Server-Centric P3P

Rakesh Agrawal Jerry Kiernan Ramakrishnan Srikant Yirong Xu
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120
{ragrawal, kiernan, srikant, yirongxu}@almaden.ibm.com

Abstract

Platform for Privacy Preferences (P3P) is the most significant effort currently underway to enable web users to gain control over their private information. P3P provides mechanisms for web site owners to express their privacy policies in a standard format that a user can programmatically check against her privacy preferences to decide whether to release her data to the web site. We discuss architectural alternatives for implementing P3P and present a server-centric implementation that reuses database querying technology, as opposed to the prevailing client-centric implementations based on specialized engines. Not only does the proposed implementation have qualitative advantages, our experiments indicate that it performs significantly better than the sole public-domain client-centric implementation and that the latency introduced by preference matching is small enough for real-world deployments of P3P.

1. Introduction

The privacy of personal information on the Internet has become a major concern for governments, businesses, media, and the public [5] [12] [13] [18] [24]. Opinion surveys consistently show that privacy concerns are a leading impediment to the further growth of web-based commerce [10] [25] [26] [27]. Initial efforts by web sites to disclose their privacy policies have had limited impact because these policies were often too lengthy for users to read and were written in a language too difficult for users to understand.

Platform for Privacy Preferences (P3P), developed by the World Wide Web Consortium (W3C), is the most significant effort underway to enable users to gain more control over what information a web-site collects. It provides a way for a web site to encode its data-collection and data-use practices in a machine-readable XML format, known as a P3P policy [9], which can be programmatically compared against a user's privacy preferences [8]. The current P3P standard only provides a mechanism for users to check a web site's privacy policy before they release personal information to the site; mechanisms for enforcing that sites act according to their stated policies are beyond its scope [9].

In this paper, we propose a server-centric architecture for P3P that reuses database technology, as opposed to the prevailing client-centric implementations based on specialized engines. The server-centric architecture has several advantages (discussed in Section 4.2) including: setting up the infrastructure necessary for ensuring that web sites act according to their stated policies, allowing P3P to be deployed in thin, mobile clients that are likely to dominate Internet access in the future, and allowing site owners to refine their policies based on the privacy preferences of their users. Our experiments indicate that the proposed server-centric architecture performs significantly better than the sole public-domain client-centric implementation [17]. These experiments also show that the proposed architecture has the necessary performance for it to be used in practical deployments of P3P.

1.1. Related Work

P3P is a nascent standard (it became a W3C Recommendation on April 16, 2002) and the tools for implementing P3P are still in their infancy. A survey of current P3P implementations is available in [19]. Two prominent implementations of P3P are: AT&T Privacy Bird and the implementation of compact P3P policies in the Microsoft Internet Explorer. In Section 3, we discuss these and some tools to aid P3P implementation.

A vision for Hippocratic databases was recently presented in [1]. Inspired by the privacy tenet of the Hippocratic Oath that has guided the conduct of physicians for centuries, Hippocratic databases make privacy their central concern. The paper proposes a strawman design for Hippocratic databases, identifies the technical challenges and problems in designing such databases, and suggests potential solutions.

The work presented in this paper is complementary to the work on Hippocratic databases. Our focus in this paper is on the specific problem of how database technology can be used for checking P3P privacy policies against users' privacy preferences, while the Hippocratic database paper is mostly focused on how to enforce a given privacy policy. The lessons from this study can be applied to the implementation of the *Privacy Constraint Validator* module in the Hippocratic database design. As we mentioned earlier, P3P does not specify any mechanisms for ensuring that sites act according to their stated policies. Thus mech-

anisms developed for Hippocratic databases may provide the basis for designing enforcement in P3P.

Related work includes research on integrating production rules with database systems (see [15] for an overview). The dominant application of the database rule systems is to monitor events (mostly updates) and cause specified actions to be triggered if certain conditions are satisfied. P3P preferences expressed in APPEL also consist of a set of rules. However, the APPEL rules have little in common with the rules studied in database systems. APPEL rules that define a user's preference are not installed in the database. They are really queries that are processed at the time a user accesses a web site to determine if the website's policy conforms to the user's privacy wishes at that moment.

1.2. Paper Layout

The rest of the paper is structured as follows. We give a brief overview of P3P in Section 2 and describe the current P3P implementations in Section 3. We give the architecture of our proposed server-centric implementation in Section 4, and discuss the pros and cons of the server-centric approach. We give the algorithms for storing P3P polices into relational tables, and converting APPEL preferences into either SQL or XQuery in Section 5. We present the results of our performance experiments in Section 6 and conclude with some remarks and directions for future work in Section 7.

We assume familiarity with the basic concepts of XML [28]. Throughout the paper, we use "element" and "attribute" as in the XML specification. For clarity, we will sometimes refer to an APPEL element as an "expression". Their subelements will be correspondingly called "subexpression". A condensed version of this report appears in the proceedings of the the 19th International Conference on Data Engineering (ICDE 2003) [2].

2. Overview of P3P

The P3P protocol has two parts:

1. *Privacy Policies:* An XML format in which a web site can encode its data-collection and data-use practices [9].
2. *Privacy Preferences:* A machine-readable specification of a user's preferences that can be programmatically compared against a privacy policy [8].

W3C has standardized the XML format for describing privacy policies and it is now a W3C Recommendation [9]. A user may specify privacy preferences in APPEL [8], which is currently a working draft in W3C.[1] APPEL provides an XML format for expressing preferences and an algorithm for matching preferences against policies.

In the rest of this section, we briefly review the core features of the P3P policy language as well as APPEL. A web site can have different privacy policies governing different parts of the site. P3P provides for a *Reference File* in which a site can set up associations between web pages and policies. We describe this mechanism in Section 2.3. See [9] and [8] for complete specifications of P3P and APPEL (respectively).

2.1. Policy Description

P3P policies are described in XML format as a sequence of STATEMENT elements that have the following subelements:

- CONSEQUENCE: describes the intended purpose for collecting information in human-readable text.
- PURPOSE: describes purposes for which information is collected. Multiple purposes can be listed in a STATEMENT if all of them have the same values for RECIPIENT, RETENTION and DATA-GROUPS; otherwise, they are specified in different STATEMENT elements.
- RECIPIENT: describes the intended users of the collected information. Multiple recipients can be specified in one statement.
- RETENTION: defines the duration for which the collected information will be kept.
- DATA-GROUP: provides the list of individual data items (specified using DATA tags) that are collected for stated purposes in the statement.

P3P has predefined values for PURPOSE (12 choices), RECIPIENT (6), and RETENTION (5). Examples of PURPOSE include:

- *current:* completion and support of activity for which data was provided,
- *individual-decision:* inferring habits, interests, and other characteristics of individuals, and
- *contact:* contacting visitors for marketing of services or products through a communication channel other than voice telephone.

Examples of RECIPIENT include:

- *ours:* ourselves,
- *same:* legal entities following our practices, and
- *unrelated:* legal entities whose practices are unknown to us.

Examples of RETENTION include:

- *stated-purpose:* discarded at the earliest time possible,
- *business-practice:* long term retention but with a destruction time table, and
- *indefinitely*.

