The Contact-Center Business Analyzer: a case for Persistent Contextual Logic Programming

Claudio Fernandes[1], Nuno Lopes*[2], Manuel Monteiro[3], and Salvador Abreu[1]

[1] Universidade de Évora and CENTRIA, Portugal
{cff,spa}@di.uevora.pt
[2] Digital Enterprise Research Institute, National University of Ireland, Galway
nuno.lopes@deri.org
[3] xseed, Lda., Portugal
manuel.monteiro@xseed.pt

Abstract. This article presents CC/BA, an integrated performance analyzer for contact centers, which has been designed and implemented using Persistent Contextual Logic Programming methods and tools. We describe the system's architecture, place it in perspective of the existing technology and argue that it provides interesting possibilities in an application area in which timely and accurate performance analysis is critical.

1 Introduction

The problem: The majority of the current contact center solutions provide extensive reporting, and some of them already provide even more sophisticated business intelligence solutions, allowing their users to analyze thoroughly the operational aspects of the contact center activity [6]. Performance indicators [5] normally made available are those related with how efficient the automatic call distributors (ACDs) are in distributing the calls (waiting times, abandoned calls, etc.) and how efficient the agents are in handling them (e.g. handled calls, call duration).

As there is normally little or even no integration with the surrounding business systems and applications, and there is no integration with the business data related with the various interactions processed by the contact center, these solutions can not correlate the operational data with the business data in order to provide more business oriented key performance indicators, like costs, profits and related ratios (operation margins, segment or customer value, ...).

* This author has been funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

Our purpose in developing CC-BA, the Business Analyzer tool, is to overcome the identified lack of systems integration and to allow to model and integrate business data with the existing operational data. Systems that might need to be integrated are workforce management tools, CRM and ERP tools.

Contexts: The idea of Contextual Logic Programming (CxLP) was introduced in the late 1980s by Monteiro and Porto [8] in the ALPES II project, and is related to similar efforts such as Miller's λProlog module system, described in [7].

The purpose of Contextual Logic Programming (CxLP) was initially to deal with Prolog's traditionally flat predicate namespace, a feature which seriously hindered the language's usability in larger scale projects. The impact of these extensions has mostly failed to make it back into the mainstream language, as the most widely distributed implementations only provide a simple, SICStus-like module mechanism, if any.

A more recent proposal [2] rehabilitates the ideas of Contextual Logic Programming by viewing contexts not only as shorthands for a modular theory but also as the means of providing dynamic attributes which affect that theory: we are referring to unit arguments, as described in Abreu and Diaz's work. It is particularly relevant for our purposes to stress the *context-as-an-implicit-computation* aspect of CxLP, which views a context as a first-class Prolog entity – a term, which behaves similarly to objects in OOP languages.

Persistence: Having persistence in a Logic Programming language is a required feature if one is to use it to construct actual information systems; this could conceivably be provided by Prolog's internal database but is quite adequately accounted for by software designed to handle large quantities of factual information efficiently, as is the case in relational database management systems. The semantic proximity between relational database query languages and logic programming have made the former privileged candidates to provide Prolog with persistence, and this has long been recognized.

ISCO [1] is a proposal for Prolog persistence which includes support for multiple heterogeneous databases and which access to technology beyond relational databases, such as LDAP directory services or DNS. ISCO has been successfully used in a variety of real-world situations, ranging from the development of a university information system to text retrieval or business intelligence analysis tools.

ISCO's approach for interfacing to DBMSs involves providing Prolog declarations for the database relations, which are equivalent to defin-

ing a corresponding predicate, which is then used as if it were originally defined as a set of Prolog facts. While this approach is convenient, its main weakness resides in its present inability to relate distinct database goals, effectively performing joins at the Prolog level. While this may be perceived as a performance-impairing feature, in practice it is not the show-stopper it would seem to be because the instantiations made by the early database goals turn out as restrictions on subsequent goals, thereby avoiding the filter-over-cartesian-product syndrome.

Contexts and persistence: Considering that it is useful to retain the regular Prolog notation for persistent relations which is an ISCO characteristic, we would like to explore the ways in which contexts can be taken advantage of, when layered on top of the persistence mechanisms provided by ISCO. In particular we shall be interested in the aspects of common database operations which would benefit from the increase in expressiveness that results from combining Prolog's declarativeness and the program-structuring mechanisms of Contextual Logic Programming.

