A Generic Persistence Model for (C)LP Systems (and two useful implementations)

J. Correas
J. M. Gómez
M. Carro
D. Cabeza
M. Hermenegildo

TR Number CLIP3/2003.1(2004)

A Generic Persistence Model for (C)LP Systems (and two useful implementations)

Technical Report Number: CLIP3/2003.1(2004)
August, 2003

Authors

J. Correas *et al.*
CLIP (Computational Logic, Implementation, and Parallelism) Group.

Keywords

Prolog, External Database, Persistent Storage, Media Independence

Acknowledgements

This work has been supported in part by the European Union IST program under contract IST-2001-34717 “Amos” and IST-2001-38059 “ASAP” and by MCYT projects TIC 2002-0055 “CUBICO” and HI2000-0043 “ADELA.” M. Hermenegildo is also supported by the Prince of Asturias Chair in Information Science and Technology at UNM.

Abstract

This paper describes a model of persistence in (C)LP languages and two different and practically very useful ways to implement this model in current systems. The fundamental idea is that persistence is a characteristic of certain dynamic predicates (i.e., those which encapsulate state). The main effect of declaring a predicate persistent is that the dynamic changes made to such predicates *persist* from one execution to the next one. After proposing a syntax for declaring persistent predicates, a simple, file-based implementation of the concept is presented and some examples shown. An additional implementation is proposed which stores persistent predicates in an external database. The abstraction of the concept of persistence from its implementation allows developing applications which can store their persistent predicates alternatively in files or databases with only a few simple changes to a declaration stating the location and modality used for persistent storage. The paper presents the model, the implementation approach in both the cases of using files and relational databases, a number of optimizations of the process (using information obtained from static global analysis and goal clustering), and performance results from an implementation of these ideas.

Resumen

Este trabajo describe un modelo de persistencia para lenguajes (C)LP y dos modos diferentes y útiles en la práctica de implementar este modelo en sistemas actuales. La idea fundamental es que la persistencia es una característica de ciertos predicados dinámicos que encapsulan estado. El efcto principal de la declaración de un predicado persistente es que los cambios dinámicos realizados se mantienen desde una ejecución a la siguiente. Tras proponer una sintaxis para la declaración de predicados persistentes, presentamos una implementación sencilla basada en ficheros, junto con algunos ejemplos. Vemos entonces que el concepto proporciona la manera más natural de realizar interfaces con bases de datos relacionales. Este interfaz se puede ver como una implementación alternativa a la de ficheros. La abstracción del concepto de persistencia permite el desarrollo de aplicaciones que pueden almacenar datos en ficheros o bases de datos con unos cambios mínimios a una declaración que establece la situación y modalidad del almacenamiento externo.

Contents

1 Introduction

State is traditionally implemented in Prolog and other (C)LP systems through the built-in ability to modify predicate definitions dynamically at runtime.[1] Generally, fact-only dynamic predicates are used to store information in a way that provides global visibility (within a module) and preserves information through backtracking. The logical view of internal rule base updates [1] confers a sensible semantics to the effect of internal state changes in a running Prolog program. This internal rule base, albeit a non-declarative component of Prolog, has many practical applications from the point of view of the needs of a programming language.[2]

However, Prolog internal rule base implementations associate the lifetime of the internal state with that of the process, i.e., they deal only with what happens when a given program is running and changes its private rule base. Indeed, the Prolog rule base lacks an important feature: data persistence. By data persistence we refer to rule base modifications surviving across program executions (and, as a later evolution, maybe being accessible to other programs –even concurrently). This feature, if needed, must be explicitly implemented by the programmer in traditional systems: it can be achieved for example by explicitly reading and writing the state of the program to an external device, be it a file with Prolog facts, or an external database. This approach offers a basic solution, but unless substantial effort is devoted to the task it will typically lack an updated reflection of the Prolog internal state in the external device, as synchronization points will be quite scattered in time. Of course Prolog rule base accesses can be replaced with sequences of goals which explicitly read / write terms to files or external databases, but this brings important changes in the program with respect to the case where no external storage is used. Also, substantial recoding is needed if the external storage medium is to be changed from a file-based one to a database-based one or the other way around. The same applies to programs using the internal Prolog rule base which have to be adapted to using an external database.

In this paper we present a conceptual model of persistence by proposing the concept of *persistent predicates*, and a number of implementations thereof. This concept implies associating an external storage medium to selected (dynamic) predicates, which are then termed *persistent*. A persistent predicate is thus a special kind of dynamic, data predicate that "resides" in some persistent medium (such as a set of files, a database, etc.) and which is typically external to the program using such predicates. The main effect is that any changes made to a persistent predicate from a program "survive" across executions, i.e., if the program is halted and restarted the predicate that the new process sees is in precisely the same state as it was when the old process was halted (provided no change was made in the meantime to the storage by other processes or the user). Notably, persistent predicates appear to a program as ordinary dynamic predicates: calls to these predicates can appear in clause bodies in the usual way without any need to wrap or mark them as "external" or "database" calls and updates to persistent predicates can be made using the standard `asserta/1`, `assertz/1`, `retract/1`, etc. predicates used for ordinary dynamic predicates (which are suitably modified). Updates to persistent predicates are guaranteed to be atomic and transactional, in the sense that if an update terminates, then the external storage has definitely been modified. This model provides a high degree of conceptual compatibility with previously existing programs which access only the local rule base, [3] while bringing at the same time several practical advantages:

- The state of dynamic predicates is, at all times, reflected in the state of the external storage device. If the program making the updates is halted just after one update and then restarted, then the updated state of the predicate will be seen. This provides security against possible data loss due to, for example, a system crash.

[1]These predicates sometimes have to be marked explicitly as *dynamic*.

[2]Examples of recent proposals to extend its applicability include using it to model reasoning in a changing world [2], and as the basis for communication of concurrent processes and objects [3, 4].

