PALLAS: A querying interface for pervasive computing using handheld devices

Stavros Polyviou
University of Cyprus
polyviou@ucy.ac.cy

Paraskevas Evripidou
University of Cyprus
skevos@ucy.ac.cy

George Samaras
University of Cyprus
cssamara@ucy.ac.cy

Abstract

In this paper we describe PALLAS (PALm Logic-based Algebraic Search), a prototype architecture, design and implementation for a search interface on a portable digital assistant. PALLAS is part of the UbAgent, a middleware infrastructure that provides an efficient way of handling the back-end processing of Pervasive Computing: accessing information from distributed databases, computation processing, and location specific services. The goal of the PALLAS' design is to offer the flexibility of advanced searches by exploiting the metadata and inferencing capabilities of the semantic web, whilst maintaining ease-of-use to a level such that it would be accessible to an end-user without expansive technical knowledge, much like the typical users of the world-wide web today. A prototype of PALLAS have been developed and tested for the PALM OS. The pervasive computing test bed used was the UbAgent infrastructure.

1. Introduction

Pervasive computing is the trend towards increasingly ubiquitous, connected computing devices in the environment [1]. Pervasive/Ubiqitous services are not about browsing the Web on a cell phone or a PDA. It is about providing personalized services that are highly sensitive to the immediate environment and needs of the user. Real world will become a Marketplace for Services, Products and Information. Project UbAgent [1] is developing a Mobile Agent based middleware for the Ubiquitous/Pervasive Computing. This middleware focuses in providing access to information, scalable computing, and location-based services. UbAgent's middleware infrastructure provides an efficient way of handling the back-end processing of Pervasive Computing: accessing information from distributed databases, computation processing, and location specific services. UbAgent is a mobile agent based middleware for Pervasive computing. It is build on top of commercially available mobile agent systems. The second tier components are the TRAcKER a location management system, the Taskhandler framework and a Unified Message System for Mobile agents. At tier three we have PaCMAn, a web-based meta computer, the DBMS-agent system for distributed information retrieval and the DVS system for the creating of personalized views. At present UbAgent deals with the back-end of Pervasive applications; provides the services and it does not deal with real life user interface, i.e. to interpret the user needs and wishes.

As a departure point for the design of Pallas, we used a visionary scenario published in the 'Scientific American' [2] by Tim Berners-Lee, James Hendler and Ora Lassila. We selected this scenario as it highlights many of the aspects of the semantic web and what it promises to deliver. It also originates from a highly reliable source, namely the designers of the semantic web itself. Our goal was to deliver a design that could accommodate all aspects of the scenario, but that would be general enough to be applied to a variety of different situations.

The scenario refers to two siblings, Pete and Lucy, who need to make arrangements for physical therapy sessions for their mother. To that end, they need to locate a number of providers that can offer the required treatment, are covered by their mother's health plan, are located conveniently close to both their homes and can be scheduled at times that can be accommodated by their busy schedules. To achieve this, they make use of two key features of the semantic web, namely the ability to locate and retrieve data from published databases and the ability to locate and invoke web services, both of which are described in an article by Hendler, Berners-Lee and Miller [Hendler et al 2002].

For our prototype implementation we did not focus on the infrastructure issues such as the implementation of RDF processors or the creation of suitable ontologies for

Proceedings of the IEEE/ACS International Conference on Pervasive Services (ICPS'04)
0-7695-2535-0/04 $ 20.00 IEEE

describing the medical or other data required by the scenario. We assumed that such infrastructure existed and simulated its function, thus focusing on the user interface and the communication protocol between the application and the semantic web.

2. Architecture

The scenario indicates the existence of a number of published databases on the semantic web. These include a database maintained by the mother's physician, which stores patient data such as any treatments the patient is undergoing, the health plans the patient is enrolled in and so forth. The medical insurance company also stores information about the plans it offers, including which treatments are covered by each plan. Health care providers can be found through a listing service. Each health care provider publishes the kinds of treatments they can offer, as well as all medical plans they participate in. There is also a rating service that rates health care providers on their service and reliability. In addition to these databases, Pete and Lucy's schedules are made available on the semantic web. Finally, a mapping web service that provides functions such as producing driving instructions or calculating distances between addresses is used.