We shall illustrate the usefulness of this approach to Contextual Logic Programming by providing examples taken from a large scale application, written in GNU Prolog/CX, our implementation of a Contextual Constraint Logic Programming language. More synthetic situations are presented where the constructs associated with a renewed specification of Contextual Logic Programming are brought forward to solve a variety of programming problems, namely those which address compatibility issues with other Prolog program-structuring approches, such as existing module systems. We also claim that the proposed language and implementation mechanisms can form the basis of a reasonably efficient development and production system.

The remainder of this article is structured as follows: section 2 introduces ISCO and Contextual Logic Programming as application development tools, proposing a unified language featuring both persistence and contexts. Section 3 addresses design issues for a performance analysis tool and introduces CC-BA. The impact of some design issues and environmental requirements is further explored in section 4, where some experimental results are presented. Finally, section 5 draws some conclusions and points at interesting directions for future work.

2 Persistent Contextual Logic Programming

Logic Programming and in particular the Plug language has long been recognized as a powerful tool for declaratively expressing both data and

processes, mostly as a result of two characteristics: the logic variable with unification and nondeterminism as a result of the built-in bactracking search mechanism.

While the Prolog is a consensual and relatively simple language, it is lacking in a few aspects relevant to larger-scale application development; some of these include:

- The lack of sophistication of its program structuring mechanisms: the ISO modules [4] proposal is representative of what has been done to take Prolog beyond the flat predicate space. The standard adds directives for managing sets of clauses and predicates, which become designated as *modules*. There are several areas where modules complicate matters w.r.t. regular Prolog: an example is the metacall issue, in which predicate arguments are subsequently treated as goals inside a module.
- The handling of persistent evolving data: one of Prolog's most emblematic characteristics is the way in which programs could easily be manipulated in runtime, by use of the `assert` and `retract` built-ins. These can be used to alter the program database, by adding or removing clauses to existing predicates. There are several issues with these, namely how do these interact with goals which have an active choice-point on a predicate being modified.
  One interesting feature provided by traditional Prolog implementations is the possibility of asserting non-factual clauses, which effectively endows the language with a very powerful self-modification mechanism. Nevertheless, the most common uses for clause-level predicate modification primitives deal with factual information only, this is true both for predicates which represent state information (which are typically updated in a way which preserves the number of clauses) and for database predicates, which will typically have a growing number of clauses.

Both of these aspects are addressed by the ISCO programming system, when deployed over GNU Prolog/CX, which compiles Prolog to native code. This implementation platform is discussed in [3] and briefly summarized in the rest of the present section.

2.1 Modularity with GNU Prolog/CX

Contextual Logic Programming was initially introduced with the intention of addressing the modularity issue. At present and in the GNU Prolog/CX implementation, CxLP provides an object-oriented framework for

the development of Logic programs; this is achieved through the integration of a passive, stateful part (the context) with a dynamic computation (a Prolog goal). The context is made up of instances of *units* which are similar to objects with state in other languages.

The current implementation has support for dynamic loading and unloading of units, which are implemented as OS-level shared libraries, or DLLs.

2.2 Persistence and large Database Predicates

Relational Database Management Systems provide efficient implementations for data storage, tailored to certain ways of accessing the information. The access is done via the SQL query language and the efficiency comes partly as a consequence of the availability of certain optimization features, such as multiple indexes on database relations. Prolog systems have long incorporated similar optimizations which map onto abstract machine-level features, such as clause indexing. Most prolog systems implement forms of indexing which are less general than the possible counterparts in an RDBMS.

Maintaining multiple indices – i.e. indexing predicates based on the analysis of anything other than the first argument – is intrinsically expensive and implies a significant runtime overhead to decide which index to use for a given call. As a consequence, very few Prolog systems incorporate any form of multi-argument indexing, usually requiring special annotations to indicate the fact.

Moreover, the combination of indices with dynamic predicates requires a dynamic rebuild of those indices, in particular one which does not survive the process in which it occurred. This means that, even if a Prolog database was appropriate for representing a given fact base, its use incurs the overhead of rebuilding the index each time:

- the predicate is updated (via `assert` or `retract`),
- the Prolog program is started

3 Application Design

The Contact Center Business Analyser is a web based application to analyse and control the performances of the numerous operations realized through managing a call center. Relying in data fetched from the company clients, it builds detailed financial scopes to be used by different management positions at the center.