[3]Conceptual compatibility is not guaranteed in the case of relational database-based predicates which are being modified while queries to the same predicates are still active, as described in [1]. The current approach follows the "Immediate Update View" as shown in [1]. This is subject of ongoing work.

- Since accesses to persistent predicates are *viewed* as regular accesses to the Prolog rule base, analyzers (and related tools) for full Prolog can deal with them in the same way as with the standard dynamic predicates. Also, since the calls to persistent predicates are standard literals, traditional analysis tools can infer the types and modes of the arguments which, as we will see, can result in optimizations. Using explicit accesses to files or external databases through low-level library predicates would make this task much more difficult.

Finally, perhaps the most interesting advantage of the notion of persistent predicates is that it abstracts away how the predicate is actually stored. Thus, a program can use persistent predicates stored in files or in external relational databases interchangeably, and the type of storage used for a given predicate can be changed without having to modify the program except for replacing a single declaration in the whole program. It also minimizes impact on the host language, as the semantics of the access to the rule base is compatible with that of Prolog.

Our approach builds heavily on the well known and close relationship between (Constraint) Logic Programming and relational databases [5]: for example, operations in the relational algebra can be easily modeled using Horn clauses (plus negation for some operations), where database tables are seen as fact-only predicates, and every record is seen as a fact. On the other hand, the embedding into Prolog allows combining full Prolog code (beyond DATALOG) with the accesses to the persistent predicates.

A number of current Prolog systems offer external database interfaces, but often with ad-hoc access builtins. In those cases in which some kind of transparency is provided (e.g. Quintus *ProDBI*, SICStus and LPA *Prodata*, ECLiPSe), the system just allows to perform queries on tables as if they were Prolog predicates, but they do not allow updating tables using the same transparent approach. We argue that none of these cases achieve the same level of flexibility and seamless integration with Prolog achieved in our proposal.

In summary, the main technical contributions of this paper are as follows:

- We introduce the notion of persistent predicates. This notion implies that for such predicates the state of the Prolog rule base is reflected in an external storage facility, thus allowing (this part of) the program state to survive across executions. The syntax to access the database facts is the same as for other Prolog goals, and only minor modifications, orthogonal to the program code and to its internal logic, are needed in a given program to achieve persistence.

- We also present two different implementations of the above: one based on storing facts on files and another one based on storing them in an external SQL database transparently to the user.

- We propose two optimizations of the implementation, based on applying the results of program analysis to reduce run-time overhead and on clustering goals to reduce database accesses.

- We present a performance study, comparing the aforementioned implementations with the straightforward use of the internal Prolog rule base. In the particular case of using an SQL database as external storage, we also analyze how the overhead due to the compilation of SQL queries and their transmission can be improved with compile-time analysis information and literal clustering.

Implementations of this model have been used in real-world applications such as the Amos [6] tool, part of a large, ongoing international project aimed at facilitating the reuse of Open Source code through the use of a powerful, ontology-based search engine working on a large database of code information.

2 A Proposal for Persistent Predicates in Prolog

We will now define a syntax for the declaration of persistent predicates. We will also present briefly two different implementations of persistent predicates which differ on the storage medium (files of Prolog terms on one case, and an external relational database on the other). Both implementations aim at providing a semantics compatible with that of the Prolog internal rule base, but enhanced with persistence over program executions.

2.1 Declaring Persistent Predicates

The syntax that we propose for defining persistent predicates is based on the assertion language of Ciao Prolog [7], which allows expressing in a compact, uniform way, types, modes, and, in general, different (even arbitrary) properties of predicates.

In order to specify that a predicate is persistent we have to flag it as such, and also to define where the persistent data is to be stored. Thus, a minimum declaration is:

```
:- include(library(persdb)).

:- pred employee/3 + persistent(payroll).
:- pred category/2 + persistent(payroll).

:- persistent_db(payroll, file('/home/clip/accounting')).
```

The first declaration states that the persistent database library is to be used to process the source code file: the `include`d code loads the persdb library support predicate definitions, and defines the local operators and syntactic transformations that implement the persdb package. The second and third line state that predicates `employee/3` and `salary/2` are persistent and that they live in the storage medium to be referred to as `payroll`, while the fourth one defines which type of storage medium the `payroll` identifier refers to (the `persistent_db/2` information can also be included in the argument of `persistent`, but using `persistent_db/2` declarations allows factoring out information shared by several predicates). It is the code in the *persdb* package that processes the `persistent/1` and `persistent_db/2` declarations, and which provides the code to access the external storage and keeps the information necessary to deal with it. In this particular case, the storage medium is a disk file in the directory specified in the directive. The predicates in Figure 1 use these declarations to compute the salary of some employee, and to increment the number of days worked:

```
salary(Empl,Salary):-                  one_more_day(Empl):-
    employee(Empl,Categ,Days),             retract(employee(Empl,Categ,Days)),
    category(Categ,PerDay),                Days1 is Days + 1,
    Salary is Days * PerDay.               assert(employee(Empl,Categ,Days1)).
```

Figure 1: Accessing and updating a persistent predicate

In this simple case, no further information about the persistent predicates is needed. However, if the external storage is to be kept in an SQL database, argument type information is required in order to create the table (if the database is empty) and also to check that the calls are made with compatible types. It is also necessary to establish a mapping (views) between the predicate functor and arguments and table name and columns. Suitable declarations to store in an external database the information related to the employees and categories are:

```
:- include(library(persdb)).

:- pred employee/3 :: string * string * int +
        persistent(employee(ident, category, time), payroll).
:- pred category/2 :: string * int         +
        persistent(category(category, money), payroll).

:- persistent_db(payroll, db(paydb, admin, 'Pwd', 'db.comp.org')).
```