Each of these databases (or services) is published at a known URL. The end user can access the URL pretty much the same way one accesses a web site today: either by memorizing it or by following a link on some known website, such as a portal or search engine. The user's agent can then retrieve an RDF description of the data published on that URL. The description uses industry-wide ontologies; for example the medical industry could have agreed ways of describing treatments or medical plan provisions. Calendar information could be published using iCal [Payne et al, 2002]. Geographical data would be described using an ontology established by the Geographical Information Systems (GIS) industry and so forth.

Furthermore, we assume that all published data is in relational form (tables and columns) or can be easily transformed to relational form. Moreover, we assume that the database management systems that manage the data and contain the schemas published on the semantic web have adequate support for distributed database management, i.e. allowing for joins across multiple disparate databases to be combined in a relatively efficient manner.

Using RDF's inferencing capabilities, the agent can establish equivalences and relationships between the data stored in disparate databases. PALLAS is running on a PDA, while the user's machine acts as a server, dispatching mobile agents where necessary. Recursive software's Voyager was used for the implementation of the agents. The PDA handles less processor-intensive tasks and communicates with the server over a wireless network using Bluetooth wireless networking, although other wireless technologies could be used.

The communication between the PDA application and the server is as follows:

- The user enters a URL in PALLAS. The request arrives at the server, which retrieves the database or service RDF description and creates a representation of the database schema, which is sent to the PDA.
- Subsequent requests for URLs received by the server require additional processing. The server consults the RDF descriptions of the newly added and existing databases, and establishes type equivalences and primary/foreign key relationships between tables. The revised database schema is sent back to the PDA. This is essentially a 'virtual' schema, designed to mask the differences between various databases and the fact that the data resides in multiple databases.
- Once the query is submitted by the user, it is sent to the server (in the form of an SQL query). The query is decomposed by the server and dispatched to the appropriate databases. Once the results sets arrive back at the server, they are combined and transmitted to the PDA.

PALLAS finally displays the results, and allows the user to take action if necessary on the results.

3. Design rationale

3.1. Goals

In designing PALLAS, several goals had to be met:

- Conformance to the Palm OS user interface guidelines
- Efficient use of the small available screen space
- Ability to perform flexible and advanced queries involving joins and predicate logic
- Small memory footprint
- Should require little or no knowledge of database concepts to be used
- Should be easy to translate to other languages

These goals were derived both from the type of device the interface would be deployed on (a PDA running the Palm operating system), the requirements of the sample scenario and the target audience.

3.2. Motivation

Currently, the most widespread end-user interface for performing advanced queries on a relational database is the Structured Query Language (SQL). Early on we decided that SQL would be an unsuitable basis for our interaction model, both for usability and technical reasons.

SQL is a programming language, and as such presents a text-based interface to the end-user. A text-based approach is of course unsuitable at a time when most users prefer GUI-based approaches rather than having to memorize command languages. The heavily English language-based SQL syntax also presents problems when targeting an international audience.

Furthermore, SQL imposes a non-logical sequence of steps. The first clause in an SQL query specifies the columns that are to be returned. The source of those columns is identified in the second step, even though it is logically the first step. In fact, the selection of columns is actually the last step performed by the DBMS. Consequently, the ordering of SQL clauses forces users into a mind set where they mistakenly think that the columns available for use in subsequent clauses, such as WHERE and GROUP BY, are restricted to the list of columns specified in the first clause.

SQL also violates the relational model by permitting the existence of duplicate rows in relations [3], which are sometimes called corrupted relations. The same operations applied to the same set of corrupted relations in a different order may lead to different results, a problem that does not occur as much with proper relations. This means that users have to be aware of the order of their actions and its ramifications. Moreover, the appearance of duplicate rows in a relation carries no useful information. As Codd states: "There does not exist a precise, accepted, context-independent interpretation of duplicate rows in a relation".

Codd also criticizes SQL's nesting feature, also known as a subquery. Subqueries are queries that appear inside predicates, i.e. expressions that restrict the kinds of rows that appear in the result set. The problem lies in the fact that in order for such nested queries to be performed, they must first be translated into a non-nested form. At the time of Codd's writing, there was no generally agreed upon method for translating such nested queries, nor a rigorous proof on the translatability of all the possible kinds of nested queries that can appear in SQL. Even if such method or proof existed, the question remains whether the results of queries involving nested queries are the ones expected by the users who formulate them.