A call center is a company department which purpose is to materialize personal communication between a company commercial campaign and their clients. The talking, realized by phone calls, is the job of the various agents working at the center, although mail and faxes are used sometimes.

The CC-BA software presents different levels of usage, divided in two groups: one for the administrator and the other for the end-users of the application. As expected, running a contact center requires many "eyes" over the operations, each pair monitoring different subjects. CC-BA maps that concept by providing several kinds of analysis, each one oriented for a particular profile. Each profile can then obtain different views of the data, grouped as several *KPIs - key performance indicators*.

Every analysis has a time window. This time period can take two forms: an interval period between two days, weeks or months, and a fixed hour schedule, also included in a time period.

3.1 Architecture

The CC-BA structure is similar to the traditional three layer web based applications, but with some slight differences. As shown in figure 1 (see page 7), we have a local database, an ISCO + GNU Prolog/CX Logical layer, and finally a small PHP layer responsible for the Prolog and Browser communication. When comparing to the traditional three layer approach, the ISCO + GNU Prolog/CX layer implements both the Logic and Presentation Layers. The Data layer is provided by a PostgreSQL database, which is populated in the ETL [9] process.

The data used by the application must be fetched from the database of the call center. However, only a small fraction of that data is relevant, and before it is loaded some transformations are needed to be done over it. This process must be done periodically and is called ETL (**E**xtraction, **T**ransformation and **L**oading).

The internal database is compact compared with the ones where the data originated from, however it is a very important component of the application and it must be extremely well tuned to perform the best it can over the queries. The tables are created through the definition of ISCO classes. Those classes, when compiled, will generate Prolog predicates that access the data as if they were regular clauses in a Logic Programming environment.

The core of the application are the available outputs formed by groups of *key performance indicators*. With the goal of achieving a modular system, the implementation of those outputs were designed in a way that adding new outputs formed by any aggregation of KPIs is a linear task.

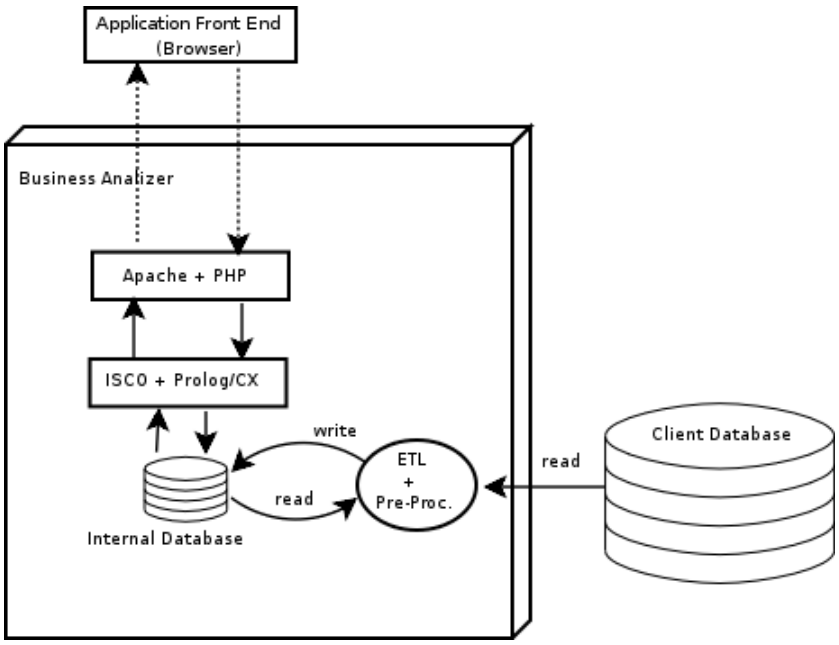

Fig. 1. CC-BA Architecture

Each *KPI* and output are coded in specific and different contextual Prolog units, such that to create a new output one only has to aggregate the *KPIs* desired into the new output unit. Any needed input parameters are pushed in through the GNU Prolog/CX context mechanism.

3.2 Example

Figure 2 (on page 8) depicts the unit that implements one use-case: the *Database Information* output. Just to name a few, we have the *Database Length* that is just the length of the contact list of a campaign, the "Successful Contacts", "Reschedules" and " Unsuccessful Contacts" that are based on the "Contact Numbers", switching the input parameter as it needed (lines 13, 16 and 19). The context mechanism of GNU Prolog/CX can also be used to specify unit parameters. In the example, the unit `param/3` is used to specify the arguments passed by, accordingly to what comes instantiated. In this case, a Client, a Campaign or an Agent.