The `db/4` declaration indicates database name (`paydb`), database server (`db.comp.org`), database user (`admin`) and password (`Pwd`). This information is processed by the `persdb` package, and a number of additional formats can be used. For example, the port for the database server can be specified (as in `'db.comp.org':2020`), the precise database brand can be noted (as, for example `odbc/4` or `oracle/4` instead of the generic `db/4`), etc. This instructs the `persdb` package to use different connection types or to generate queries specialized for particular SQL dialects. In addition, values for the relevant fields can also be filled in at run time, which is useful for example to avoid storing sensitive information, such as password and user names, in program code. This can be done using hook facts or predicates, which can be included in the source code, or asserted by it, perhaps after consulting the user. These facts or predicates are then called when needed to provide values for the arguments whose value is not specified in the declaration. For example, a declaration such as:

```
:- persistent_db(payroll, db(paydb, _, _, 'db.comp.org')).
```

would call the hook predicates `db_user/2` and `db_passwd/2`, which are expected to be defined as

```
db_user(payroll, User):- ...
db_password(payroll, Password):- ...
```

Note also that, as mentioned before, the declarations corresponding to `employee/3` and `category/2` specify the name of the table in the database (which can be different from that of the predicate) and the name of each of its columns. It may also have a type signature. If a table is already created in the database, then this declaration of types is not strictly needed, since the system will retrieve the schema from the database. However, it may still be useful so that (compile-time or run-time) checking of calls to persistent predicates can be performed:

- The types are needed when tables are to be automatically created in the database since databases usually need the user to provide explicit types for every column. In some cases the type inferencing algorithms (see below) can infer the types from those of other predicates in the program.
- The types are also useful so that type errors (i.e., trying to send a record with a number in a place where a string was expected) can be caught and reacted to as soon as possible — maybe even at compile time or, if at run time, before the database implementation raises an error. Note that in a file-based implementation these type declarations are not needed, but they are still encouraged in order to ensure type-safety (and make it trivial to migrate the code to the database implementation).
- Lastly, types and modes can be read and inferred by a global analysis tool, such as, e.g., CiaoPP [8, 9], and used to optimize the generation of SQL expressions and to remove superfluous runtime checks at compile time (see Section 2.3.1).

It is interesting to point out that the predicate code in Figure 1 does not have to change to work with a new placement of the database: the way the database is accessed is taken care of by the library package, and the information about where the database lives is defined in the declarations. The program always contains standard internal database access and aggregation predicates, independently of whether the storage medium is the internal Prolog rule base, file-based, or SQL-database based.

A dynamic version exists of the `persistent` declaration, which allows defining new persistent predicates on the fly, under program control. Also, in order to provide greater flexibility, lower-level operations (of the kind available in traditional Prolog-SQL interfaces) are also available, which allow establishing database connections. These are the library operations the above examples are compiled into. Finally, a persistent predicate can also be made to correspond to a complex view of several database tables.

We will now give a longer code example: the complete source of a program implementing a simple, persistent queue, and which interacts with the user. The elements of the queue are kept as persistent data facts, and the program state can therefore be recovered after it has being stopped and resumed:

```
:- module(queue, [main/0]).
:- include(library(persdb)).

:- pred queue/1 + persistent(file('/tmp/queue')).

main:-
     write('Action ( in(Term). | out. | list. | halt. ): '),
     read(A),
     (  handle_action(A)
     -> main
     ;  write('Unknown command.'), nl, main ).

handle_action(end_of_file) :- halt.
handle_action(halt) :- halt.
handle_action(in(Term)) :- assertz(queue(Term)).
handle_action(out) :-
     (  retract(queue(Term))
     -> write('Out '), write(Term)
     ;  write('EMPTY!') ), nl.
handle_action(list) :-
     findall(T,queue(T),Contents),
     write('Contents: '), write(Contents), nl.
```

An example interaction is the following (note that the program is stopped and restarted):

```
$ ./queue
Action ( in(Term). | out. | list. | halt. ): in(first).
Action ( in(Term). | out. | list. | halt. ): in(second).
Action ( in(Term). | out. | list. | halt. ): list.
Contents: [first, second]
Action ( in(Term). | out. | list. | halt. ): halt.
$ ./queue
Action ( in(Term). | out. | list. | halt. ): out.
Out first
Action ( in(Term). | out. | list. | halt. ): list.
Contents: [second]
Action ( in(Term). | out. | list. | halt. ): out.
Out second
```

```
Action ( in(Term). | out. | list. | halt. ): out.
EMPTY!
Action ( in(Term). | out. | list. | halt. ): halt.
```

2.2 File-Based Implementation

The file-based implementation of persistent predicates provides a light-weight, simple, and at the same time quite powerful form of persistence. It has the advantage of being standalone in the sense that it does not require any external support other than the file management capabilities provided by the operating system. This is thanks to the fact that the persistent predicates are stored in one (or more) auxiliary files under the direct control of the persistent library. This implementation is especially useful when building small to medium-sized standalone (C)LP applications which require persistent storage and which may have to run in an environment where the existence of an external database manager is not ensured. Also, it is very useful even while developing applications which will connect to databases, because it allows working with persistent predicates maintained in files when developing or modifying the code and then switching to using the external database for testing or "production" by simply changing a declaration.

The storage medium where persistent predicates live in is associated to a directory, under which the file(s) which keep the state of the predicates is (are) placed. These files are stored in plain ASCII format and using Prolog syntax. One obvious advantage of this approach is that such files can, if needed, be edited by hand using any text editor, or even easily read and written by other applications not necessarily using this persistent model or even written in Prolog.

The implementation attempts to provide at the same time efficiency and security. Three files are used for each predicate: the *data file*, which stores a *base state* for the predicate; the *operations file*, which stores the differential between the base state and the predicate state in the program (i.e., operations pending to be integrated into the data file); and the *backup file*, which stores a security copy of the data file.

When no program is accessing the persistent predicate (because, e.g., no program updating that particular predicate is running), the data file reflects exactly the facts in the Prolog internal rule base. When any insertion or deletion is performed, the corresponding change is made in the Prolog internal rule base, and a record of the operation is *appended* to the operations file. In this moment the data file does not reflect the state of the internal Prolog rule base, but it can be reconstructed by applying the changes in the operations file to the state in the data file. This strategy incurs only in a relatively small, constant overhead per update operation (the alternative of keeping the data file always up to date would lead to an overhead linear in the number of records in it).

