the condition evaluates to true, and it is forwarded to the action processor that sends the SMS.

Another rule,

ON <travel:FlightCanceled travel:flight="F"/>
WHERE nearbyHotel(H) and price(H,P)
DO *book the hotel H that has the lowest price*

will again bind a single tuple in the event part, and extends this to about 5-10 tuples over variables F (that has the same value for all tuples), H and P in the query part.



<eca:Event> event component binds $X_1, \ldots, X_n$ </eca:Event>
<eca:Query> query component over $X_1, \ldots, X_n, \ldots, X_k$ join vars: $X_1, \ldots, X_n$ binds $X_{n+1}, \ldots, X_k$ </eca:Query>
<eca:Test> over $X_1, \ldots, X_k$ </eca:Test>
<eca:Action> action component uses $X_1, \ldots, X_k$ </eca:Action>
register event component
upon detection: result variables
send query
receive result
send action +vars
(Composite) Event Detection Engine
Query Engine
Action/Process Engine
existing variable bindings
π
$\bowtie$
new variable bindings
relevant input variables
Query Service
query answers

Figure 2: Use of Variables in an ECA Rule

In the above example cases, there is no need for a database service, but an in-memory Java data structure is sufficient, and even faster than the communication overhead by JDBC with a database. Only if the queries (there can be a sequence of queries) return many tuples, the use of a database pays. The only operators on variable bindings needed so far are joins, selection (for the test) and projection (for projecting on the required input tuples for the individual subqueries).

The prototype of the ECA engine has been developed in [BFMS06] with a Java VariableBindings class that implements the common functionality for handling sets of tuples of variable bindings. The communication between the services, i.e., back from the event detection service to the ECA engine, then to the query engines and back, and then to the action processor is realized in XML markup of the variable bindings according to the following DTD:

```
<!ELEMENT variable-bindings (tuple*)>
<!ELEMENT tuple (variable*)>
<!ELEMENT variable ANY>
   <!ATTLIST variable name CDATA #REQUIRED>
```

The content of the variable element is either text-only (for strings and numbers) or can be an XML fragment.

Variable Bindings in MARS CCS Processes. For expressing the action part of ECA rules, the *Calculus of Communicating Processes (CCS)* Process Algebra [Mil83] has been extended with relational data flow, i.e., sets of tuples of variable bindings are propagated through the process. As a language of the MARS framework, CCS processes can embed subprocesses from arbitrary languages.

Here, the operations on such sets of tuples are more involved than in the relatively simple linear data flow of ECA rules: CCS provides alternatives (when one branch executes for some tuples, these tuples have to be removed from the other branches), parallel execution with join (wait until all branches come back, where the subprocesses can even be asynchronous and partial, i.e., some tuples proceed faster than others due to external queries), and safe negation (tuples that match bindings returned by a subprocess are removed).

Furthermore, standalone CCS processes (i.e., not only as action part of ECA rules) showed to be useful for specifiying *query workflows* against (Deep) Web sources, which turned out to develop into a separate subproject. Here, the embedded subprocesses are atomic subprocesses in form of queries in the *Deep Web Query Language (DWQL)* that acts as a wrapper for the tool described in [HSL08]. In this case, the tuples of (input) variable bindings are communicated to the DWQL service, and the result bindings are communicated back to the CCS service. Here, queries return large numbers of tuples (e.g., for a graph exploration workflow that computes *Erdös numbers*, i.e., minimal coauthoring distance, based on DBLP), so the need for development of a database-based handling of variable bindings became apparent.