P3P also has predefined types of data items. It is also possible to assign CATEGORIES to data items.

A policy can provide *opt-in* or *opt-out* values for the *required* attribute of PURPOSE and RECIPIENT elements.

[1] P3P does not require that APPEL be necessarily used as the language for expressing privacy preferences. However, we are not aware of any alternate proposal.

```
<POLICY>
  ... ...
  <STATEMENT>
    <PURPOSE><current/></PURPOSE>
    <RECIPIENT><ours/><same/></RECIPIENT>
    <RETENTION><stated-purpose/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#user.name"/>
      <DATA ref="#user.home-info.postal/>
      <DATA ref="#dynamic.miscdata">
        <CATEGORIES><purchase/></CATEGORIES>
      </DATA>
    </DATA-GROUP>
  </STATEMENT>

  <STATEMENT>
    <PURPOSE>
      <individual-decision required="opt-in"/>
      <contact required="opt-in"/>
    </PURPOSE>
    <RECIPIENT><ours/></RECIPIENT>
    <RETENTION><business-practices/></RETENTION>
    <DATA-GROUP>
      <DATA ref="#user.home-info.online.email/>
      <DATA ref="#dynamic.miscdata">
        <CATEGORIES><purchase/></CATEGORIES>
      </DATA>
    </DATA-GROUP>
  </STATEMENT>
</POLICY>
```

Figure 1: Volga's Privacy Policy in P3P

The opt-in value says that the user must provide explicit consent to the stated purpose/recipient. The opt-out value gives the user flexibility to reject the specified purpose/recipient, but she needs to take additional action for the opt-out to take effect.

An Example Policy Volga is a bookseller who needs to obtain certain minimum personal information to complete a purchase transaction. This information includes name, shipping address, and the credit card number. Volga also uses the purchase history of customers to offer personalized book recommendations, for which it needs customer's email address.

Figure 1 shows how Volga's policy may look like in the P3P policy language. The first STATEMENT says that the name, postal address, and miscellaneous purchase data (i.e., book titles, credit card number, etc.) will be used for completing the current purchase transaction.

The second STATEMENT allows Volga to use miscellaneous purchase data for creating personalized recommendations and email them to the customer. However, the opt-in value of the *required* attribute of the purposes "individual-decision" and "contact" implies that the explicit customer consent is necessary. By default, the value of the *required* attribute is set to *always*, which precludes the possibility of customer opt-in or opt-out.

2.2. Privacy Preferences

Privacy preferences are expressed in APPEL as a list of RULEs [8]. Rules are evaluated in the order in which they are specified. A rule consists of two parts:

```
<appel:RULESET>
  <appel:RULE behavior="block">
    <POLICY>
      <STATEMENT>
        <PURPOSE appel:connective="or">
          <admin/><develop/><tailoring/>
          <pseudo-analysis/><pseudo-decision/>
          <individual-analysis/>
          <individual-decision required="always"/>
          <contact required="always"/>
          <historical/><telemarketing/>
          <other-purpose/><extension/>
        </PURPOSE>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="block">
    <POLICY>
      <STATEMENT>
        <RECIPIENT appel:connective="or">
          <delivery/><other-recipient/>
          <unrelated/><public/><extension/>
        </RECIPIENT>
      </STATEMENT>
    </POLICY>
  </appel:RULE>

  <appel:RULE behavior="request"/>
    <appel:OTHERWISE/>
  </appel:RULE>
</appel:RULESET>
```

Figure 2: Jane's Privacy Preferences in APPEL

- *Rule behavior:* Specifies the action to be taken if the rule fires. The behavior can be *request*, implying that the policy conforms to preferences specified in the rule body. It can be *block*, implying that the policy does not respect user's preferences. See [8] for other behaviors.
- *Rule body:* Provides the pattern that is matched against a policy. The format of a pattern follows the format used in specifying privacy policies described earlier.

An interesting aspect of APPEL is the *connective* attribute, which defines the logical operators of the language. A connective can be: *or*, *and*, *non-or* (negated or), *non-and* (negated and), *or-exact*, and *and-exact*. Every element (expression) in an APPEL rule has a connective associated with it, the default connective being *and*. The two unusual connectives are and-exact and or-exact, whose semantics are as follows:

- *and-exact:* A successful match is made if (a) all of the contained expressions can be found in the policy and (b) the policy contains only elements listed in the rule. For *and*, only part (a) needs to be satisfied, not part (b).
- *or-exact:* A successful match is made if (a) one or more of the contained expressions can be found in the policy, and (b) the policy only contains elements listed in the rule. For *or*, only part (a) needs to be satisfied, not part (b).

An Example APPEL Preference Jane is a privacy-conscious consumer who wants retailers to use her personal information only to complete her purchase transactions. At the same time, Jane likes individualized recommendations emailed to her and does not mind her purchase history to be used for this purpose. However, she wants the possibility of opting-in or out of this service and does not want to release her information to a retailer who does not offer this choice.

Figure 2 shows Jane's preferences, comprising of two rules. The first rule blocks all purposes other than current, but leaves the possibility of opting-in or out of the individual-decision and contact purposes. The second rule ensures that the only possible recipients of Jane's data are the retailer or its agents following the same privacy practices. The final rule allows data to be released if the first two rules do not fire.

Observe that Volga's policy conforms to Jane's preferences. The first rule in Jane's preferences does not fire because the set of purposes in Volga's policy does not intersect with the corresponding set in Jane's rule, except for contact and individual-decision. For these two elements, the "required" attribute does not match (opt-in versus always), and hence the elements do not match either. Note that if individual-decision was not specified as opt-in in Volga's policy, the default value of always would have been presumed for the required attribute. Then, the first rule in Jane's preferences would have fired, stopping Jane from providing her data to the site. Similarly, the second rule in Jane's preferences does not fire because none of the recipients in Jane's rule are present in Volga's policy.

2.3. The Reference File

A site may offer many services, each implemented with a specific set of web pages. This site can have multiple privacy policies corresponding to the practices associated with the various services. A site's reference file assigns individual policies with subsets of the URIs at a site.

For each policy referenced, the reference file has a set of INCLUDE/EXCLUDE declarations that together define the URIs covered by the policy. Once a specific policy for a requested URI has been located using the reference file, the APPEL preferences can be matched against the selected P3P policy to determine if the request for the URI content should proceed.

3. Current P3P Implementations

We first describe the client-centric architecture for implementing P3P outlined in [29], along with some actual implementations. Next, we describe some tools.

3.1. Client-Centric Architecture

A hypothetical architecture for implementing P3P has been described in [29]. There are two parts to deploying P3P. Web sites first create and install policy files at their sites (see Figure 3), possibly using some tools discussed below.