When a program using a file-based persistent predicate starts up, the data file is first copied to a backup file, and all the pending operations are performed on the data file by loading it into memory, re-executing the updates recorded in the operations file, and saving a new data file. The order in which the operations are performed and the concrete O.S. facilities (e.g., file locks) used ensure that even if the process aborts at any point in its execution, the data saved up to that point can be completely recovered upon a successful restart. The backup of the data file is used to prevent data loss if the system crashes during this operation. For more security, the data file can also be explicitly brought up to date on demand at any point in the execution of the program.

2.3 External Database Implementation

Another obvious alternative implementation is by using a relational database as the persistent storage medium. This is clearly useful, for example, when the data already resides in such a database (where it is perhaps also accessed by other applications) or the amount of data is very large. We present another implementation of persistent predicates which keeps the storage in a relational database. A more extensive description of this interface (including the design of an

ODBC mediator) can be found in [10, 11].

One of the most attractive features of our approach is that this view of external relations as just another storage medium for persistent predicates provides a very natural and transparent way to perform simple accesses to relational databases from (C)LP programs. As mentioned in Section 2.1, this implementation allows reflecting selected columns of a relational table as a persistent predicate. The implementation also provides facilities for reflecting complex views of the database relations as individual persistent predicates. Such views can be constructed as conjunctions, disjunctions or projections of database relations, and may include SQL-like aggregation operations (using where appropriate the corresponding Prolog primitives, such as `findall`, `bagof`, `setof`, etc.)

The architecture of the database interface (Figure 2), has been designed with two goals in mind: simplifying the communication between the Prolog side and the relational database server, and providing platform independence, allowing inter-operation when using different databases.

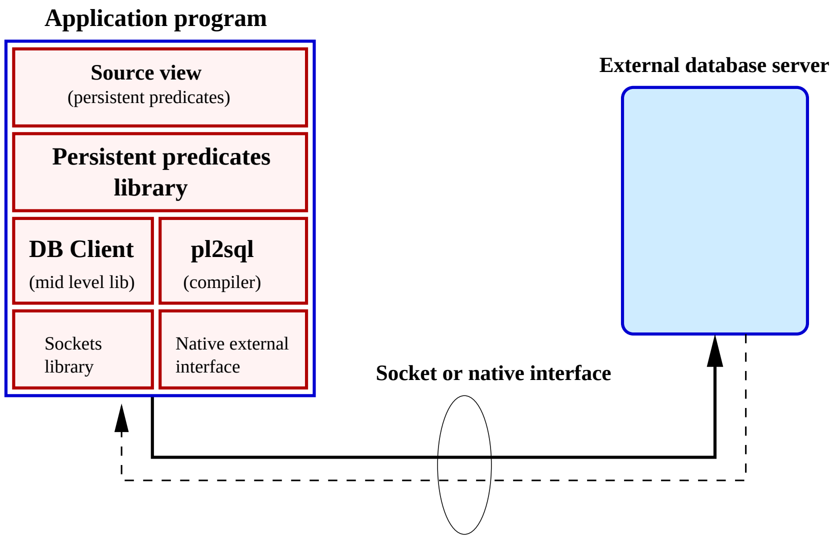

Figure 2: Architecture of the access to an external database

The interface is built on the Prolog side by stacking several abstraction levels over the socket and native code interfaces (Figure 2). Typically, database servers allow connections using TCP/IP sockets and a particular protocol. Alternatively, linking directly a shared object or a DLL may be needed. In other cases a special-purpose mediator which acts as a bridge between a socket and a native interface (e.g., certain versions of ODBC) has been developed [10, 11]. Thus, the low level layer is highly specific for each database implementation (e.g. MySQL, Postgres, ORACLE, etc.). The mid-level interface (which is similar in level of abstraction to that present in most current Prolog systems) abstracts away these details.

The higher-level layer implements the concept of persistent predicates so that calls and database updates to persistent predicates actually act upon relations stored in the database by means of automatically generated mid-level code. In the base implementation, at compile-time, a “stub” definition is included in the program containing one clause whose head has the same predicate name and arity as the persistent predicates and whose body contains the appropriate mid-level code, which basically implies activating a connection to the database (logging on) if the connection is not active, sending the appropriate SQL code, recovering the solutions (or the first solution and the DB handle for asking for more solutions), retrieving additional solutions on backtracking or eventually failing, and closing the connection (logging off the database), therefore freeing the programmer from having to pay attention to low-level details (but the mid-level interface is always available, of course).

The SQL code in particular is generated using a Prolog to SQL translator based on the excellent work of Draxler [12]. Modifications were made to the code of [12] so that the compiler can deal with the different idioms used by different databases, the different types supported, etc. as well as blending with the high-level way of declaring persistence, types, modes, etc. that we have proposed (and which is in line with the program assertions used throughout in the Ciao system). All conversions of data types are also automatically handled by the interface, using the type declarations provided by the user or inferred by the global analyzers.

In principle the SQL code corresponding to a given persistent predicate, literal, or group of literals needs to be generated dynamically at run-time for every call to a persistent predicate since the mode of use of the predicate affects the code to be generated and can change with each run-time call. Clearly, a number of optimizations are possible. In general, a way to improve performance is by reducing overhead in the run-time part of the Prolog interface by avoiding any task that can be accomplished at compile-time, or which can be done more efficiently by the SQL server itself. We study two different optimization techniques based on these ideas: the use of static analysis information to pre-compute the SQL expressions at compile time (which is related to adornment-based query optimization in deductive databases [13]), and the automatic generation of complex SQL queries based on Prolog query clustering.

