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

J. Correas*, J. M. Gómez*, M. Carro*, D. Cabeza*, and M. Hermenegildo*,**

(*) School of Computer Science, Technical University of Madrid (UPM)
(**) Depts. of Comp. Science and El. and Comp. Eng., U. of New Mexico (UNM)

Abstract. This paper describes a model of persistence in (C)LP languages and two different ways to implement it in current systems. The fundamental idea is that persistence is a characteristic of certain dynamic predicates which encapsulate state. The main effect of declaring a predicate persistent is that the dynamic changes made to such predicates survive from one execution to the next one. We propose a syntax for declaring persistent predicates and then a simple, file-based implementation is presented. It is then argued that the concept developed provides the most natural way to interface with relational databases. Such an interface is then developed as simply one more implementation alternative to the simple, file-based approach. The abstraction of the concept of persistence from its implementation allows developing applications which can store data alternatively on files or databases with only a few simple changes to a declaration stating the location and modality used for persistent storage. Performance results comparing the different implementations, and also with and without information obtained from static global analysis, is presented.

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 (maybe within a module) and preserves information through backtracking. This facility is built on top of the so-called *internal database*. The logical view of internal database updates [LO87] confers a sensible semantics to the effect of database changes in a running Prolog program, and this internal database, 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 database implementations associate the lifetime of the information with that of the process, i.e., they deal only with what happens when a given program is running and changes its own database. Indeed, the Prolog database lacks an important feature present in all database systems: data persistence. Data persistence means that database modifications will survive

[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 [Kow96], and as the basis for communication of concurrent processes and objects [CH99,PB02].

across program executions, and maybe be accessible to other programs—even concurrently. This feature, if needed, must be explicitly taken care of by the programmer in traditional systems, for example by explicitly reading and writing the state of the database to an external device, be it a file with Prolog facts, or a true 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 database state in the external device, as synchronization points may be quite scattered in time. Prolog database accesses can of course 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 database, which have to be adapted to using an external database.

In this paper we present a conceptual model of persistence, *persistent predicates*, and a number of implementations thereof. This concept allows associating an external storage medium to selected (dynamic) predicates, which are then termed *persistent*. A persistent predicate is thus a special kind of dynamic, fact-only 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 "survive" across executions, i.e., if the program is halted and restarted, the new process sees persistent predicates which are in the same state as they were 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 database calls, and updates to persistent predicates can be made using the standard `asserta/1`, `assertz/1`, `retract/1`, etc. 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 compatibility with previously existing programs which access only the local database, 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.
- The external database can be used as a means to communicate and share data among programs, which can access it as if it were part of the local database of each program.
- Since database accesses are *viewed* as regular accesses to the Prolog database, 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.

A number of current Prolog systems have features which are related to the capabilities which our approach offers: Quintus Prolog offers *ProDBI* (also available for SICStus under the generic name *Prodata*), which allows queries (but not updates) on tables as if they were Prolog predicates; SICStus Prolog also has an interface to the Berkeley Database system [OBS99]; XSB, and SWI include *PrologSQL*, which can compile on demand a conjunction of literals to SQL using the Draxler compiler [Dra91], but which do not provide transparent persistence. However, we argue that none of these cases achieve the same level of flexibility and seamless integration with Prolog achieved in our proposal.

Implementations of this model have been used in several non-trivial tools such as, for example, the `WebDB` *deductive database engine* [GCH98], a generic database system with a highly customizable *html interface*. `WebDB` allows creating and maintaining Prolog-based databases as well as databases residing in conventional engines using any standard WWW browser. They have also been used in real-world applications such as the Amos [Car02] 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 one). Both implementations aim at providing a semantics compatible with that of the Prolog internal database, but enhanced with persistence over program executions.

2.1 Declaring Persistent Predicates

The syntax that we propose for defining persistent predicates is built upon the assertion language of Ciao Prolog [PBH00], 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

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

Fig. 1. Accessing and updating a persistent predicate

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 database to be referred to as `payroll`, while the fourth one defines which type of database the `payroll` identifier refers to. 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 database is kept on a disk file under the specified directory. The predicates in Figure 1 use these declarations to compute the salary of some employee and to increment the number of worked days.

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 name 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.mycomp.org')).
```

The `db/4` term in `persistent_db` declaration indicates database name (`paydb`), database server (`db.mycomp.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.mycomp.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(pay_db, db_user/2, db_passwd/2, 'db.mycomp.org')).
```

would call the 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), the name of each of its columns, and also the type signature. If a table is already created in the database, then this declaration of types is not strictly needed, since the system can retrieve the schema from the database. However, it is still useful so that (compile-time or run-time) checking of calls to persistent predicates can be performed:

- Types are needed when tables are to be automatically created since databases usually require types to be explicitly provided for every column. In some cases the type inferencing algorithms (see below) can deduce 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-safeness (and to 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 [HPBLG03,HBPLG99], and used to optimize the generation of SQL expressions and to remove superfluous runtime checks at compile time (see Section 2.3).