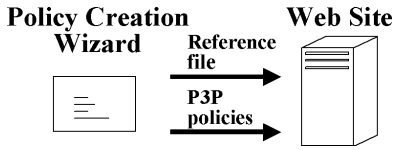

Figure 3: Creation and Installation of Policies (Client-Centric)

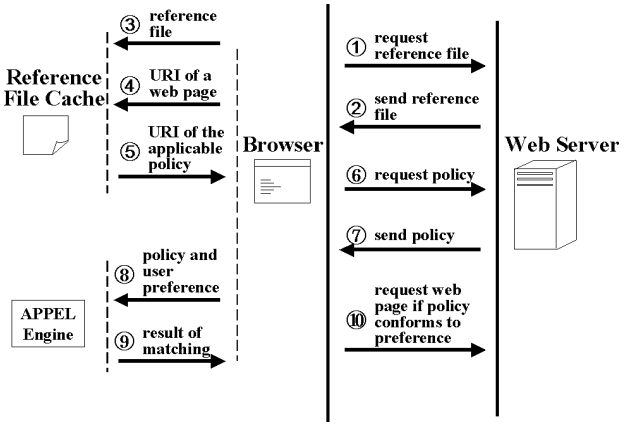

Figure 4: Policy-Preference Matching (Client-Centric)

Then as the users browse a web site, their preferences are checked against a site's policy before they access the site (see Figure 4).

3.2. Implementations

There are two prominent implementations of the above architecture: Microsoft IE6 and AT&T Privacy Bird.

IE6 implementation of Compact P3P Policies Internet Explorer 6 allows a user to specify her privacy preference for handling cookies. When the user requests a page from a web site, IE6 allows the website to place a cookie only if the site provides a compact version of the applicable P3P privacy policy, and that policy is compatible with the user's preference. (See [9] for details about the compact policies.) The user can manually override this decision by specifying websites whose cookies should always be allowed (or disallowed).

Privacy Bird AT&T Privacy Bird is available from privacybird.com as a browser extension to IE5 and up. It accepts user-defined APPEL privacy preferences, and also includes an APPEL engine to compare a user's APPEL preference with a web site's P3P policy as the user is browsing the web.

3.3. Tools

Some tools have become available to aid with P3P implementation and deployment. A complete inventory can be found at [19]; we briefly survey some of them.

Creating Policies. P3PEdit, available from p3pedit.com, is a web-based privacy policy generator. Users create their policies by answering short privacy-related questions in

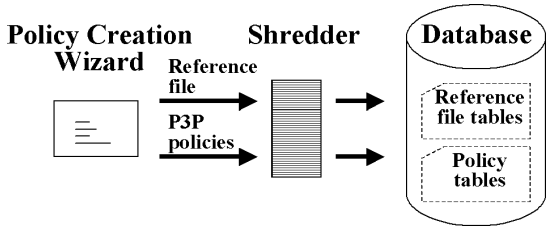

Figure 5: Creation and Installation of Policies (Server-Centric)

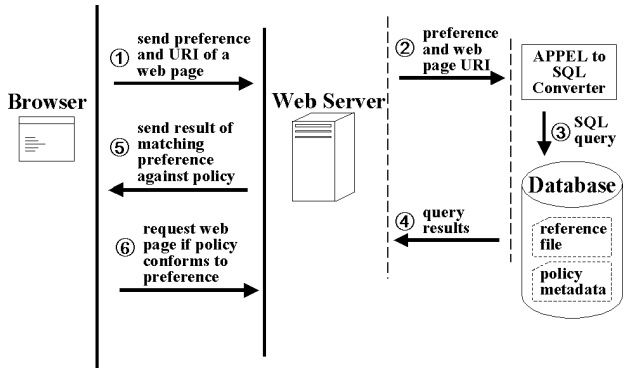

Figure 6: Policy-Preference Matching (Server-Centric)

plain English. IBM Tivoli Privacy Wizard [16] lets a company define privacy policies using a web-based GUI tool. The policies so created can be exported in the P3P format.

Creating APPEL Preferences. JRC APPEL Preference Editor [17] is a Java-based editor for preparing APPEL preferences. Each APPEL RULE can be added either by choosing from a set of predefined RULEs, or by using an advanced mode that gives options for creating a rule.

Checking APPEL Preferences. JRC P3P Proxy [17] is a centralized proxy service that conducts P3P privacy policy checking on behalf of subscribed users. A user can specify her APPEL preference for her account. After a user changes the proxy settings of her browser, her further browsing requests are redirected to the proxy service. The proxy handles the matching of P3P and APPEL and takes appropriate actions on behalf of the user.

4. Server-Centric Architecture for P3P

We propose a server-centric architecture for deploying P3P as an alternative to the prevailing client-centric architecture. In this architecture, a website deploying P3P first installs its privacy policies in a database system as shown in Figure 5. Then database querying is used for matching a user's preferences against privacy policies as shown in Figure 6.[2]

[2]We are assuming that the client preferences will continue to be expressed in APPEL and they will be translated into database queries before the matching takes place. This translation step may become unnecessary should the proposed architecture catch on. For in that case, database queries may replace APPEL for representing privacy preferences and the GUI tools for generating preferences may directly generate database queries.

We envision three variations of this architecture:

1. Convert privacy policies into relational tables [3] [7] [11] [14] [20] [21] [23], and convert an APPEL preference into an SQL query for matching.
2. Store privacy policies in relational tables, define an XML view over them [3] [6] [7] [21] [22], and use an XQuery [4] derived from an APPEL preference for matching.
3. Store privacy policies in a native XML store and use an XQuery derived from an APPEL preference for matching.

Given that P3P policies are expressed in XML, storing them in a native XML store in the third variation is straightforward. This variation also requires converting an APPEL preference into an XQuery for matching, the techniques for which are also those of the second variation.

4.1. Other Alternatives

There are two orthogonal dimensions in the space of choices for implementing P3P:

1. What engine should be used for matching a preference against a policy? Should it be a specialized engine (e.g. a native APPEL engine) or should it be a general purpose engine (e.g. a database engine)?
2. Where should the matching take place? Should it happen at the client or the server?

	Client	Server
Specialized engine	Current	?
Database engine	?	Proposed

Figure 7: Architectural Choices

Figure 7 shows the decision matrix. The current P3P deployments are using specialized APPEL engines to do preference matching at the clients. It is possible to continue to use a specialized engine, but move the matching to the server. However, this choice is less attractive as we lose the benefits of using the database engine for matching. Similarly, it is possible to do the checking at the client, but use database querying. This alternative has the advantage of avoiding the need for a specialized engine. However, it will require moving the database tables from the server to a light-weight main memory database system in the client, which is also not very attractive. We have, therefore, focussed on the alternative of using database querying at the server.

4.2. Advantages and Disadvantages

The following are some of the advantages of the server-centric architecture of P3P over the client-centric architecture:

- The preference checking at the client leads to heavier clients, which is a problem for thin, mobile devices that are likely to dominate Internet access in the future. Our proposal allows for lean clients.

- An upgrade in P3P specification may require an upgrade in every client, which can be a couple of orders of magnitude greater effort than upgrading all the servers.
- As new privacy-sensitive applications emerge, they will each require building preference checking into them, rather than reusing checking done at the server.
- Site owners can refine their policies if they know what policies have a conflict with the privacy preferences of their users. The current architecture does not allow the site owners to obtain this information.