2.3.1 Using static analysis information to pre-compute SQL expressions

As hinted at before, the computation of SQL queries can be certainly sped up by creating skeletons of SQL sentences at compile-time, and fully instantiating them at run-time. In order to create the corresponding SQL sentence for a given goal (or set of goals –see below) to a persistent predicate at compile-time, information regarding the instantiation status of the variables that appear in the goal is needed. This mode information can be provided by the user using the Ciao assertion language. More interestingly, this information can typically be obtained automatically by using program analysis. As mentioned before, in the Ciao system this is accomplished by **CiaoPP**, a powerful program development tool which includes a static analyzer, based on Abstract Interpretation [8, 9]. If the program is run through CiaoPP, and selecting the appropriate options, the output will contain, at every program point and in the form of `true` assertions, the abstract substitution resulting from the analysis using a given domain. Although the essential information here is argument groundness (we just need to know which database columns must appear in the `WHERE` part of the SQL expression), we use the sharing/freeness abstract domain because it provides more accurate groundness information than simple groundness analyses.

For example, assume that we have an SQL-based persistent predicate as in Section 1:

```
:- pred employee/3 :: string * string * int +
        persistent(employee(ident, category, time), payroll).
```

and consider also the program shown in the left side of Figure 1. The literal `employee/3` will be translated by the persistence library to a mid-level call which will at run-time call the `pl2sql` compiler to compute an SQL expression corresponding to `employee(Empl,Categ,Days)` based on the groundness state of `Empl`, `Categ` and `Days`. These expressions can be precomputed for a number of combinations of the groundness state of the arguments (with still some run-time overhead to select among these combinations, which unfortunately can obviously be many if the number of arguments is large). Furthermore, if the static analyzer can infer that `Empl` is ground when calling `employee(Empl,Categ,Days)`, we will be able to build at compile-time the SQL query for this goal as:

```
SELECT ident, category, time FROM employee WHERE ident = '$Empl$';
```

The only task that remains to be performed at run-time, before actually querying the database, is to replace `$Empl$` with the actual value that `Empl` is instantiated to and send the expression to the database server.

A side effect of (SQL-)persistent predicates is that they provide useful information which can improve the analysis results for the rest of the program: the assertion that declares a predicate (SQL-)persistent also implies that on success all the arguments will be ground. This additional groundness information can be propagated to the rest of the program. For instance, in the definition of `salary/2` in Figure 1 `category/2` happens to also be a persistent predicate living in an SQL database, and we will surely be provided with groundness information for `category/2` so that the corresponding SQL expression will be generated at compile-time as well.

2.3.2 Query clustering

The second possible optimization on database queries is query clustering. A simple implementation approach would deal separately with each literal calling a persistent predicate, generating an individual SQL query for every such literal. Under some circumstances, mainly in the presence of intensive backtracking, the flow of tuples through the database connection generated by the Prolog backtracking mechanism will produce limited performance.

In the case of complex goals formed by consecutive calls to persistent predicates (separated only by perhaps some tests or aggregation operations) it is possible to take advantage of the fact that database systems include a great number of well-developed techniques to improve the evaluation of complex SQL queries. The Prolog to SQL compiler is in fact able to translate such complex conjunctions of goals into efficient SQL code. The compile-time optimization that we propose requires identifying literals in clause bodies which call SQL-persistent predicates and are contiguous (or can be safely reordered to be contiguous) so that they can be clustered and, using also the mode information, the SQL expression corresponding to the entire complex goal compiled as a single unit. This is a very simple but powerful optimization, as will be shown later. Optimizations of this kind may use sophisticated goal reordering techniques, although the one presented here is just an example to highligh the speedup that may be obtained with these techniques.

For example, in `salary/2` of the the program in Figure 1, if `category/2` is defined as:

```
:- pred category/2 :: string * int +
        persistent(category(category, money), payroll).
```

and assuming that we have analysis information which ensures that `salary/2` is always called with a ground term in its first argument, then a single SQL query will be generated at compile-time for both persistent predicates, such as:

```
SELECT ident, category, time, rel2.money
FROM employee, category rel2
WHERE ident = '$Empl$' AND rel2.category = category;
```

2.4 Transactions and concurrency handling

There are two main issues to address in these implementations of persistence related to transactional processing and concurrency. The first one is *consistency*: when there are several processes changing the same persistent predicates concurrently, the final state must be consistent w.r.t. the changes made by every process. The other issue to be addressed in the implementations presented in previous sections is *visibility*: every process using a persistent predicate must be aware of the changes made by other processes which use that predicate. Another, related issue is what means exist in the source language to express that a certain persistent predicate could be accessed by several threads or processes, and how several accesses and modifications to a set of persistent predicates are grouped so that they are implemented as a single transaction.

Regarding the source language issues, the Ciao language already includes a way to mark dynamic data predicates as concurrent [3], i.e., of marking that such predicates could be modified by several threads or processes. Also, a means has been recently developed for marking that a group of accesses and modifications to a set of dynamic predicates constitute a single atomic transaction [14]. One of the methods proposed in [14] to this end is by declaring that a given Prolog predicate is *transactional*. The effect of this is that all the literals in the body of the predicate which refer to concurrent predicates (persistent or not) are executed atomically. Both of these source level annotations (marking predicates as concurrent and marking them as transactional) can be applied to persistent predicates and operations on them, respectively. In the following we outline how this affects the implementations described before for the file-based and database-based implementations of persistent predicates.

With respect to the the issue of consistency, for predicates marked as concurrent, the file-based implementation described provides a basic consistence mechanism for each individual update based on the operating system file locking facilities. In the case of using a database as storage, such consistency is provided automatically by the database itself. For the case in which several changes to a given set of persistent predicates are marked as transactional, and should thus be considered an atomic unit, when such a transactional predicate is executed, a transaction in the underlying storage medium is started. If the predicate fails, the transaction is rolled back, undoing all the changes made. If the transactional predicate succeeds, the transaction is committed.[4]

Visibility of changes to other processes is provided automatically when using an external database for storing persistent predicates, since most commonly used database systems are based on client-server architectures which already provide this kind of visibility. In the case of file-based persistent predicates, the implementation described needs to be modified. This involves checking if the files used to store a persistent, concurrent predicate have been changed after reading them, and starting a re-reading process only in this case.