4 Design of the Service and its Usage

4.1 Requirements

The above use cases provided the base for analyzing the requirements in this setting where distributed, autonomous services cooperate based on the variable bindings:

1. All types of language services (event detection, process algebra processors, query engines, ECA rule engines) must have access to the functionality,
2. The database(s) should not be local to the processors, but the data flow *between* the processors should be *replaced* by cooperation via the envisaged autonomous *Variable Bindings Service (VBS)*: requests and answers then do not include the actual variable bindings in XML markup (whose serialization and de-serialization is time-consuming), but just an identifier how the variable bindings can be accessed in the VBS.
3. Considering the "simple ECA" use cases, it should be absolutely transparent whether the VBS is used, or whether the main-memory-based Java VariableBindings class is used in some step. Especially, if e.g., in a join operation, two instances that use different storage models are combined, this should be handled transparently.
4. Furthermore, it must be possible that the decision to use the main memory-based VariableBindings implementation can be revised at any point by a service. This happend e.g., when a query that has been invoked for a single tuple, returns hundreds of tuples, or when a set of variable bindings stored in the database is extended with an XML binding (cf. Section 5).

4.2 Consequences

Amongst the requirements, (1) is actually just an observation that confirms that a generic functionality/wrapper in form of a Java class that belongs to a mars-common package that is imported by the services is needed.

(2) excludes a solution based on local databases. This decision is supported by the observation that it would not be attractive that every service must be complemented by a local database installation. As a consequence, a separate VBS has to be designed; there will be multiple instances of that service throughout the Web. For the rest of the paper, it remains to develop how the actual functionality is distributed between the VBS and the Java class.

(3) suggests the usual object-oriented implementation by an abstract class VariableBindings with instantiable subclasses MemoryVariableBindings and DBVariableBindings, but then (4) is not satisfied: then, an instance of MemoryVariableBindings could not turn into an instance of DBVariableBindings. Instead, *delegation* has to be used:

There is a class VariableBindings that provides the common functionality (e.g., maintaining the format of the tuples) and has a data member myVariableBindings which references either an instance of MemoryVariableBindings or an instance of DBVariableBindings, respectively, that is *only* concerned with the storage issue and the actual operations. Both are implementations of an interface VariableBindingsImpl. With this, a VariableBindings instance can execute

```
// if current myVariableBindings is an instance of MemoryVariableBindings:
myVariableBindings = new DBVariableBindings(myVariableBindings);
```

and the other way round to transparently switch its internal choice of implementation.

The class VariableBindings implements the "organizational" functionality that is common to both variants, e.g., handle the metadata including the format of the tuples, and the actual algebraic operations on the tuples are provided by VariableBindings and delegated to MemoryVariableBindings or DBVariableBindings.

4.3 Final Design

Every service can be configured to be able to use any VBS services via JDBC, and a threshold can be specified from what number of tuples on the DBVariableBindings implementation is used. If a threshold is specified, also a default VBS to be used has to be specified (by its JDBC access information) that is used when new bindings are generated. If variable bindings are communicated to a service, this service accesses the VBS (also via JDBC) where the variable bindings are hosted, which can be different from its default one.

The decision was made that all of the actual VBS *functionality* that has to do with the metadata is implemented in Java in VariableBindings, DBVariableBindings, and MemoryVariableBindings. While MemoryVariableBindings implements the relational operations on its internal data structure, the DBVariableBindings class implements the operations in the database. Here, the database is only the "dumb" SQL backend to the DBVariableBindings class, which invokes it with dynamically generated SQL commands (constructed from the metadata information) via JDBC. So, *any* database can serve with minor preparations (see

Section 4.4). Figure 3 shows the distribution of functionality between the VBS and the Java classes that are imported by the services that use the VBS.

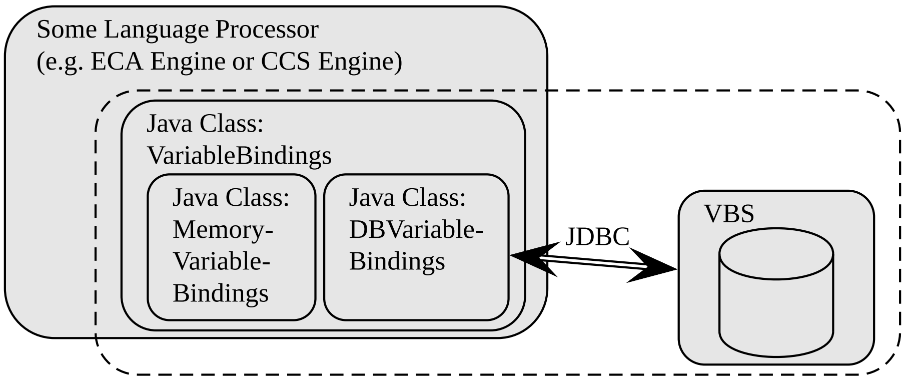


Figure 3: Distribution of Functionality between VBS and imported Java Classes

The core signature and functionality of DBVariableBindings is as follows:

- attributes mytablename, jdbcURI/user/pwd hold the current table name and the JDBC connection information;
- new DBVariableBindings(MemoryVariableBindings): initial constructor: given a tuple or an instance of MemoryVariableBindings to be converted, create a table whose attribute names are the variable names, and insert the tuple(s); the table is created in the default VBS;
- unary relational operations that modify the table and/or its contents:
  - projection: ALTER TABLE t DROP COLUMN *variables to be removed*,
  - selection: DELETE FROM t WHERE NOT *selection condition*,
  - bindInAllTuples(name, value): adds a new variable with a given value to all tuples:
    ALTER TABLE t ADD *var datatype* DEFAULT *value*;
    ALTER TABLE t MODIFY COLUMN *var* DEFAULT NULL;
- binary relational operations (with the other operand also DBVariableBindings):
  - natural join: create a new table that contains the result:
    INSERT INTO t_{new}
    (SELECT * FROM t, t' WHERE *equality of all shared variables*)

    and set mytablename to the new table name;
  - union: add tuples given in main memory representation, or the contents of a complete table,
  - minus (as removeMatchingBindings(other)):
    DELETE FROM t WHERE EXISTS
    (SELECT * FROM t' WHERE *equality of all shared variables*);
- provide an iterator over the tuples. The class VariableBindings implements the Java interface “Iterable” and can just be used in any Java loops.

Binary Operations. The implementation of binary operations is only given for two operands of the same implementation. This is accomplished by a preparing step:

If both operands are MemoryVariableBindings and their number of tuples is below the threshold (default: 2), and none of them has any variable that is bound to XML values, use MemoryVariableBindings (cf. Section 5). Otherwise, convert each of them is necessary to DBVariableBindings and use the DBVariableBindings implementation of the operand.

Example 2 *Fig. 4 shows the implementation of VariableBindings.naturalJoin(other). The fragment illustrates the polymorphism wrt. the use of DBVariableBindings and Memory-VariableBindings. The code snippet in Figure 5 shows the implementation of DBVariable-Bindings.naturalJoinInternal(other) that contains the dynamic generation of the INSERT INTO ... (SELECT FROM t_1, t_2 WHERE t_1.var$_{i_1}$=t_2.var$_{j_1}$ and . . . and t_1.var$_{i_k}$=t_2.var$_{j_k}$...) SQL command that realizes the natural join.*

The communication with the database is done via JDBC. Since the statements *must* be created dynamically according to the format of the VariableBindings instance (i.e., the variable names) and with the current table name, SQLJ is no alternative here. Statements are submitted as batches to minimize the communication overhead.

```
public class VariableBindings implements Cloneable
{
 protected VariablePropertiesList varsProps // format: names, datatypes, nullable?
 // one of these contains the actual bindings:
 public VariableBindingsImpl myVariableBindings;
    ...   [snipped] ...
 public void naturalJoin(VariableBindings other, boolean eliminateDups)
  { if (other.isTrueMemBindings())  return;      // join with "true"?
    if (other.isEmpty()) {  makeClear();  return;  }

    // if XML: turn both into Mem; otherwise if self>threshold, turn it into DB
    prepareToCombine(other);
    setVarPropsList(VBUtils.getVarsAfterJoin(this, other));

    if(isMem() && other.isMem()) // Mem x Mem
    { myVariableBindings.naturalJoin(
        (MemoryVariableBindings)other.getVariableBindings(), eliminateDups);
    }
    else // one of them is DB => do it in the DB
    { if (other.isMem() && other.size() == 1) // DB x MemTuple
        naturalJoin(other.getFirstTuple());
      else // usual case: VarBdgs naturalJoin:  at least one of them is DB
      { turnBothIntoDB(other);
        myVariableBindings.naturalJoin(other.getVariableBindings(),
           getVarPropsList(), eliminateDups);
 }}}}
```