Using databases for preference matching (in the server-centric architecture) yields the following additional advantages:

- We are creating the infrastructure necessary for enhancing P3P with enforcement in the future. The privacy data tables built for checking preferences against policies may serve as meta data for ensuring that policies are followed.
- Specialized preference checking engines are reinventing querying. We, on the other hand, can reuse the proven database technology for checking preferences against policies.
- Policies of a website will not stay static forever. Versions of policies can be better managed using a database system than the current file system based implementations.

The server-centric architecture has some disadvantages too, including:

- There needs to be a greater amount of trust in the server. For example, the server can see the user's preferences. Similarly, the user has to trust the database software being used by the server, whereas the user can (in principle) choose the checking software used in the browser.
- By caching a reference file, the client may avoid some checks, assuming a user visits many pages that are governed by the same policy. On the other hand, it is possible to design a hybrid architecture in which the reference file processing is done at the client while the preference checking is done at the server.

5. System Description

We now describe the important components of our implementation of the server-centric architecture described in the previous section. We discuss the schema of the tables used for storing the policies and how those tables are populated. We also describe how we translate APPEL into SQL and XQuery.

5.1. Database Schema for P3P

The SQL query corresponding to an APPEL preference will depend on the SQL tables used for storing the P3P policies. For pedagogical reasons, we first give a simplified relational schema and explain translation of APPEL preferences into SQL in terms of this schema. Later, we give the optimized relational schema used in our implementation.

```
// e.name() returns the name of the element e
for each element e defined in the P3P policy do
    create a table such that
        (a) the name of the table is e.name()
        (b) the columns of the table consist of
            (i)  an id column whose name is e.name()
                 concatenated with "_id"
            (ii) foreign key comprising of the primary key
                 of the table corresponding to the parent element
            (iii) one column for each attribute of e
        (c) the primary key of the table comprises of
            concatenation of columns in (i) and (ii)
```

Figure 8: Schema Decomposition Algorithm

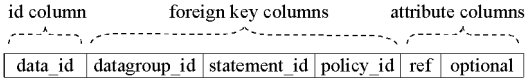

Figure 9: The Data Table

Figure 8 shows the algorithm for decomposing the P3P schema into tables. Figure 9 shows, as an example, the table created for the DATA element using this algorithm. The Data table will contain one row for every DATA element appearing in a policy. The data_id field will contain an identifier for this row. The foreign-key will consist of the primary key of the table corresponding to the parent element, DATA-GROUP. Finally, the primary key for the DATA table will consist of the concatenation of data_id with the foreign key.

5.2. Storing Policies in the Database

Having defined the relational schema for storing P3P information, the next step is to populate the tables with the data. Figure 10 gives the algorithm.

5.3. Translating APPEL Preferences into SQL Queries

Recall that an APPEL preference may contain multiple rules, and that these rules are matched against a policy in the order in which they appear. We translate each rule into

```
add(Element e, ForeignKey f) {
    create a unique id;
    create a record consisting of
        (a) id,
        (b) foreign key f, and
        (c) all attributes of element e;
    insert the record into the table e.name();
    for each subelement se of e do
        add(se, id concatenated with f);
}

Initial call: add(POLICY, φ);
```

Figure 10: Data Population Algorithm

```
1   String main (Rule r) {
2       String sql = "SELECT" + r.behavior() +
3                    "FROM"    + applicablePolicy() +
4                    "WHERE";
        // recursively match subexpressions of r
5       let θ = r.connective();   // θ is either "OR" or "AND"
6       for each subexpression se of r do
7           sql += "EXISTS (" + match(se) + ")" + θ;
8       return sql;
9   }

10  String match (Expression e) {
11      String sql;
12      sql += "SELECT *" +
13             "FROM"      + e.name() +
14             "WHERE";
        // generate path connecting e with parent element
15      sql += e.foreignKey() + "=" + e.parent().primaryKey();
        // match attributes of e
16      for each attribute attr of e do
17          sql += "AND" + attr.name() + "=" + attr.value();
        // recursively match subexpressions of e
18      String sqlSub;
19      let θ = e.connective();   // θ is either "OR" or "AND"
20      for each subexpression se of e do
21          sqlSub += "EXISTS (" + match(se) + ")" + θ;
22      return sql + "AND (" + sqlSub + ")";
23  }
```

Figure 11: Algorithm for Translating an APPEL Preference into an SQL Query

a SQL query using the algorithm in Figure 11, and submit the queries to the database in order. To simplify exposition, the algorithm pseudocode assumes that the APPEL rule uses only "or" and "and" connectives. Translations of other connectives are given in Appendix A. The pseudocode also omits checks for not generating superfluous parenthesis as well as unneeded trailing "OR" or "AND" operators in the query.

The main() function in the SQL translation mirrors the structure of the APPEL rule. The SELECT clause (line 2) specifies the behavior, i.e., the action to be taken if the rule is satisfied. The FROM clause (line 3) provides the policy_id, and the WHERE clause (line 4–7) provides the code for matching the expressions in the rule against the elements in the P3P policy. The policy_id on line 3 is obtained by calling the applicablePolicy() function. This function (details omitted) generates a subquery that queries tables storing the data from the P3P reference file, and returns the id of the applicable policy against which the rule must be evaluated.

An APPEL expression is satisfied by matching its attributes and the constituent subexpressions which are connected through the APPEL logical operators. The match() function generates the SQL code for matching an APPEL expression as follows:

```
1   <appel:RULE behavior="block">
2     <POLICY>
3       <STATEMENT>
4         <PURPOSE appel:connective="or">
5           <admin/>
6           <contact required="always"/>
7         </PURPOSE>
8       </STATEMENT>
9     </POLICY>
10  </appel:RULE>
```

Figure 12: Simplified First Rule from Jane's APPEL Preference

```
                // main(<appel:RULE>)
1   SELECT 'block'    // rule's behavior
2   FROM ApplicablePolicy
                // ApplicablePolicy represents
                // subquery that returns record
                // with ID of applicable policy.
3   WHERE
4   EXISTS (
                // match(<POLICY>)
5       SELECT *
6       FROM Policy
7       WHERE Policy.policy_id=ApplicablePolicy.policy_id AND
8       EXISTS (
                // match(<STATEMENT>)
9         SELECT *
10        FROM Statement
11        WHERE Statement.policy_id = Policy.policy_id AND
12        EXISTS (
                // match(<PURPOSE>)
13          SELECT *
14          FROM Purpose
15          WHERE
16          Purpose.policy_id = Statement.policy_id AND
17          Purpose.statement_id = Statement.statement_id AND
18          (EXISTS (
                // match(<admin>)
19            SELECT *
20            FROM Admin
21            WHERE
22            Admin.policy_id = Purpose.policy_id AND
23            Admin.statement_id = Purpose.statement_id AND
24            Admin.purpose_id = Purpose.purpose_id )
                // back to match(<PURPOSE>)
25          OR      // line 21 of match()
26          EXISTS (
                // match(<contact required=...>)
27            SELECT *
28            FROM Contact
29            WHERE
30            Contact.policy_id = Purpose.policy_id AND
31            Contact.statement_id = Purpose.statement_id AND
32            Contact.purpose_id = Purpose.purpose_id AND
                // lines 16-17 of match()
33            Contact.required = 'always' )
34            )       // back to match(<PURPOSE>)
35        )           // back to match(<STATEMENT>)
36    )               // back to match(<POLICY>)
37  )                 // back to match(<appel:RULE>)
```

Figure 13: SQL Translation of APPEL in Figure 12

1. Select elements in the P3P policy from the table corresponding to this APPEL expression (lines 12–13).
2. Ensure that the elements belong to their parent elements by joining this table to the parent table, with the join condition that the table's foreign key is the same as the parent table's primary key (lines 15).
3. Match any attributes specified in the APPEL expression (lines 16–17).
4. Recursively match any subexpressions (lines 20–21), with the appropriate connective.