Another potential problem resulting from the design of SQL, is that it permits the generation of Cartesian products. According to Codd, "[the] Cartesian product should be de-emphasized, and used primarily as a tool for explanatory or conceptual purposes". This is because "the Cartesian product contains no more information than its components contain together", yet it consumes many more resources (disk space, bandwidth, memory) than its components consume together. A novice SQL user is very likely to produce a Cartesian product, as they are the default when it comes to multi-table queries in SQL.

Our purpose of course is not to advise against the use of SQL in relational database management systems, but rather to point out its unsuitability as a model of interaction for our particular purpose. What we are suggesting is of course a GUI-based rather than a text-based solution, however what we feel distinguishes our approach from that of commercially available query builders, is that the latter tend to follow the SQL model of interaction and in most cases surface or even resort to SQL code for some aspects of the query.

This means that such GUI-based builders do little in the way of addressing the aforementioned problems, such as eliminating duplicate rows or preventing Cartesian products from happening. This is probably due to the fact that their target audience is users with some SQL background, so any deviations from the SQL task flow could be perceived negatively. A corollary of this approach is that query builders often don't go quite far enough in supporting advanced querying features, as strict adherence to SQL syntax becomes an impediment as queries become more complex. Our challenge is therefore to produce a GUI-based query builder for non-SQL users that is as powerful and flexible as SQL but easier to use than existing GUI-based query builders, and all of this in a PDA form factor!

3.3. Solution

In light of the above, we searched for an alternative basis for our interaction model, and arrived at the conclusion that relational algebra was the most suitable option. Relational algebra, much like traditional algebra, uses operators that combine one or more operands into a result. The operands and results in relational algebra are

relations, rather than numbers as in traditional algebra. Relational algebra also makes heavy use of predicate logic. Although Codd advocates the use of four-level logic, we opted for three-level logic, which is implemented in most commercial relational database management systems.

Unlike SQL, relational algebra does not allow Cartesian products, except in some extreme cases of some relational operators. This eliminates a large class of problems arising from improper use of SQL by inexperienced users. It also allows for greater flexibility in the order of steps in a query. This is because in relational algebra corrupted relations are not allowed, so the same operators in different arrangements yield the same results, relieving the user form the burden of having to make explicit decisions about the sequence of operations in a query.

A third advantage is derived from relational algebra's operational closure, the fact that the results of operations can be used as operands in other operations. This allows "an interrogator to conceive his or her ongoing sequence of queries based on the information gleaned to date" [3]. In comparison, SQL has a lot of special cases which do not allow the incremental creation of a query. The ability to perform steps in a flexible sequence and to incrementally build a result set is highly compatible with the principles of modern direct manipulation graphical user interfaces.

Finally, relational algebra does not support the nesting of queries (i.e. subqueries), a feature that is difficult to use, analyze and support in a graphical user interface. The relational algebra equivalent to the most commonly used form of subquery is the relational division operator.

What should be clarified at this point is the fact that we do not intend to surface relational algebra in the user interface, but to exploit some of its desirable qualities in our design, namely:

- The avoidance of Cartesian products which protects novice users
- The relative immunity to the sequence of operators which imposes a lesser burden on user's memory
- The ability to incrementally create queries afforded by operational closure
- The avoidance of nested queries
- Its independence from the grammar of a particular human language (unlike SQL)

In addition to avoiding the deficiencies of SQL and exploiting the advantages of relational algebra, our design was based on the observation that most inexperienced database users have difficulty in formulating joins. Most joins are based on primary-foreign key relationships even though the relational model allows joins to occur on non-key columns as well (in fact, this flexibility is touted by Codd as an advantage over pre-relational database systems).

We therefore decided to incorporate a key-based join discovery feature into PALLAS. This feature examines the primary/foreign key relationships in the virtual schema and returns all possible join paths between two tables. The paths can be direct or indirect (i.e. involving other tables), and their length can be restricted. In the future we plan to extend this feature (and the design) to support non-key joins as well; rather than examining primary/foreign key relationships, the domains from which the columns of two tables draw their values from could be examined instead. In order for this feature to be effective however, a high granularity in the definitions of domains would be required.

4. User interface description

4.1. Introduction

The PALLAS user interface consists of three main sections: the search page, the criteria page, and the results page. Each can be accessed by pressing one of the available toggle buttons (figure 2). The search page is where users select the source of their data (i.e. the database tables). The criteria page is where users specify predicates used for filtering the result set. Finally, the results page contains options regarding the display of the result set.