Figure 4: Code fragment for naturalJoin

Additional functionality is provided for exchanging variable bindings stored in a table in a VBS between two services, see Section 4.5.

```
public class DBVariableBindings implements VariableBindingsImpl{
 private String tablename;
    ...[snipped] ...
 public DBVariableBindings naturalJoinInternal(DBVariableBindings other,
      VariablePropertiesList newVarsProps, boolean eliminateDups)
  {
    String newTablename = DBProxy.getTablename();
    Collection<String> commonVars = new HashSet<String>();
    Collection<String> otherOnlyVars = new HashSet<String>();

    VBUtils.sharedVars(getVars(), other.getVars(), commonVars, otherOnlyVars);
    ArrayList<String> newVarOrder = new ArrayList<String>();
    StringBuffer attrslist = new StringBuffer();

    for (String varname : commonVars)
      attrslist.append("t1." + ’"’ + varname + ’"’ + " ,");
    for (String varname : otherOnlyVars)
      attrslist.append("t2." + ’"’ + varname + ’"’ + " ,");
    attrslist.deleteCharAt(attrslist.length()-1);
    StringBuffer createTable =
      new StringBuffer("CREATE TABLE " + newTablename + " (");
    createTable.append(DBBindingsHelper.createTableColumnsStatement(newVarsProps));
    createTable.append(")");

    // Create join query with WHERE clause
    StringBuffer joinCondition = new StringBuffer ("");
    commonVars.remove(DBBindingsHelper.EmptyColumnName);
    if (!commonVars.isEmpty())
    {
      joinCondition.append("WHERE ");
      for (String varname : commonVars)
         joinCondition.append("t1." + ’"’ + varname + ’"’ + "=" +
            "t2." + ’"’ + varname + ’"’ + " AND ");
      joinCondition.delete(joinCondition.length()-5, joinCondition.length());
    }
    try {
      Connection conn = DBProxy.getConnection();
      Statement statement = conn.createStatement();
      statement.addBatch(createTable.toString());

      StringBuffer s =
        new StringBuffer("INSERT INTO " + newTablename + "(SELECT ");
      if (eliminateDups) s.append("DISTINCT ");
      s.append(attrslist);
      s.append(" FROM " + tablename + " t1, " + other.tablename + " t2 ");
      s.append(joinCondition);
      s.append(")");
      statement.addBatch(s.toString());
      statement.addBatch("COMMIT");
      statement.executeBatch();
      statement.close();
      conn.close();
    } catch (SQLException e) { e.printStackTrace(); }
    // the result does not yet belong to any VarBindings:
    return new DBVariableBindings(null, newTablename);
 }}
```

Figure 5: Code fragment for DBVariableBindings’ internal naturalJoin

4.4 Requirements on the Database

Since multiple services store their variable bindings in a VBS, the decision was made that the database administers the table names by an SQL sequence which is queried by the constructor of DBVariableBindings that then creates a new table CREATE TABLE mars_table_i with the next number i. This sequence has to be created by the database administrator.

4.5 Distributed Usage

When variable bindings stored in a table of some VBS are communicated from one service to another, for the communication a variant of the XML markup is used that does not contain the tuples themselves, but just the information in which database the variable bindings are stored, and which table holds them.