5.3.1 Translation Example

Figure 13 shows the SQL translation of a simplified version of the first rule in Jane's preference (Figure 12). The function calls that generated the code are shown as grayed-out comments.

The main() function generates the outer-most SELECT clause (Figure 13, lines 1–3), which returns the behavior of Jane's rule when an applicable policy matches it. The behavior ('block' in this case) is obtained from the behavior attribute of the rule. The applicable policy is found by a subquery (generated by applicablePolicy()) that accesses tables storing the data from the reference file. For simplicity, the example translation assumes that the result of this subquery has been stored in the one-row temporary table "ApplicablePolicy" containing the id of the applicable policy.

The rest of the SQL query is generated by recursively calling the match() function for every subexpression in the rule. The match() function is first called to generate SQL for the outer most POLICY expression, as shown in lines 5–8. The SQL selects POLICY elements from the Policy table and uses a join to ensure these elements have the applicable id. The match() function is then recursively called to generate SQL for the STATEMENT expression. The generated SQL (lines 9–12) selects STATEMENT elements from the Statement table and uses a join to ensure these elements belong to their parent POLICY elements. The match() function is next called for the PURPOSE expression, which has an "or" connective. The generated SQL is shown in lines 13–18 and 25–26. Notice that the "or" connective appears as a SQL "OR" operator (line 25). Finally, the match() function is called to generate SQL for the "admin" (lines 19–24) and "contact" (lines 27–33) expressions. Note that the SQL code for "contact" also specifies (line 33) that the "required" attribute has to have the value "always" for this disjunct to be true.

5.4. Optimizations

The algorithm in Figure 8 generates a schema that has a uniform structure which makes the translation algorithm easy to understand. We take the schema generated by this algorithm and reduce the number of tables in order to reduce the number of joins in the generated SQL queries. These optimizations draw upon the rich body of research on generating a relational schema from an XML schema [7] [11] [14] [20] [21] [23].[3]

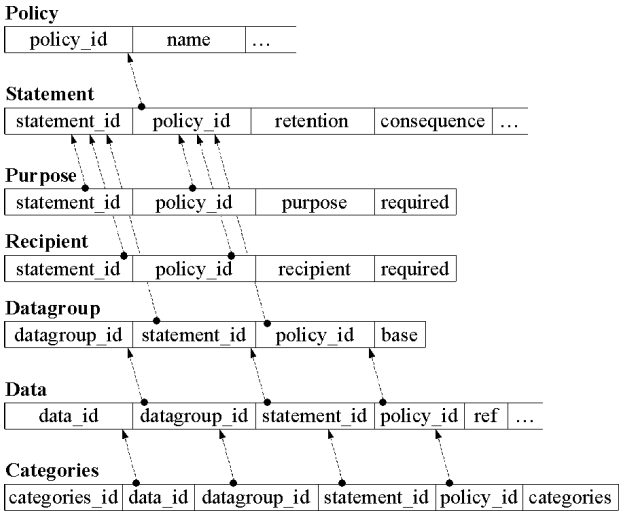

Figure 14: Optimized tables for storing policies. Arrows represent foreign key relationships.

Figure 14 shows the schema used in our implementation that incorporates the following optimizations:

- We do not create separate tables for P3P subelements that define the values for PURPOSE, RECIPIENT, and CATEGORIES. Instead, we store them in the table for their corresponding parent element. We also add additional columns in the parent table for storing their attributes. For example, the "Purpose" table has a "purpose" column that is used to store the values of purposes appearing as subelements of PURPOSE, and a "required" column for storing the value of the subelements' "required" attribute.

  Moreover, the tables for PURPOSE and RECIPIENT do not need an id column because there can be only one such element in a STATEMENT. Thus, for these two tables, the statement_id concatenated with policy_id suffices as a primary key.

- We do not create separate tables for subelements defining the values of RETENTION (e.g. "stated-purpose"). We store them with the grand-parent element STATEMENT, since in P3P, each STATEMENT can have only one RETENTION element, and the latter can have only one subelement.

- Instead of a separate table for CONSEQUENCE, we store its value in a nullable "consequence" column in the Statement table because a STATEMENT can contain at most one CONSEQUENCE.

[3] Rather than hand-crafting our relational schema, we would have liked to use a schema generation tool. Unfortunately, despite our best efforts, (including contacting authors), we could not get access to the externally developed tools. And our locally developed prototype [6] [21] [22] could not handle the full richness of the P3P schema.

```
SELECT 'block'
FROM ApplicablePolicy
WHERE
EXISTS (
   SELECT *
   FROM Policy
   WHERE Policy.policy_id=ApplicablePolicy.policy_id AND
   EXISTS (
      SELECT *
      FROM Statement
      WHERE Statement.policy_id = Policy.policy_id AND
      EXISTS (
         SELECT *
         FROM Purpose
         WHERE
         Purpose.policy_id = Statement.policy_id AND
         Purpose.statement_id = Statement.statement_id AND
         ( Purpose.purpose = 'admin'
            OR
            Purpose.purpose = 'contact' AND
            Purpose.required = 'always'
         ) ) ) )
```

Figure 15: SQL Translation of Jane's Preference for the Optimized Schema

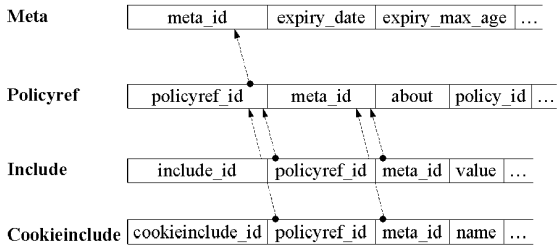

Figure 16: Tables for storing reference files. Arrows represent foreign key relationships.

The translation algorithm is modified to work against this schema. Figure 15 shows the translation of Jane's simplified first rule given in Figure 12. The translation algorithm also has special functions for some subexpressions (such as PURPOSE and RECIPIENT) in order to merge several subqueries into a single subquery. Thus the two subqueries in Figure 13 (lines 19–24 and lines 27–33) are merged into a single subquery in Figure 15.