4.2. The search page

The initial interface screen is very simple; beyond the ability to switch between the different pages of the interface, the user has the option of tapping on the popup trigger (the word 'Find' with the downwards arrow next to it) or the selector trigger (the dotted box).

The popup trigger is used to select the way data between different tables will be combined. The first popup trigger includes the options 'Find' and 'Find all'. Subsequent popup triggers contain the options 'and corresponding' and 'and all'. The corresponding options are equivalent across all popup triggers; the wording on the first popup trigger differs purely for readability reasons.

The selector triggers are used for invoking the form shown in figure 2. Here, the user can enter the URL of the

database they wish to access on the semantic web. The '…' button invokes a special form containing bookmarks or the recently used URLs (not discussed here for brevity).

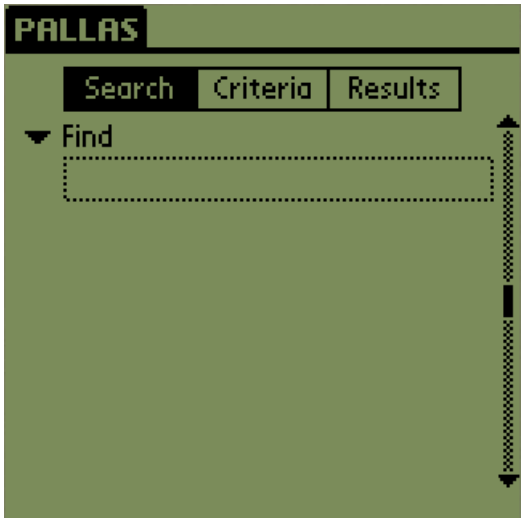

Figure 1: the initial interface screen

Once the URL is specified, it is sent to the server which dispatches an agent to the database host, retrieves the RDF description of the database schema, integrates the schema fragment into the virtual schema, and sends this additional information to the PDA, which revises its own schema model.

The published tables for which the user has access are shown on the list to the left, while the columns contained in each table are shown on the left for the highlighted table. Columns can be included in the result set by checking the corresponding checkbox. The primary reason for displaying the columns is for assisting the user in making an educated choice for which tables to include in their query. After all, what the user is in essence interested in is the data stored in the columns, not the names of the tables. Although the results page is where users should make all decisions regarding the content and format of their result set, it was thought that since the column names are displayed anyway, allowing users to make preliminary choices at this point would be convenient and a more efficient use of the displayed information.

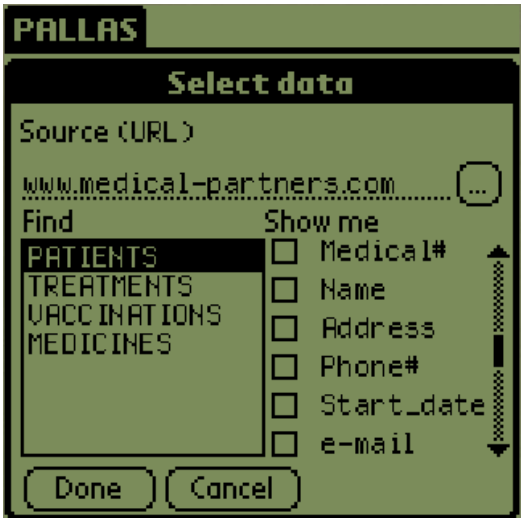

Figure 2: the 'Select data' form

The list of displayed tables is not necessarily identical to the list of tables in the schema model. The interface filters out the so-called 'intersection tables'. An intersection table is used for capturing a many-to-many relationship between two tables. It was thought best not to tax the user with the burden of remembering or discovering whether two entities have a one-to-many or a many-to-many relationship in order to properly join them. The join discovery feature will transparently include any necessary intersection tables in the query. If however the intersection table carries some additional information with regards to the relationship itself, then it is displayed in order to allow users to include that information in their result set.