3 Empirical results

We now study from a performance point of view the alternative implementations of persistence presented in previous sections. To this end, both implementations (file-based and SQL-based) of persistent predicates, as well as the compile-time optimizations previously described, have been integrated and tested in the Ciao Prolog development system [15].

In the following sections we compare the relative performance of the alternative implementations of persistence: in Section 3.1 we compare different implementations of persistent predicates with access to the Prolog internal rule base and with direct access to a MySQL database, and in Section 3.2 we review the improvements obtained in the SQL-based implementation when using compile-time optimizations with respect to the non-optimized implementation.

3.1 Performance without Compile-time Optimizations

The objective in this case is to check the relative performance of the various persistence mechanisms and contrast them with the internal Prolog rule base. The queries issued involve searching on the database (using both indexed and non-indexed queries) as well as updating it.

The results of a number of different tests using these benchmarks can be found in Tables 1 and 2. In Table 1, a four-column, 25,000 record database table is used to check the basic capabilities and to measure access speed. Each one of the four columns has a different measurement-related purpose: two of them check indexed accesses —using `int` and `string` basic data types—, and the other two check non-indexed accesses. The time taken by queries for the different combinations are given in the rows *non-indexed numeric query*, *non-indexed string query*, *indexed numeric query*,

[4]However, the implementation of transactional behaviour in Ciao is currently still experimental and the subject of separate work.

and *indexed string query* (time spent in 1,000 consecutive queries randomly selected). Row *assertz* gives the time for creating the 25,000 record table by adding the tuples one by one. Rows *non-indexed numeric retract*, *non-indexed string retract*, *indexed numeric retract*, and *indexed string retract* provide the timings for the deletion of 1,000 randomly selected records by deleting the tuples one by one.

Table 2 provides the results of the same tests on a 24-column table. This is used to measure the influence of the cost of data transfer in the different implementations when there is a much larger amount of data stored in the persistent predicates. The first four columns are identical to the ones in the four-column table before and are used for accessing the tuples, while the other 20 columns contain additional data which brings the individual record data size to approximately 2Kb.

The timings were taken on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 1Gb of RAM memory, running Red Hat Linux 8.0, and averaging several runs and eliminating the best and worst values. Ciao version 1.9.78 and MySQL version 3.23.54 were used.

The meaning of the columns is as follows:

prologdb (data) Is the time taken when accessing directly the internal (assert/retract) state of Prolog. Obviously, this rule base is not persistent, since it is memory-based and its contents will disappear when the Prolog process exits.

prologdb (concurrent) Is the same, but tables are marked as *concurrent.* This toggles a variant of the assert/retract database which allows concurrent access to the Prolog rule base. Atomicity in the updates is ensured and several threads can access concurrently the same table and synchronize through facts in the tables (see [3]). This measurement has been made in order to provide a fairer comparison with a database implementation, which has the added overhead of having to take into account concurrent searches/updates, user permissions, etc.[5]

persdb This is the implementation presented in Section 2.2, i.e., the file-based persistent version. The code is the same as above, but marking the predicates as persistent. Thus, in addition to keeping incore images of the rule base, changes are automatically flushed out to an external, file-based transaction record. This record provides persistence, but also introduces the additional cost of having to save updates. The implementation ensures atomicity and also basic transactional behavior.

persdb/sql This is the implementation presented in Section 2.3, i.e., where all the persistent predicates-related operations are made directly on an external SQL database. The code is the same as above, but marking the predicates as SQL-persistent. No information is kept incore, so that every database access imposes an overhead on the execution.[6]

sql This is, finally, a native implementation in SQL of the benchmark code, i.e., what a programmer would have written directly in SQL, with no host language overhead. To perform these tests the database client included in MySQL has been used. The SQL sentences have been obtained from the Ciao Prolog interface and executed using the MySQL client in batch mode.

Several conclusions can be drawn from Tables 1 and 2:

[5]Note, however, that this is still quite different from a database, apart, obviously, from the lack of persistence. On one hand databases typically do not support structured data, and it is not possible for threads to synchronize on access to the database, as is done with concurrent dynamic predicates. On the other hand, in concurrent dynamic predicates different processes cannot access the same data structures, which is possible in SQL databases. However, SQL databases usually use a server process to handle requests from several clients, and thus there are no low-level concurrent accesses to actual database files from different processes, but rather from several threads of a single server process.

[6]Clearly, it would be interesting to perform caching of read data, but note that this is not trivial since given that there can be concurrent updates to the database, an invalidation protocol must be implemented. This is left as future work.

	4 column tables				
	prologdb (data)	**prologdb** (concurrent)	**persdb**	**persdb/sql**	**sql**
assertz (25000 records)	590.5	605.5	5,326.4	16,718.3	3,935.0
non-indexed numeric query	7,807.6	13,584.8	7,883.5	17,721.0	17,832.5
non-indexed string query	8,045.5	12,613.3	9,457.9	24,188.0	23,052.5
indexed numeric query	1.1	3.0	1.1	1,082.4	181.3
indexed string query	1.1	3.0	1.5	1,107.9	198.8
non-indexed numeric retract	7,948.3	13,254.5	8,565.0	19,128.5	18,470.0
non-indexed string retract	7,648.0	13,097.6	11,265.0	24,764.5	23,808.8
indexed numeric retract	2.0	3.3	978.8	2,157.4	466.3
indexed string retract	2.0	3.1	1,738.1	2,191.9	472.5

Table 1: Speed in milliseconds of accessing and updating (4 column tables)