5.5 Tables for Storing the Reference File Information

Figure 16 shows the schema of the tables used for storing information in the reference file. The META element is the top level element in a reference file. The Policyref table corresponds to the POLICY-REF subelement of a META element. Each row references in the about column a single policy among many possible privacy policies that a site may have. The policy_id column gives the unique id of the corresponding policy. There can be multiple POLICY-REF elements within a META element, mapping distinct portions of the site to specific privacy policies. The portion of a site covered by a policy is derived from INCLUDE and

```
1   String main (Rule r) {
2       String xq =
3           "if (document(" + applicablePolicy()+ ")[";
4       let θ = r.connective(); // θ is either "OR" or "AND"
5       for each subexpression se of r do
6           xq += match(se) + θ;
7       xq += "]) then <" + r.behavior() + ">";
8       return xq;
9   }

10  String match (Expression e) {
        // match attributes of e
11      String xqAttr;
12      for each attribute attr of e do
13          xqAttr += "@" + attr.name() + "=" + attr.value();
14          xqAttr += "AND";

        // recursively match subexpressions of e
15      String xqSub;
16      let θ = e.connective(); // θ is either "or" or "and"
17      for each subexpression se of e do
18          xqSub += match(se) + θ;

19      return e.name() + "[" + xqAttr + "(" + xqSub + ")]";
20  }
```

Figure 17: Algorithm for Translating an APPEL Preference into an XQuery

EXCLUDE subelements. A POLICY-REF element can also have COOKIE-INCLUDE and COOKIE-EXCLUDE subelements which are used to isolate policies corresponding to cookies.

5.6. Translating APPEL Preferences into XQuery

We assume that a policy is either stored as an XML document in a native XML store, or there is a reconstruction view [6] [21] [22] that renders a P3P policy according to its original XML schema starting from a tabular representation of the policy. In either case, the XQuery that implements an APPEL rule will be the same. When using the reconstruction view, it is the query system's responsibility to accept the XQuery expressed over the XML view of the policy and transform it into an equivalent SQL query against the tabular representation chosen to store policies.

Aside from the fact that navigation in SQL is expressed with joins while navigation in XQuery is expressed using XPath [4], the translation of APPEL into XQuery generally resembles the translation of APPEL into SQL. A simplified version of the translation algorithm that only handles "or" and "and" connectives is shown in Figure 17. The full algorithm is given in Appendix B.

The main() function generates an XQuery *if* statement that returns the rule behavior if the condition expressed by the rule is met by the applicable policy.

The body of the rule is translated by the match() function. Lines 11 through 14 generate the XQuery code for matching attributes. Line 17 iterates through the subexpressions of APPEL expression e and uses a recursive call

```
if (document("applicable-policy")
   POLICY
    [STATEMENT
      [PURPOSE
        [admin OR
         contact [@required = "always"]
         ]]])
then
  return <block/>
```

Figure 18: XQuery Translation of Jane's Simplified APPEL Preference

to the match() function to create predicates for the subexpressions. The logical connective among these predicates is set by θ which can either be an "and" or "or" , depending upon the connective associated with e (line 18). XPath predicates are expressed using brackets. They restrict the matching elements in the policy. This is shown on line 19, where e.name() creates the name of the expression , followed by a predicate applied to attributes and subexpressions of e.

5.6.1 Translation Example

Figure 18 shows the translation of the simplified first rule of Jane's APPEL preference (Figure 12). The XQuery contains an *if* statement that returns <block> if the condition of the statement is met. The condition is expressed as an XPath which starts by selecting the applicable policy. The outer most restriction specifies that the POLICY element has to have at least one STATEMENT subelement. The next restriction is expressed on STATEMENT and selects statements that have at least one PURPOSE subelement. The restriction on PURPOSE is a disjunction of the predicates on two subelements "admin" and "contact". For the subelement "contact", its *required* attribute must also be set to the value "always" in order for this disjunct to be true. If the result of the XPath is not empty, the *if* condition is true, thereby implementing the simplified rule of Jane's preference.

6. Performance Experiments

We now present the results of our experiments to study the performance of our database implementation of P3P.

6.1. Experimental Setup

Our experiments measured the time to match a P3P policy with an APPEL preference, first using a native APPEL engine and then using a database engine. Both the APPEL engine and the database engine were run on a Windows NT 4.0 server with dual 600 MHz processors and 512 MB of memory.

The APPEL engine we used is available from the Joint Research Center (JRC) [17]. To the best of our knowledge, it is the only APPEL engine currently available in public domain. The database system we used was DB2 UDB 7.2. The policy database was created under DB2's default settings, with the application heap size set to 4 MB.

Preference	#Rules	Size (KB)
Very High	10	3.1
High	7	2.8
Medium	4	2.1
Low	2	0.9
Very Low	1	0.3
Average	4.8	1.9

Figure 19: JRC APPEL Preferences

To measure the performance of the alternative of translating APPEL preferences into XQuery and executing them against the XML policy view over privacy data tables, we used the XTABLE[4] prototype [6] [21] [22]. XTABLE was responsible for generating SQL from XQuery, which was then run against DB2.

We could not find any public-domain native XML store in which we could define and populate our data tables to run experiments. Although some public domain demonstrations of XQuery implementation are available (see [30]), they only run against canned databases.

6.2. Data Set

We used 29 P3P policies in our experiments. They were obtained by crawling the web sites of the Fortune 1000 companies looking for P3P policies. We found 29 companies with P3P policies, including companies such as AT&T, IBM, McGraw Hill, and Progressive Insurance Group. Sizes of these policies vary from 1.6 to 11.9 KBytes, with the average size being 4.4 KBytes. These policies contained a total of 54 statements (about 2 statements per policy on average).

We used 5 APPEL preferences in our experiments. These preferences were taken from the JRC site [17] and constitute their test suite. JRC designed these preferences for different levels of sensitivity for privacy: Very High, High, Medium, Low, and Very Low. Figure 19 lists the sizes of preferences and the number of rules they contained.

6.3. Performance Results

6.3.1 Shredding

We measured the time needed for shredding each of the 30 privacy policies and storing the shredded policies into privacy tables in DB2 as per the schema defined in Section 5.4. The average shredding time was 3.19 seconds, with the maximum and minimum being 11.94 and 1.17 seconds respectively. Since a policy changes infrequently, the lifetime cost of shredding can be considered negligible.

6.3.2 Matching

Figure 20 shows the performance of matching preferences against policies for the three implementations: native APPEL, SQL, and XQuery. Each preference was matched against every policy. The figure shows the average, maximum, and minimum in seconds for matching a preference

[4]The XTABLE prototype has also been referred to as the XPERANTO prototype in the past.

	APPEL Engine	SQL			XQuery
		Convert	Query	Total	
Average	2.63	0.08	0.08	0.16	1.65
Max	9.08	0.14	0.24	0.34	5.00
Min	0.98	0.04	< 0.001	0.04	0.30

Figure 20: Execution time for matching a preference against a policy (seconds)

Preference	APPEL Engine	SQL			XQuery
		Convert	Query	Total	
Very High	2.65	0.09	0.08	0.17	2.63
High	2.68	0.10	0.14	0.24	2.33
Medium	2.66	0.13	0.14	0.27	-
Low	2.60	0.06	0.03	0.09	1.51
Very Low	2.54	0.04	< 0.01	0.05	0.31

Figure 21: Per-preference-type execution times for matching a preference against a policy (seconds)

against a policy. For the SQL implementation, we separate the time needed for converting APPEL into SQL (conversion time) and the time needed for matching (query time). The total time is the sum of conversion and query times. The XQuery numbers include both the time for converting APPEL into XQuery, and the time taken by XTABLE to convert XQuery into SQL.