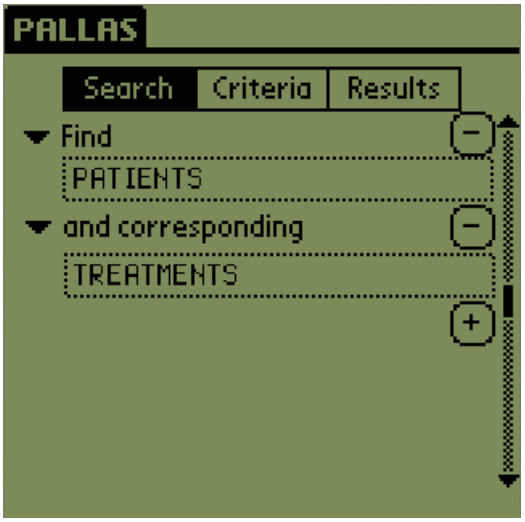

Figure 3: the search page

A second type of filtering to the table list occurs based on the tables that have already been selected; only the tables that can be joined to at least one of the already selected tables are displayed. Since non-key based joins are currently not supported, allowing users to include non-key related tables to the query would lead to the generation of Cartesian products. It is of course possible that there is more than one way to join an additional table to the list of existing tables. In such as case, a special form, shown in figure 4, appears after the 'Done' button is pressed, allowing the user to select which correspondences they wish to include.

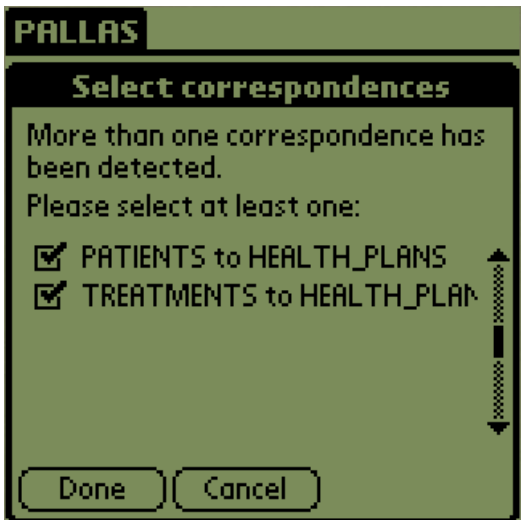

Figure 4: the 'Select correspondences' form

As mentioned earlier, the popup triggers affect the way data from different tables is combined. Each time a table is added to the query, the join discovery algorithm is invoked to generate the necessary conditions for joining the new table to the existing tables, namely whether to use an inner or outer join. There are four types of joins: the inner join, left outer join, right outer join and full outer join. The interface has been designed in such a way so as to allow users to perform all four types of joins in an easy and natural way.

In the example shown in figure 3, the user has performed an inner join, in other words the query will return only the patients that are undergoing treatments and the treatments that are applied to at least one patient. If the user wanted to obtain all the treatments regardless of whether any patient is currently undergoing the treatment, then they simply select the 'and all' option from the popup trigger. Had the order of the tables been reversed, the user would select 'Find all' from the first popup trigger to achieve the same effect. SQL in comparison requires the terms "LEFT OUTER JOIN' and 'RIGHT OUTER JOIN' in such a case, in other words it uses positional terms. This means that the user must first associate the table with a position (left or right) before selecting the right term, as opposed to the more direct approach of associating the high level goal of obtaining all data in a table with the table itself. Full outer joins are just as easily created by selecting 'Find all' and 'an all' from the popup triggers, which reads quite naturally as 'Find all patients and all treatments' as opposed to 'SELECT DISTINCT * FROM PATIENTS OUTER JOIN TREATMENTS ON …'.

Additional tables are added by pressing the plus (+) button which follows the last completed table entry. The user can remove a table from the query by pressing the minus (-) button, or can replace a table with another by tapping on the selector trigger and selecting a different table on the 'Select data' form. Both of these actions could have significant ramifications to the query, as they removal or replacement of a table from the query might lead to the creation of Cartesian products, as some tables might become 'detached' from the join graph. In such as case, either new join paths are discovered by the join discovery algorithm, or the user is asked whether they wish to discard some additional tables as well, for which there are no longer any correspondences with the other tables.

4.3. The criteria form

Once the list of tables has been specified, the user can proceed to the second part of the interface, namely the criteria form. This is where the user can specify a set of predicates connected with logical operators that filter the rows of the joined tables. This part of the interface corresponds to the WHERE clause in SQL.

This is one of the most complicated parts of the relational model, since predicates can be combined in arbitrary ways to an arbitrary nesting level using logical operators and parentheses. Moreover, the meaning of the logical operators is not always clear to end users. According to Jakob Nielsen [5] users generally do a very poor job at formulating Boolean queries as the correspondence between the logical operators 'AND' and 'OR' does not always coincide with the meaning of their query expressed in plain English (for example 'find all cats *and* dogs' translates into a Boolean query using the OR operator).