It is interesting to point out that the predicate code in Figure 1 does not have to be changed to work with a new placement of the database: database accessees are taken care of by the library package, and the information about where the database lives is defined in the declarations.

A dynamic version of the persistent declaration exists, 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.

2.2 File-Based Implementation

The file-based implementation of persistent predicates provides a lightweight, simple, and at the same time quite powerful form of database. 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 (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 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 database. When any insertion or deletion is performed, the corresponding change is made in the Prolog internal database, and a record of the operation is *appended* to the operations file. At this moment the data file does not reflect the state of the internal Prolog database, 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.

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 alternative implementation is by using a relational database as the 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 [CCG+98,CHGT98].

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.

The interface is built on the Prolog side by stacking several abstraction levels over the socket and native (C) interfaces. Typically, database servers allow connections using TCP/IP sockets and a particular protocol. Alternatively, linking

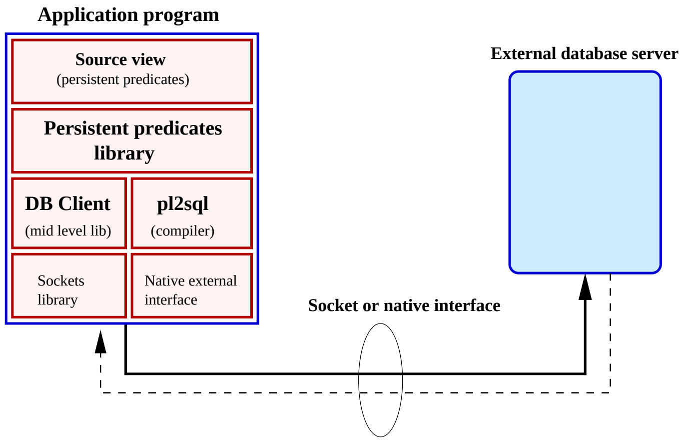

Fig. 2. Architecture of the access to an external database

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 [CCG+98,CHGT98]. 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 persistent predicates so that predicate calls and updates to them 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 to ask 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 these low-level details.

The SQL code is generated using a Prolog to SQL translator based on the excellent work of Draxler [Dra91]. Modifications were made to that code so that the compiler can deal with the different idioms, types, etc. used by different databases, as well to blend it with the high-level way of declaring persistence, types, modes, etc. that we have proposed. All conversions of data types are also automatically handled by the `persdb` interface.

In principle, the SQL code corresponding to a given persistent predicate, literal, or group of literals needs to be generated 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, performance can be improved by reducing run-time overhead on the Prolog side; in our case, 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 precompute the SQL expressions at compile time (which is related to adornment-based query optimization in deductive databases [RU93]), and the automatic generation of complex SQL queries based on clustering of Prolog queries. More optimizations are of course possible (and studied in the realm of relational and deductive databases [RU93]).

Using static analysis information to precompute SQL expressions Computation of SQL queries can be sped up by creating skeletons of SQL sentences at compile-time and instantiating them at runtime. In order to create the corresponding SQL sentence for a/some given goal(s), information regarding the instantiation state of the goal(s) variables is needed. This information can be provided with the Ciao assertion language. More interestingly, this information can typically be obtained automatically by using program analysis.