Figure 21 shows the performance numbers broken down per five preference types. For the XQuery implementation, we do not have numbers for the Medium preference. The XTABLE translation of the XQuery into SQL was too complex for DB2 to execute in this case.

The numbers shown in Figure 20 and 21 are the "warm" numbers. They reflect the time likely to be experienced in deployed systems. The system was warmed up by first matching an extra (artificial) preference and discarding this time. This factors out one-time costs such as the JVM loading the classes. The difference between the warm and cold average matching times was about 1.4 seconds for the native APPEL engine, 1 second for SQL, and 3 seconds for the XQuery implementation. For the SQL implementation, we stopped and restarted DB2 after matching each preference to avoid any advantage due to DB2 query caching.

Several conclusions can be drawn from these figures. First is the surprisingly good performance of the SQL implementation when compared to the native APPEL engine. We would have been satisfied if the SQL implementation came close to the APPEL implementation. But the SQL implementation turns out to be more than 15 times faster, even with the conversion time included in the SQL numbers. If we just compare the matching time, the SQL implementation is 30 times faster. The latter is a meaningful comparison as it is not unreasonable to think of a P3P deployment in which the preference generation GUI tool produces preferences as a set of SQL statements.

To understand this large performance difference, we profiled the APPEL engine. Before matching a preference against a policy, the APPEL engine first augments every data element in the policy with the corresponding categories predefined in the P3P base schema (see Section 5.4.6 in [8]). We found that this augmentation accounts for most of the difference in performance. In a client-centric architecture, the APPEL engine running in the client has to incur this cost for every preference checking. Our SQL implementation, on the other hand, does this expansion while shredding the policy into relational tables, and incurs no corresponding cost at the time of preference checking. Since a policy changes infrequently, the cost of shredding amortized over a large number of matchings of different preferences against a policy can be considered negligible.

We were hoping for a better performance from the XQuery alternative, particularly since the translation algorithm is simpler and the generated XQueries are easier to comprehend than the SQL queries. This performance gap points out that there are still untapped optimizations that XTABLE can perform in generating SQL from XQueries.

More important than the relative comparison is the absolute time needed for matching preferences against policies. Figures 20 and 21 show that the latency introduced by our SQL implementation for preference matching is more than acceptable for it to be used in practical P3P deployments.

7. Conclusion and Future Work

The following are the contributions of this paper:

- Identification of P3P as an important application area for database systems.
- Investigation of alternative architectures for implementing P3P.
- Proposal for a server-centric architecture based on database querying technology.
- Mapping of a P3P policy schema into a relational schema for storing policy data.
- Algorithms for translating privacy preferences expressed in APPEL into SQL as well as XQuery.
- Performance experiments showing that the proposed architecture has adequate performance for it to be used in practical deployments of P3P.

An interesting topic for future work would be to explore the use of database query languages for directly expressing and representing privacy preferences. In particular, it would be useful to identify the minimal subsets of SQL and XQuery needed for this purpose. Another direction for future research would be to develop and implement database mechanisms for ensuring that the privacy policies are indeed being followed.

Acknowledgments We wish to thank Dan Gruhl for the crawl of the Fortune 1000 web sites for P3P policies. Thanks are also due to Lorrie Cranor of AT&T for answering our questions regarding P3P and Privacy Bird.

References

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *28th Int'l Conference on Very Large Databases*, Hong Kong, China, August 2002.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Implementing P3P using database technology. In *19th Int'l Conference on Data Engineering*, Bangalore, India, March 2003.

[3] M. Benedikt, M. Fernandez, J. Freire, and A. Sahuguet. XML and data management. In *WWW-2002 Tutorial*, Honolulu, Hawaii, May 2002.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu, editors. *XQuery 1.0: An XML Query Language*. W3C Working Draft, April 2002.

[5] Business Week. *Privacy on the Net*, March 2000.

[6] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB Workshop*, Dallas, Texas, May 2000.

[7] S. Chaudhuri and K. Shim. Storage and retrieval of XML data using relational databases. In *VLDB Tutorial*, Roma, Italy, 2001.

[8] L. Cranor, M. Langheinrich, and M. Marchiori. *A P3P Preference Exchange Language 1.0 (APPEL1.0)*. W3C Working Draft, April 2002.

[9] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle. *The Platform for Privacy Preferences 1.0 (P3P1.0) Specification*. W3C Recommendation, April 2002.

[10] L. Cranor, J. Reagle, and M. Ackerman. Beyond concern: Understanding net users' attitudes about online privacy. Technical Report TR 99.4.3, AT&T Labs–Research, April 1999.

[11] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, pages 431–442, Philadephia, Pennsylvania, June 1999.

[12] The Economist. *The End of Privacy*, May 1999.

[13] European Union. *Directive on Privacy Protection*, October 1998.

[14] D. Florescu and D. Kossman. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[15] E. N. Hanson and J. Widom. An Overview of Production Rules in Database Systems. *The Knowledge Engineering Review*, 8(2):121–143, June 1993.

[16] IBM Tivoli Privacy Wizard. http://www.tivoli.resource_center/maximize/privacy/wizard_code.html.

[17] JRC P3P Resource Centre. http://p3p.jrc.it.

[18] Office of the Information and Privacy Commissioner, Ontario. *Data Mining: Staking a Claim on Your Privacy*, January 1998.

[19] References for P3P implementations. http://www.w3.org/P3P/implementations.

[20] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Effecient relational storage and retrieval of XML documents. In *WebDB Workshop*, Dallas, Texas, May 2000.

[21] J. Shanmugasundaram, J. Kiernan, R. Krishnamurthy, E. Viglas, J. Naughton, and I. Tatarinov. A general technique for querying XML documents using a relational database system. *SIGMOD RECORD*, 30(3), 2001.

[22] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, Roma, Italy, 2001.

[23] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, Edinburgh, Scotland, 1999.

[24] Time. *The Death of Privacy*, August 1997.

[25] A. Westin. E-commerce and privacy: What net users want. Technical report, Louis Harris & Associates, June 1998.

[26] A. Westin. Privacy concerns & consumer choice. Technical report, Louis Harris & Associates, Dec. 1998.

[27] A. Westin. Freebies and privacy: What net users think. Technical report, Opinion Research Corporation, July 1999.

[28] The World Wide Web Consortium. *Extensible Markup Language (XML)*. http://www.w3.org/XML.

[29] The World Wide Web Consortium. *P3P 1.0: A New Standard in Online Privacy*. Available from `http://www.w3.org/P3P/brochure.html`.

[30] The World Wide Web Consortium. *XML Query*. http://www.w3.org/XML/Query.

A. SQL Translations of Other Connectives

The translation algorithm in Figure 11 assumes that the APPEL connective was either "or" or "and". We now generalize the translation algorithm to handle other connectives: "non-or", "non-and", "or-exact" and "and-exact". The generalization mainly consists of an extended connect() function and a new others() function. The latter is used in the translation of "or-exact" and "and-exact" to ensure that the policy does not contain excess evidences. The full algorithm is shown in Figure 24.