- XML serialization: the DTD is extended

  ```
  <!ELEMENT variable-bindings (tuple*)>
    <!ATTLIST variable-bindings database CDATA #IMPLIED
                                tablename CDATA #IMPLIED>
  <!ELEMENT tuple (variable*)>
  <!ELEMENT variable ANY>
     <!ATTLIST variable name CDATA #REQUIRED>
  ```

  and it is required that the variable-bindings element *either* contains tuples or has the database and tablename attributes. The communication of variable bindings stored in a table in a VBS has thus the form

  <variable-bindings database="f(*jdbc-uri, user, password*)" tablename="*tablename*"/>

  where f(*jdbc-uri, user, password*) encrypts the communicated data (e.g., by a public-key mechanism where the public keys of the services can be accessed via the LSR).
- a constructor from the above XML communication format, a VariableBindings instance is created whose myVariableBindings property refers to a new instance of DBVariableBindings that refers to the database table. The constructor reconstructs the metadata from the VBS database's data dictionary. Recall that this table can reside in a VBS that is the same or different from the receiving service's default VBS.
- If the receiving service does not support the use of the VBS that holds the table (which can also be queried from the LSR), the variable bindings are sent explicitly in the XML markup. This is done without actually materializing the whole XML tree:
  - the XML serialization of DBVariableBindings serializes tuple-wise from a JDBC result set into the HTTP message,
  - the recipient reads from the HTTP message with a SAX parser that either creates an instance of MemoryVariableBindings (if the service does not support the use of a VBS at all), or creates an a table in his default VBS and creates an SQL insert command for each tuple that is read, and submits the commands as JDBC batch statements to the VBS,

The cooperation between two services is illustrated in Figure 6.

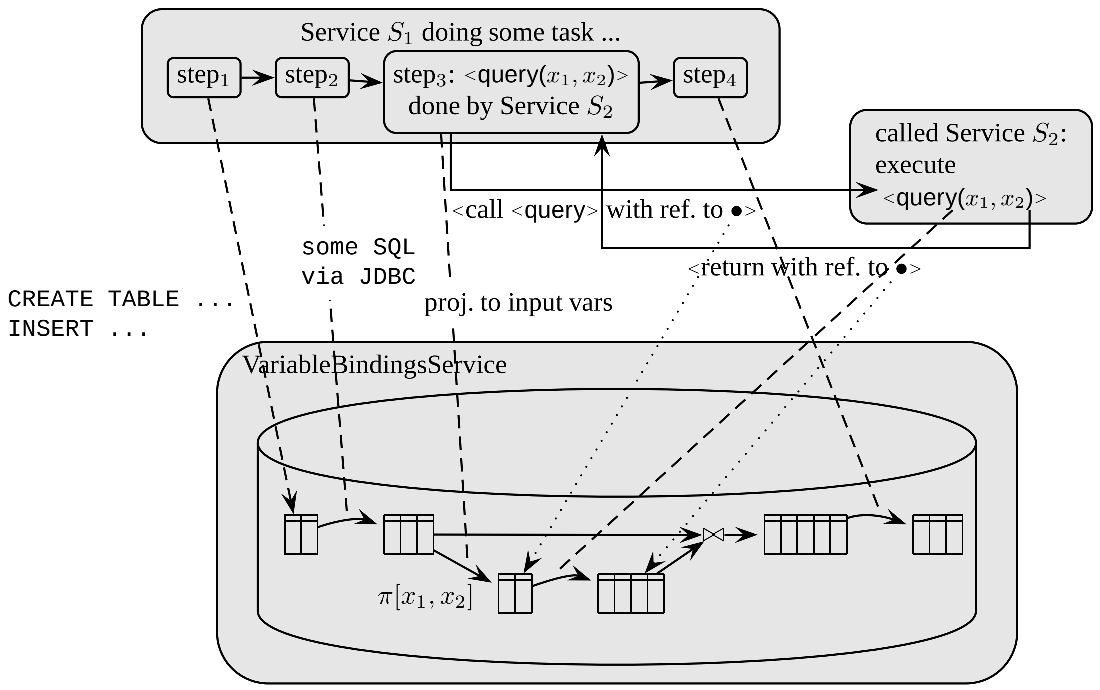

Service S_1 doing some task ...
step$_1$
step$_2$
step$_3$: <query(x_1, x_2)> done by Service S_2
step$_4$
called Service S_2: execute <query(x_1, x_2)>
<call <query> with ref. to ●>
<return with ref. to ●>
some SQL via JDBC
proj. to input vars
CREATE TABLE ... INSERT ...
VariableBindingsService
$\pi[x_1, x_2]$