For example, assume that we have an SQL-based persistent predicate `employee/3` as in Section 2.1 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 call the `pl2sql` compiler at run-time to compute an SQL expression corresponding to `employee(Employee,Category,Days)` based on the groundness state of `Employee`, `Category` and `Days`. These expressions can be precomputed for different combinations of the groundness state of the arguments (with still some run-time overhead to select among these combinations, which unfortunately can obviously be large if the number of arguments is large). Furthermore, if the static analyzer can infer that only `Employee` is ground when calling `employee(Employee,Category,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 = '$Employee$';
```

The only task that remains to be performed at run-time, before actually querying the database, is to replace `$Employee$` with the actual value that `Employee` 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 as (SQL-)persistent also implies that all the arguments will be ground on success. This additional groundness information can then be propagated to the rest of the program. For instance, in the definition of `salary/2` in Figure 1, if `category/2` is also a persistent predicate stored in an SQL database, we will surely be provided with more groundness information.

Query clustering Another possible optimization on database queries is query clustering. A simple implementation approach would deal separately with each literal calling a persistent predicate, generating a separate 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 hinder performance.

Complex goals formed by consecutive calls to persistent predicates (separated only by perhaps some tests or aggregation operations) can be compiled to take advantage of the fact that database systems include well-developed techniques to improve the evaluation of complex SQL queries. The Prolog to SQL compiler is in fact able to translate 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, compiled into SQL as a single unit.

For example, in the predicate `salary/2` of the the program in Figure 1, and assuming that we have analysis information which ensures that `salary/2` is always called with a ground term in its first argument, a single SQL query will be generated at compile-time for both persistent predicates:

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

3 Empirical results

We will study now, 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 tested in the Ciao Prolog development system [BCC+02].

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 database. 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 Table 1. In these tests, a 25.000 record database table is used to check the basic capabilities and to measure access speed. For every type of database, Table 1 groups the results in different types of tests based on the two basic data types (`int` and `string`): the first row shows the time spent creating the table by adding the tuples one by one; the second group of rows tests databases access for performing different database operations based on indexed or non-indexed integer columns; finally, the last group of tests shows database access times for database operations based on indexed or non-indexed string columns. Times have been taken performing every database operation (except `assertz`) for 1.000 randomly selected database records. The Prolog code used in the benchmarks is the same for all the cases; only the storage place of the (persistent) predicates has been changed.

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 after eliminating the best and worst values. Ciao Prolog version 1.9#78 and MySQL version 3.23.54 were used. The meaning of the columns is as follows:

prologdb (data) This column reflects the time taken when accessing directly the internal (non-persistent, based on assert/retract) database of Prolog.

	prologdb (data)	**prologdb** (concurrent)	**persdb**	**persdb/sql**	**sql**
assertz (25000 records)	590.5	605.5	5326.4	16718.3	3935.0
numeric queries					
non-indexed numeric query	7807.6	13584.8	7883.5	17721.0	17832.5
indexed numeric query	1.1	3.0	1.1	1082.4	181.3
non-indexed numeric retract	7948.3	13254.5	8565.0	19128.5	18470.0
indexed numeric retract	2.0	3.3	978.8	2157.4	466.3
string queries					
non-indexed string query	8045.5	12613.3	9457.9	24188.0	23052.5
indexed string query	1.1	3.0	1.5	1107.9	198.8
non-indexed string retract	7648.0	13097.6	11265.0	24764.5	23808.8
indexed string retract	2.0	3.1	1738.1	2191.9	472.5

Table 1. Speed of accessing and updating

prologdb (concurrent) It is the same as the previous one, but dynamic predicates are marked as `concurrent`, which toggles a variant of the internal database which allows concurrent access to the Prolog database. Atomicity in the updates is ensured and several threads can access concurrently the same table and synchronize through facts in the tables [CH99]. 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.[3]

persdb This corresponds to the file-based implementation (Section 2.2) Thus, in addition to keeping incore images of the database, changes are automatically flushed out to an external, file-based transaction record, which also adds the derived from having to save updates. The implementation ensures atomicity and also basic transactional behavior.

persdb/sql This is the SQL-based implementation (Section 2.3) No information is kept incore, so every database access imposes an overhead on the execution with respect to the internal database case.[4]

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, using the database client included in MySQL. The SQL sentences have been obtained from the Ciao Prolog interface and then executed in batch mode.

Several conclusions can be drawn from Table 1:

[3] 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.

[4] 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.

Importance of indexing The impact of indexing is noticeable in the tables, especially for the internal Prolog database and 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, 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 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 database management (up to 50% slower than simple data predicates in some cases), in exchange for the locking and synchronization features they provide. In fact, this slow-down makes the concurrent Prolog internal database 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).

Overhead of the Prolog interface Table 1 shows that direct SQL queries (i.e., typed directly at the database top-level interface) behave somewhat better than those 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 a conservative approach which creates a different database connection for every query and later closes it was used in the implementation of SQL-based persistence used for the tests. This is useful in practice in order to limit the number of open connections to the database, which is limited.