A.1. Non-or and Non-and

The "non-or" and "non-and" connectives negate the matching conditions specified by the "or" and "and" connectives. Line 21 in Figure 24 achieves this by prefixing a NOT to the SQL generated for "or" and "and".

A.2. Or-exact and And-exact

The "or-exact" and "and-exact" connectives are more restrictive than the "or" and "and" connectives. For an evidence to match an expression, the "or" ("and") connective requires that at least one (all) subexpressions of the expression are matched by some subevidences of the evidence. The "or-exact" ("and-exact") has the same requirement as "or" ("and"), but additionally requires that there is no subevidence that does not have a matching subexpression. For example, the APPEL expression Figure 22 does not match the policy evidence in Figure 23 because the second DATA subevidence in the policy does not have any matching APPEL subexpression. Note that these two would match if the connective were an "or".

```
<DATA-GROUP appel:connective = "or-exact">
  <DATA ref="#user.name"/>
  <DATA ref="#user.gender"/>
</DATA-GROUP>
```

Figure 22: An APPEL expression with or-exact connective

```
<DATA-GROUP>
  <DATA ref="#user.name"/>
  <DATA ref="#user.home-info.postal">
</PURPOSE>
```

Figure 23: A fragment of a policy

The above requirement is enforced in Figure 24 by providing an additional condition in the translation. This condition is expressed as a double negation (line 23): there does not exist any subevidence that does not match any subexpression. The generated translation consists of a NOT followed by a code block comprising the second negation ("exists a subevidence that does not match any subexpression"). This code block is generated by the others() function.

Figure 25 shows, as an example, the SQL translation for the APPEL expression given in Figure 22.

```
1  String main (Rule r) {
2      return "SELECT" + r.behavior() +
3             "FROM"    + applicablePolicy() +
4             "WHERE"  + connect(r);
5  }

7  String connect(Expression e) {
8      String sqlAttr = genAttr(e);
9      let θ = e.connective();
10     String sqlSub;
11     for each subexpression se of e do
12         sqlSub += "EXISTS (" +
13                   path(se) + "AND" + connect(se) + ")";
14         // orAnd() returns the "or" or "and" part of θ
15         sqlSub += θ.orAnd();
16     String sql = sqlAttr "AND (" + sqlSub + ")";

18     if θ is "or" or "and" then
19         return sql;
20     else if θ is "non-or" or "non-and" then
21         return "NOT (" + sql + ")";
22     else // θ is "or-exact" or "and-exact"
23         String sqlNoOther = "NOT (" + others(e) + ")";
24         return sql + "AND" + sqlNoOther;
25 }

26 String genAttr (Expression e) {
27     String sql;
28     for each attribute attr of e do
29         sql += attr.name() + "=" + attr.value() + "AND";
31     return sql;
32 }

33 // Expressions in P have the same name
34 String genExistsBody (Expression[ ] P) {
35     let a = the first expression in P;
36     String sql = path(a) + "AND NOT (";
37     for each expression e in P do
38         sql += connect(e) + "OR";
39     return sql + ")";
40 }

42 String path (Expression e) {
43     return sql = "SELECT *" +
44                  "FROM"      + e.name() +
45                  "WHERE"     + e.foreignKey() + "="
46                              + e.parent().primaryKey();
48 }

49 String others (Expression e) {
50     String sql;
51     Expression[ ] S = all listed subexpressions of e;
52     partition S by subexpression names;
53     for each partition P do
54         sql += "EXISTS (" + genExistsBody(P) + ") OR";
55     Expression[ ] U = all unlisted subexpressions of e;
56     for each subexpression se in U do
57         sql += "EXISTS (" + path(se) + ") OR";
58     return sql;
59 }
```

Figure 24: Full Algorithm for Translating an APPEL preference into an SQL Query

```
SELECT *
FROM Datagroup
WHERE Datagroup.policy_id = Statement.policy_id AND
   Datagroup.statement_id = Statement.statement_id AND
   ( EXISTS (
      SELECT *
      FROM Data
      WHERE Data.policy_id = Datagroup.policy_id AND
         Data.statement_id = Datagroup.statement_id AND
         Data.datagroup_id = Datagroup.datagroup_id AND
         Data.ref = '#user.name' )
   OR
   EXISTS (
      SELECT *
      FROM Data
      WHERE Data.policy_id = Datagroup.policy_id AND
         Data.statement_id = Datagroup.statement_id AND
         Data.datagroup_id = Datagroup.datagroup_id AND
         Data.ref = '#user.gender' )
   )
   AND
   NOT (
   EXISTS (
      SELECT *
      FROM Data
      WHERE Data.policy_id = Datagroup.policy_id AND
         Data.statement_id = Datagroup.statement_id AND
         Data.datagroup_id = Datagroup.datagroup_id AND
         NOT ( Data.ref = '#user.name'
            OR
            Data.ref = '#user.gender' ) )
)
```

Figure 25: SQL Translation of "Or-exact"

B. XQuery Translations of Other Connectives

Figure 26 shows the full algorithm for translating an APPEL rule into an XQeury. This algortithm handles all the connectives. Figure 27 shows the fragment of the XQuery generated for the APPEL expression given in Figure 22. The relative simplicity of the XQuery translation stems from the availability of *for all* in XQuery, whereas SQL requires the use of NOT EXISTS NOT to simulate *for all*.

```
1   String main (RuleBody r) {
2       String xq =
3           "if (document(" + applicablePolicy()+ ")[" +
4           matchSubexpressions(r) +
5           "]) then <" + r.behavior() + ">";
6       return xq;
7   }

8   String match (Expression e) {
        // match attributes of e
9       String xpAttr;
10      for each attribute attr of e do
11          xpAttr += "@" + attr.name() + "=" " + attr.value() + """;
12          xpAttr += "AND";
        // match subexpressions of e
13      String xpSub = matchSubexpressions(e);
14      return e.name() + "[" + xpAttr + "AND (" + xpSub + ")]";
15  }

16  String matchSubexpressions (Expression e) {
17      String xpSub;
18      let θ = e.connective();
        // θ.orAnd() returns the "or" or "and" part of θ
19      for each subexpression s of e do
20          xpSub += match(s) + θ.orAnd();
21      if θ is "non-or" or "non-and" then
22          xpSub = "NOT (" + xpSub + ")";
23      else if θ is "or-exact" or "and-exact" then
24          xpSub = "(" + xpSub + ") AND" + noOther(e);
25      return xpSub;
26  }

27  String noOther (Expression e) {
28      String xp = "every $s in ./* satisfies (";
29      for each subexpression s of e do
30          xp += "$s/self::" + match(s) + "OR";
31      return xp + ")";
32  }
```

Figure 26: Full Algorithm for Translating an APPEL preference into an XQuery

```
DATA-GROUP
  [( DATA[@ref = "#user.name"]
     OR
     DATA[@ref = "#user.gender"]
   )
   AND
   all $s in ./* satisfies (
     $s[name() eq "DATA"][@ref = "#user.name"]
     OR
     $s[name() eq "DATA"][@ref = "#user.gender"] )
  ]
```

Figure 27: XQuery Translation of "Or-exact"