In order to alleviate this problem whilst maintaining as much of the flexibility and expressive power of predicate logic as possible, it was thought best to not surface the logical operators in the interface at all, but rather use short

descriptions of what the operators actually do. Each predicate is essentially a criterion that the returned rows (records) must satisfy. Two or more criteria connected with AND operators means that only the rows that satisfy all the criteria will be returned. If instead they are connected with OR operators, then all rows that satisfy at least one criterion will be returned. Unlike the words 'AND' and 'OR', such descriptions are unambiguous. Whether these descriptions are actually more accessible to users than logical operators however, needs to be verified through user testing.

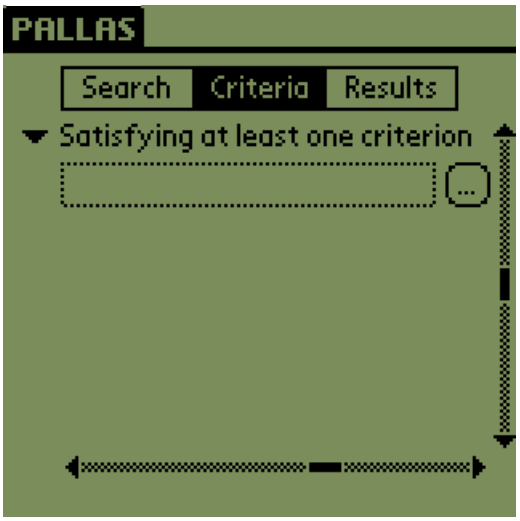

Figure 5 the initial criteria page interface

The initial criteria form interface is shown in figure 5 and an example with some criteria in figure 6. The appearance of the interface is quite similar to that of the search page. Selector triggers are used to determine how the criteria are to be combined. 'Satisfying at least one criterion' connects all predicates with OR operators, whereas 'Satisfying all criteria' connects them with AND operators. Inverted versions of these options also exist on the list ('Failing at least one criterion', 'Failing all criteria'). Not surfacing the operators in the interface not only avoids the aforementioned usability problems, it also reduces clutter and preserves screen space.

Pressing the '…' button next to each criterion, invokes the form shown in figure 7. This allows users to add or remove criteria. It also allows users to insert groups of criteria into the query, essentially creating parenthesized groups of criteria. Such groups appear indented with respect to the containing group, and begin with a popup trigger, just like the original criteria group. This creates the need for a horizontal scrollbar, to allow users to scroll to a group at an arbitrary nesting level.

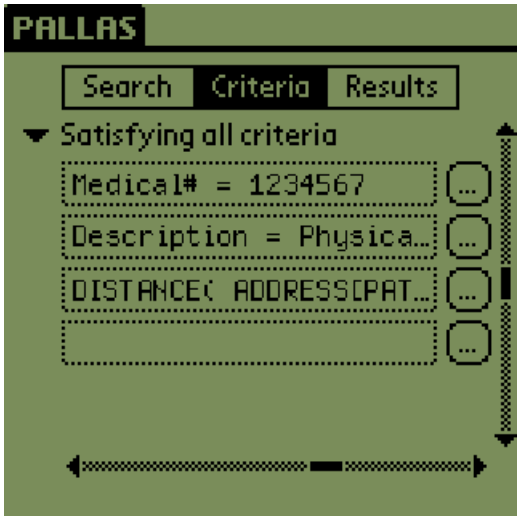

Figure 6: the criteria page with criteria added

An example of a nested criterion group is shown in figure 8. In this example the corresponding logical expression is 'Medical# = 1234567 AND Description = 'Physical therapist' AND (City = 'San Francisco' OR City = 'San Jose')'.

Figure 7: the 'Criterion options' form

Two further advantages of this approach is that users do not need to remember the precedence order of logical operators, as parentheses are added transparently whenever they are required. Also, in common cases such as combining a group of criteria using the same logical operator, less space and fewer steps are required than having to selected or type an operator between each predicate.

Figure 8: Example of nested criterion group

Tapping on the selector triggers allows the editing of a predicate, using the form shown in figure 9. The first popup trigger allows the user to select from a list of available columns. The second popup trigger includes the different types of supported predicates: comparison, range comparison, pattern match and missing test. The interface is modified depending on the user's selection. The remaining versions of the 'Edit criterion' form will not be discussed for reasons of brevity.