	24 column tables				
	prologdb (data)	**prologdb** (concurrent)	**persdb**	**persdb/sql**	**sql**
assertz (25000 records)	639.6	701.0	27,291.4	35,294.8	8,230.0
non-indexed numeric query	12,559.5	19,646.6	13,227.5	116,986.4	107,230.0
non-indexed string query	14,528.1	20,007.9	14,892.5	126,473.9,	117,261.3
indexed numeric query	2.4	3.8	2.0	2,441.4	498.8
indexed string query	2.4	4.0	2.0	2,240.6	502.5
non-indexed numeric retract	13,401.4	18,375.0	13,733.6	136,895.0	115,040.0
non-indexed string retract	13,727.0	18,954.9	17,267.5	133,811.5	119,628.8
indexed numeric retract	3.0	4.0	5,406.0	14,708.6	933.8
indexed string retract	3.0	4.3	10,838.0	15,168.1	1,056.3

Table 2: Speed in milliseconds of accessing and updating (24 column tables)

Sensitivity to the amount of data to be transferred The impact of this factor should appear mainly in the tests that need to send and receive data to/from an external (database) process (**persdb/sql** and **sql**). This communication is performed using a socket connection, which is of course sensitive to the size of the data in transit. In fact, these benchmarks always more than double the time elapsed when moving from using 4 column tables to using 24 column tables (in some cases the time is multiplied by 7).

The built-in mechanism of Prolog is less sensitive to the number of columns: asserting is affected very little by it. Queries show more performance variation, but this variation is smaller than in the case of involving socket communication.

Finally, the case of the file-based implementation lies in between the previous ones. All updates are reflected both in a transaction file and in the incore image of the database. Therefore queries show a similar behavior to the purely memory-based rule base operations of Prolog, while the slow-down of updates, due to the overhead of being recorded, is more similar (though smaller) than in the case of SQL-based queries.

Incidence of indexing The impact of indexing is readily noticeable in the tables, especially for the internal Prolog rule base but also for the file-based persistent database. The MySQL-based tests do present also an important speedup, but not as relevant as that in the Prolog-only tests. This behavior is probably caused by the overhead imposed by the SQL database requirements (communication with MySQL daemon, concurrency and transaction availability, much more complex index management, integrity constraint handling, etc). In addition to this, Prolog systems are usually highly optimized to take advantage of certain types of indexing, while database systems offer a wider class of indexing possibilities which might not be as efficient as possible in some determinate cases, due to their generality.

Impact of concurrency support Comparing the Prolog tests, it is worth noting that concurrent predicates bring in a non-insignificant load in rule base management (up to 50% slower than simple data predicates in some cases), in exchange for the locking and synchronization features they provide (as mentioned before, the concurrent Prolog internal rule base allows concurrent access from different threads of the same Prolog process and a sophisticated form of process synchronization). In fact, this slow-down makes the concurrent Prolog internal rule base show somewhat lower performance than using the file-based persistent database, which has its own file locking mechanism to provide inter-process concurrent accesses (but not from different threads of the same process: in that case both concurrency and persistence of predicates needs to be used).

Incidence of the Prolog interface in SQL characteristics Table 2 shows that direct SQL queries (i.e., typed directly at the database top-level interface) behave somewhat better than using the interface from Prolog, as is to be expected. However, there is a set of cases in which the difference is very significant (assert and indexed query and retract). This behavior can be explained considering that in the implementation of SQL-based persistence used for the tests the conservative approach was used of using a different connection to the database server for every query requested, and closing it when the query has finished (this is useful in practice in order to limit the number of open connections to the database, on which there is a limitation). We plan to perform additional tests turning on the more advanced setting in which the database connection is kept open.

3.2 Performance with Compile-time Optimizations

We have also implemented the two optimizations described in Section 2.3.1 (using static analysis information and query clustering) and measured the improvements brought about by these optimizations. The tests have been performed on two SQL-persistent predicates (`p/2` and `q/2`) with 1,000 facts each and indexed on the first column. There are no duplicate tuples nor duplicate values in any column (simply to avoid overloading due to unexpected backtracking). Both `p/2` and `q/2` contain exactly the same tuples.

Table 3 presents the time (in milliseconds) spent performing 1,000 repeated queries in a failure-driven loop. In order to get more stable measures average times were calculated for 10 consecutive tests, removing the highest and lowest values. The system used to run the tests was the same as in section 3.1.

The *single queries* part of the table corresponds to a simple call to `p(X,Z)`. The first row represents the time spent in recovering on backtracking all the 1,000 solutions to this goal. The second and third rows present the time taken when performing 1,000 queries to `p(X,Z)` (with no backtracking, i.e., taking only the first solution), with, respectively, the indexing and non-indexing argument being instantiated. The two columns correspond to the non-optimized case in which the translation to SQL is performed on the fly, and to the optimized case in which the SQL expressions are pre-computed at compile-time, using information from static analysis.

The '*complex queries*:`p(X,Z),q(Z,Y)`' part of the table corresponds to calling this conjunction with the rows having the same meaning as before. Information about variable groundness (on the first argument of the first predicate in the second row and on the second argument of the first predicate in the third row) obtained from global analysis is used in both of these rows. The two columns allow comparing the cases where the queries for `p(X,Z)` and `q(Z,Y)` are processed separately (and the join is performed in Prolog via backtracking) and the case where the compiler performs the clustering optimization and pre-compiles `p(X,Z),q(Z,Y)` into a single SQL query.

Finally, the '*complex queries*:`p(X,Z),r(Z,Y)`' part of the table illustrates the special case in which the second goal calls a predicate which only has a few tuples (but matching the variable bindings of the first goal). More concretely, `r/2` is a persistent predicate with 100 tuples (10% of the 1,000 tuples of `p/2`). All the tuples in `r/2` have in the first column a value which appears in the second column of `p/2`. Thus, in the non-optimized test, the Prolog execution mechanism will backtrack over the 90% of the solutions produced by `p/2` that will not succeed.