Sensitivity to the amount of transferred data Some preliminary tests have been done about this issue. The benchmarks used a database table similar to the already describe, but with 2Kb of additional (and useless) data per record. As can expected, the SQL database benchmarks always more than double the time elapsed when switching from small records to large records. 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 socket communication. Finally, the case of the file-based implementation lies in between the previous ones.

3.2 Performance with Compile-time Optimizations

We have also implemented the two optimizations described in Section 2.3 (using static analysis information and query clustering) and measured the improvements brought about by them. The tests have been performed on two SQL-based persistent predicates (`p/2` and `q/2`) with 1.000 records each one and indexed on the first column, and one SQL-based persistent predicate (`r/2`) with 100 records and also indexed on the first column. There are no duplicate tuples nor duplicate values in any column, to avoid overloading due to unexpected backtracking.

Both `p/2` and `q/2` contain exactly the same tuples; `r/2` contains a randomly selected subset of `p/2`.

Table 2 presents the time (in milliseconds) spent for 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 first two columns correspond to non-optimized queries, while the third and fourth columns correspond to optimized ones. *Total* time refers to the time to complete the query; *Prolog* time is the Prolog preprocessing part of the query; *Diff* is the difference between them, and represent the communication time plus the database processing time proper. Prolog time does not appear when irrelevant: for example, when traversing solutions in the first row, which does not imply a significant use of the Prolog preprocessing part of the query).

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 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 and instantiating, respectively, the indexing and non-indexing argument. 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 precomputed at compile-time, using information from static analysis.

The '*complex queries*: `p(X,Z),q(Z,Y)`' section of the table corresponds to calling this conjunction (the rows have 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 to precompute an SQL query for the conjunction. Then we compare 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 cases where the compiler performs clustering optimization and `p(X,Z),q(Z,Y)` is executed as 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. All the arguments in the first position of `r/2` appear 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 2 for single queries show that the improvement due to compile-time SQL 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. The run-time speedup obtained when comparing dynamic and static generation of SQL is quite significant when Prolog time is taken into account.

Looking now at the case of complex goals, we observe that the speedup 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

Single queries: p(X,Y)						
	on-the-fly SQL generation			precomputed SQL expressions		
	Total	Prolog	Diff.	Total	Prolog	Diff.
Traverse solutions	36.6	–	–	28.5	–	–
Indexed ground query	1010.0	197.5	812.5	834.9	27.6	807.3
Non-indexed ground query	2376.1	195.4	2180.7	2118.1	27.3	2090.8

Complex queries: p(X,Z),q(Z,Y)						
	on-the-fly (non-clustered)			precomputed (clustered)		
	Total	Prolog	Diff.	Total	Prolog	Diff.
Traverse solutions	1039.6	–	–	51.6	–	–
Indexed ground query	2111.4	406.8	1704.6	885.8	33.3	852.5
Non-indexed ground query	3550.1	395.0	3155.1	2273.8	42.6	2231.2

Complex queries: p(X,Z),r(Z,Y)		
	on-the-fly (non-clustered)	precomputed (clustered)
	Total	Total
Asymmetric query	1146.1	25.1

Table 2. Comparison of optimization techniques

(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 second 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 speedup 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 speedup in grouping queries decreases. Anyway, the speedup of the Prolog preprocessing part in all these tests is especially relevant.

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 speedup, as expected.

4 Acknowledgments

This work has been supported in part by the European Union IST program under contract IST-2001-34717, Amos, and by MCYT project TIC 2002-0055, CUBICO. The authors would like to thank I. Caballero, José F. Morales, S. Genaim, and C. Taboch for their collaboration with some implementation aspects and for feedback and discussions on the system.

References

[BCC+02] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. 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), May

2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[Car02] M. Carro. The Amos project: An Approach to Reusing Open Source Code. In M. Carro, C. Vaucheret, and K.-K. Lau, editors, *Proceedings of the CBD 2002 / ITCLS 2002 CoLogNet Joint Workshop*, pages 59–70, School of Computer Science, Technical University of Madrid, September 2002. Facultad de Informática.

[CCG+98] I. Caballero, D. Cabeza, S. Genaim, J.M. Gomez, and M. Hermenegildo. persdb˙sql: SQL Persistent Database Interface. Technical Report CLIP10/98.0, December 1998.

[CH99] M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.

[CHGT98] D. Cabeza, M. Hermenegildo, S. Genaim, and C. Taboch. Design of a Generic, Homogeneous Interface to Relational Databases. Technical Report D3.1.M1-A1, CLIP7/98.0, RADIOWEB Project, September 1998.

[Dra91] Christoph Draxler. *Accessing Relational and Higher Databases through Database Set Predicates in Logic Programming Languages*. PhD thesis, Zurich University, Department of Computer Science, 1991.

[GCH98] J.M. Gomez, D. Cabeza, and M. Hermenegildo. WebDB: A Database WWW Interface. Technical Report CLIP11/98.0, December 1998.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[HPBLG03] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[Kow96] R. A. Kowalski. Logic Programming with Integrity Constraints. In *Proceedings of JELIA*, pages 301–302, 1996.

[LO87] Timothy G. Lindholm and Richard A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference and Symposium*, pages 21–39. The MIT Press, 1987.

[OBS99] Michael Olson, Keith Bostic, , and Margo Seltzer. Berkeley DB. In *1999 Summer Usenix Technical Conference*, June 1999.

[PB02] A. Pineda and F. Bueno. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[PBH00] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[RU93] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.