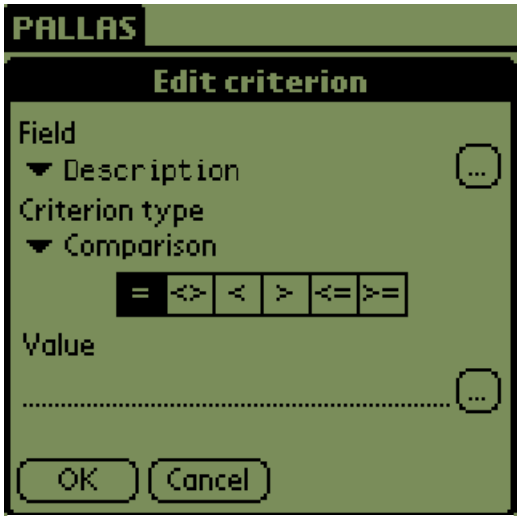

Figure 9: the 'Edit criterion' form

For a simple comparison, the user can select a comparator using the push buttons, and can enter a literal value to compare the column to using the field. The '…' button invokes the 'Create expression' form shown in figure 10. As the name implies, this form allows the creation of arbitrary arithmetic expressions involving literal values, function calls and database columns. Designing an expression builder involves significant challenges, as expressions can be nested to an arbitrary level. For example when invoking a function, each argument passed to the function is itself an expression, as are the operands of any arithmetic operator.

A text field is provided for the user to enter their expression in textual form. The field in this particular case however, is supported by a parse-as-you-type parser which is used for handling the creation, deletion and editing of placeholders. A placeholder is a piece of text that should be created, edited and deleted as a unit, rather than character by character. If for example the expression contains a function call, the function signature should be added and removed as a unit, i.e. removing the function call from the expression involves removing the expressions used as arguments for the function as well. Allowing the user to selectively erase part of the function name or say the closing parenthesis of a function call would most likely lead to an invalid expression, as would the entering of part of the function name and so forth.

Placeholders can also be used as guides. For example, when calling a function the necessary parameters need to be provided. Since the parameters are expressions themselves, this could lead to an arbitrary nesting of expression builders.

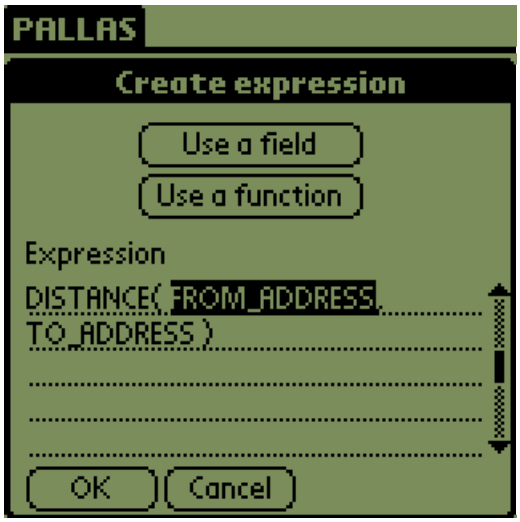

Figure 10: the 'Create expression' form

Placing placeholders for each parameter alleviates this problem; the placeholders provide guidance with respect to the number, type and purpose of the parameter. Since they are selectable and replaceable in their entirety, it is very easy for the user to highlight a placeholder and replace it

with an expression, all from within the original expression builder form.

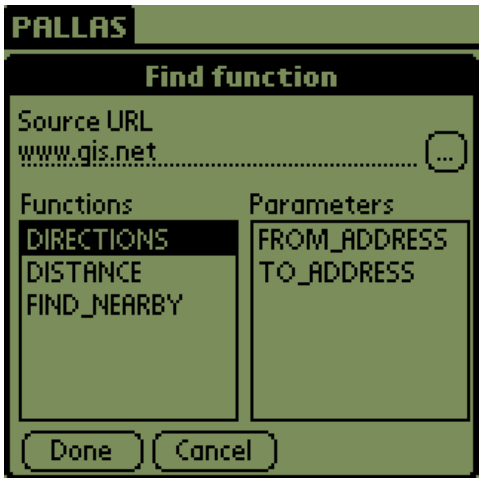

Figure 11: the 'Find function' form

Thus the two forms shown in figures 11 and 12 can be used to insert placeholders at the current cursor position. Note that the 'Find function' dialog allows the entering of a URL where the web service publishing the function resides. The 'Find field' dialog filters the list of available columns based on the domain of the argument represented by the selected placeholder (if any).