The results in Table 3 for single queries show that the improvement due to compile-time SQL

Single queries: `p(X,Y)`		
	on-the-fly SQL generation	pre-computed SQL expressions
Traverse solutions	36.6	28.5
Indexed ground query	1,010.0	834.9
Non-indexed ground query	2,376.1	2,118.1
Complex queries: `p(X,Z),q(Z,Y)`		
	non-clustered	clustered
Traverse solutions	1,039.6	51.6
Indexed ground query	2,111.4	885.8
Non-indexed ground query	3,550.1	2,273.8
Complex queries: `p(X,Z),r(Z,Y)`		
	non-clustered	clustered
Asymmetric query	1146.1	25.1

Table 3: Comparison of optimization techniques

Single queries: `p(X,Y)`		
	on-the-fly SQLn generation	pre-computed SQL expressions
Indexed ground query	197.5	27.6
Non-indexed ground query	195.4	27.3
Complex queries: `p(X,Z),q(Z,Y)`		
	non-clustered on-the-fly	pre-computed clustered queries
Indexed ground query	406.8	33.3
Non-indexed ground query	395.0	42.6

Table 4: Comparison of optimization techniques (*Prolog time only*)

expression generation is between 10 and 20 percent. These times include the complete process of a) translating (dynamically or statically) the literals into SQL and preparing the query (with our without optimizations), and b) sending the resulting SQL expression to the database and processing the query in the database. Since the optimization only affects the time involved in a), we measured also the effect of the optimizations when considering only a), i.e., only the time spent in Prolog. The results are shown in Table 4. In this case the run-time speed-up obtained when comparing dynamic generation of SQL at run time and static generation at compile time (i.e., being able to pre-compute the SQL expressions thanks to static analysis information) is quite significant. The difference is even greater if complex queries are clustered and translated as a single SQL expression: the time spent in generating the final SQL expression when clustering is pre-computed is only a bit greater than in the atomic goal case, while the non-clustered, on-the-fly SQL generation of two atomic goals needs twice the time of computing a single atomic goal. In summary, the optimization results in an important speedup on the Prolog side, but the overall weight of b) in the current implementation is more significant.

Returning to the results in Table 3, but looking now at the complex goals case, we observe that the speed-up obtained due to the clustering optimization is much more significant. Traversing solutions using non-optimized database queries has the drawback that the second goal is traversed twice for each solution of the first goal: first to provide a solution (as is explained above, `p/2` and `q/2` have exactly the same facts, and no failure happens in the second goal when the first goal provides a solution), and secondly to fail on backtracking. Both call and redo imply accessing the database. In contrast, if the clustering optimization is applied, this part of the job is performed inside the database, so there is only one database access for each solution (plus the last access when there are no more solutions). In the second and third rows, the combined effect of compile-time SQL expression generation and clustering optimization causes a speed-up of around 50% to 135%, depending on the cost of retrieving data from the database tables: as the cost of data retrieval increases (e.g., access based on a non-indexed column), the speed-up in grouping queries decreases.

Finally, the asymmetric complex query (in which the second goal succeeds for only a fraction of the solutions provided by the first goal) the elimination of useless backtracking yields the most important speed-up, as expected.

The authors would like to thank Ignacio Caballero, José Francisco Morales, Samir Genaim, and Cohavit Taboch for their collaboration with some implementation aspects and for feedback and discussions on the system.

References

1. Lindholm, T.G., O'Keefe, R.A.: Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In Lassez, J.L., ed.: Logic Programming: Proceedings of the Fourth International Conference and Symposium, The MIT Press (1987) 21–39
2. Kowalski, R.A.: Logic Programming with Integrity Constraints. In: Proceedings of JELIA. (1996) 301–302
3. Carro, M., Hermenegildo, M.: Concurrency in Prolog Using Threads and a Shared Database. In: 1999 International Conference on Logic Programming, MIT Press, Cambridge, MA, USA (1999) 320–334
4. Pineda, A., Bueno, F.: The O'Ciao Approach to Object Oriented Logic Programming. In: Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop), Copenhagen (2002)
5. Ullman, J.D.: Database and Knowledge-Base Systems, Vol. 1 and 2. Computer Science Press, Maryland (1990)
6. Carro, M.: The Amos project: An Approach to Reusing Open Source Code. In Carro, M., Vaucheret, C., Lau, K.K., eds.: Proceedings of the CBD 2002 / ITCLS 2002 CoLogNet Joint Workshop, School of Computer Science, Technical University of Madrid, Facultad de Informática (2002) 59–70
7. Puebla, G., Bueno, F., Hermenegildo, M.: An Assertion Language for Constraint Logic Programs. In Deransart, P., Hermenegildo, M., Maluszynski, J., eds.: Analysis and Visualization Tools for Constraint Programming. Number 1870 in LNCS. Springer-Verlag (2000) 23–61
8. Hermenegildo, M., Puebla, G., Bueno, F., López-García, P.: Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In: 10th International Static Analysis Symposium (SAS'03). Number 2694 in LNCS, Springer-Verlag (2003) 127–152
9. Hermenegildo, M., Bueno, F., Puebla, G., López-García, P.: Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In: 1999 International Conference on Logic Programming, Cambridge, MA, MIT Press (1999) 52–66
10. Caballero, I., Cabeza, D., Genaim, S., Gomez, J., Hermenegildo, M.: persdb˙sql: SQL Persistent Database Interface. Technical Report CLIP10/98.0 (1998)
11. Cabeza, D., Hermenegildo, M., Genaim, S., Taboch, C.: Design of a Generic, Homogeneous Interface to Relational Databases. Technical Report D3.1.M1-A1, CLIP7/98.0, RADIOWEB Project (1998)
12. Draxler, C.: Accessing Relational and Higher Databases through Database Set Predicates in Logic Programming Languages. PhD thesis, Zurich University, Department of Computer Science (1991)
13. Ramakrishnan, R., Ullman, J.D.: A survey of research on deductive database systems. Journal of Logic Programming **23** (1993) 125–149
14. Pattengale, N.D.: Transactional semantics. Technical Report CLIP3/04.0, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain (2004)
15. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., Puebla, G.: The Ciao Prolog System. Reference Manual (v1.8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM) (2002) System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.