Figure 6: Cooperation via a VBS

5 Some SQL Details

Case-Sensitive and Other Column Names. SQL column names are usually case-insensitive. As variable names are usually case-sensitive (especially, in a framework like MARS that allows for combination of different languages), and potentially all kinds of symbols are allowed, it would not be safe to use their names directly as column names. Even more, in some languages, the variables are not only named, but they are URIs (then, the language code fragment can be represented as an RDF graph that allows to analyze it). For that, all tables are created with column names of the form "*varname*", where *varname* is the actual variable name. As all accesses are via JDBC statements from the DBVariableBindings class, this is just an internal matter.

XMLType: No Comparison, no Join. In MARS, variables may be bound to XML fragments. While the SQLX standard contains XMLType, this datatype cannot yet be fully used: there is no comparison on it! Actually, a comparison of XML instances would be expensive, since the attributes are not ordered, thus, the ASCII serialization could not be used as a workaround. It would need a specific implementation of deep-equality.

For most other applications, this is no problem, as the rows have another key, and the database is just used for storing XML, operating on it, reading it etc., but usually no comparisons are needed.

In contrast, an important part of the generic functionality of variable bindings is to join them – which needs join conditions, and thus comparisons of values. In the MARS environment, the following is a common situation:

- given: a set of tuples $\{\beta_1, \ldots, \beta_n\}$ of variable bindings for variables $v_1, \ldots, v_k$,
- next step: a query that has some projection $v_{i_1}, \ldots, v_{i_\ell}$ as input,
- the result of the query binds variables $v_{i_1}, \ldots, v_{i_\ell}, w_1, \ldots, w_m$,
- this result has to be joined with $\{\beta_1, \ldots, \beta_n\}$ over the join variables $v_{i_1}, \ldots, v_{i_\ell}$,

where some v_{i_j} can be of type XML. Especially, if $\ell = 1$, and v_{i_1} is bound to XML fragments, and the query is an application of an XPath expression for extracting values *from* the XML fragment, this is the only join variable.

So, in case that *any* variable is of type XMLType, VariableBindings delegates to MemoryVariableBindings (that uses XML deep-equality), no matter how many tuples.

6 Conclusion

The use case discussed in this paper shows how generic database functionality is used in the MARS framework for operating on sets of tuples of variable bindings, and for providing the base for cooperation on them between different services. The database service is complemented with a Java class that is imported by other Web Services that use the functionality. The database service is actually only used as (i) central storage and (ii) to execute SQL statements on it that are generated dynamically by the Java class. Providing such a service does only require *minimal* preparation effort: create an SQL sequence.

Bibliography

[BFMS06] E. Behrends, O. Fritzen, W. May, and D. Schubert. An ECA Engine for Deploying Heterogeneous Component Languages in the Semantic Web. In *Web Reactivity (EDBT Workshop)*, Springer LNCS 4254, pp. 887–898, 2006.

[BFMS08] E. Behrends, O. Fritzen, W. May, and F. Schenk. Embedding Event Algebras and Process Algebras in a Framework for ECA Rules for the Semantic Web. *Fundamenta Informaticae*, 82:237–263, 2008.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB*, pages 606–617, 1994.

[HSL08] T. Hornung, K. Simon, and G. Lausen. Mashing Up the Deep Web - Research in Progress. In *WEBIST*, pages 58–66. INSTICC Press, 2008.

[MAA05] W. May, J. J. Alferes, and R. Amador. Active Rules in the Semantic Web: Dealing with Language Heterogeneity. In *RuleML*, Springer LNCS 3791, pp. 30–44. Springer, 2005.

[Mil83] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, pages 267–310, 1983.