Figure 12: the 'Find field' form

4.4. The results form

The results from is where the user decides how the returned data will displayed and initiates the search. An example is shown in figure 13.

The user can select the columns they want to see in their results set by checking the corresponding checkbox. The 'Arrange…', 'Group…' and 'Sort…' buttons invoke other forms used for arranging the order in which the columns should appear, whether the displayed rows should be grouped and sorted based on the contents of the columns.

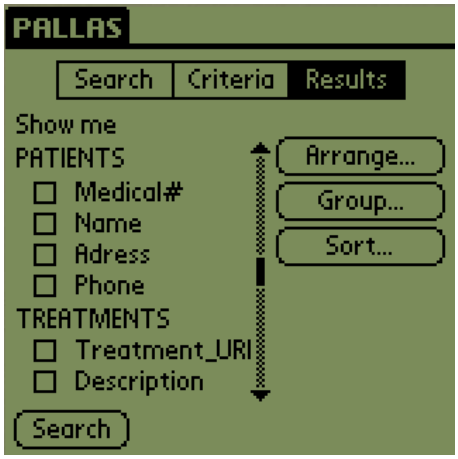

Figure 13: the results form

5. Prototype test bed.

Pallas was tested with the scenario described in the Introduction. The test bed used was the UbAgent Infrastructure [1].

The queries are performed over a mediated schema. For this example we have assumed that each schedule of the people involved is exported in a database, a feature supported by the popular calendar applications. Each time PALLAS need to fetch information from database it sends the relevant SQL statement to the UbAgent. UbAgent dispatches a DBMS agent which travels to the destination machine and performs the query using JDBC.

The next after schedules of the siblings and the schedules of the health providers are fetched by the mobile agents is to find a schedule that satisfy all the constrains. UbAgent's web-based meta-computer, called PaCMAn, was used to compute the best scheduled for all the participants. PaCMAn uses a number of mobile agents. Each agent supports the basic communication and synchronization tasks of the classical parallel worker assuming the role of a process in a parallel processing application.

6. Conclusion and future work

The semantic web will usher a new era in the publication, discovery, dissemination and combination of information, facilitating interoperability and communication among disparate database systems and application servers. A powerful yet usable user interface that will allow end users to harness the semantic web's power however, should be devised. We have described here our initial attempt at designing and implementing such an interface and we hope that some of the ideas presented here will provide departure points for future efforts and discussions on the subject.

The proposed design delivers a powerful search interface, offering such advanced features as full support for predicate logic, complex arithmetic expressions, web service integration and a powerful join discovery feature. All of this is achieved in a small form factor and with significant ease-of-use benefits over the dominant search interface, namely the language SQL.

There remain however some areas where the proposed design needs to be extended, in order to match SQL in flexibility and power and to better support the relational model. These areas are:

- Support for non-key based joins.
- Support for UNIONs, INTERSECTIONs and DIFFERENCEs.
- Inclusion of the relational division operator.
- Ability to save queries and use them as sources in subsequent queries.
- Usability improvements based on user testing.

We are confident that building on the current foundation of our research, we should be able to deliver a design that delivers the full power and flexibility of SQL whilst avoiding some of its pitfalls and design flaws. More importantly, all of this could be delivered in an easy to use interface that can scale down to a handheld device such as a PDA.

7. References

[1] G Samaras and P Evripidou "UbAgent: A mobile Agent Middleware Infrastructure for Ubiquitous/Pervasive Computing". Third International Conference on Intelligent System Design and Applications Conference 2003, August 10-13, 2003,Tulsa, Oklahama, USA

[2] Berners-Lee, Tim, Hendler, James and Lassila, Ora "The semantic web", Scientific American, May 2001 www.sciam.com/print_version.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21

[3] Codd, Edgar "The relational model for database management: version 2", Addison-Wesley, 1990 ISBN 0-201-14192-2

[4] Hendler, James, Berners-Lee, Tim and Miller, Eric "Integrating Applications on the Semantic Web", Journal of the Institute of Electrical Engineers of Japan, Vol 122(10), October 2002, p. 676-680

[5] Nielsen, Jakob "Search Usability", useit.com Alertbox, July 15 1997 www.useit.com/alertbox/9707b.html

[6] Payne, Terry, Singh, Rahul and Sycara, Katia "Calendar agents on the semantic web", IEEE Intelligent Systems, May/June 2002, p. 84-86