Of all the parameters passed by context, there is one which requires special attention: the dates. Every analysis falls in a window period, and therefore all queries have time boundaries. Since the system had strict

```
:- unit(trat_bd).

output(Output):-
    var('campaign', CP),
    campaign@(code=CP, list_size=Total),
    format(d, Total):format(Total_DB),

    param(_,CP,_).proc_regs:value(Regs),
    format(d, Regs):format(ProcRegs),

    rate(Regs, Total):value(ProcRate),

    param(_,CP,_).contact_numbers([successful]):value(Succ),
    format(d, Succ):format(SuccCont),

    param(_,CP,_).contact_numbers([reschedule]):value(Resc),
    format(d, Resc).format(RescCont),

    param(_,CP,_).contact_numbers([unsuccessful]):value(Unsucc),
    format(d, Unsucc):format(UnsuccCont),

    Use is Succ + Unsucc,
    format(d, Use):format(UseContact),

    rate(Use, Total):value(PenetRate),
    rate(Succ, Use):value(SuccRate),
```

Fig. 2. Unit "trat_bd"

performance requirements, we had to explore alternative representations for some information. In this case, every date is mapped to an integer value, over which it is possible to set finite-domain constraints in GNU Prolog/CX.

3.3 ETL

The ETL (Extraction, Transform and Load) process consists of loading the database with the information provided about the Contact Center operations. This is an "offline" process, i.e. it can be performed at any time, without human intervention.

The extracted data is to be provided to the system in a predefined format, which can be outputed by the operational system of the Contact Center.

In the transformation process, the extracted data is converted into data that can be directly loaded into CC-BA. This process starts by checking the date of the last valid ETL and proceeds to loading the interactions performed on a later date. This will include:

- loading new campaigns and agents
- delete older data from the tables "interaction" and "agentWork"
- manage the agents and campaign states

The existence of new campaigns or agents impose the input of data from the system administrator, such as the income of the agent and the costs of each interaction of a campaign. This will leave those agents and campaigns in an unprocessed state until such information can be entered, once the required information is defined the calculations can be performed. The ETL process can also recognize new interactions in an inactive campaign or agent and automatically active the campaign or agent.

4 Performance Evaluation

One of the most sensitive facts about the CCBA is its efficiency. It was necessary to guarantee that all the available analysis were computed in a reasonable time. An initially identified problem involved some of the analysis having a large search space. Several relations in the data model will contain, at some point and for the time intervals being considered, tens of millions of records. This fact makes it necessary to ensure that all queries are made in a quick and efficient way.

Since the results of the raw data model were not satisfactory, we had to find other approaches to improve performance. Also, due to some technological restrictions (described later) it was necessary to reduce the number of joins in the low-level (SQL) queries. We opted for the use of data accumulation together with some pre-processing; the following sections provide more detail on this.

4.1 Data Accumulation

With the exception of one output, all queries have a minimal granularity of one day. This observation allows us to make daily accumulated values of the records in the database, thereby obtaining very significant performance gains w.r.t. the raw data. To achieve this, the following relations were created, along with the system data model:

`agentWork_daily`**:** This relation consists of the daily accumulates of a agent's work, i.e. the number of minutes an agent works per day and per campaign as well as the cost associated with that work.

`interaction_daily`**:** Here we collect the daily cumulates of all the information pertaining to the interactions made by the agents, grouped by the outcome of the interaction, the number of interactions of that type, the total duration of the calls and the total invoicing value.

`interaction_daily_results`**:** This relation is similar to the previous one but, here, we do not include the agent information. This allows for a smaller search space to be used in queries that do not need agent information, for example searching by campaigns or clients.

4.2 Data Pre-processing

Data pre-processing is the action of inserting the correct data in the accumulator tables. This action needs to be done in two separate situations:

- After the ETL phase (see section 3.3);
- After the manager changes the values of a given campaign, or new entities are configured.

As a consequence of the changes made by the manager to the invoice model of a campaign or to the costs of the agents, the accumulator tables have to be updated. Since all changes affect all the data in the database (not only after the change is made), all data in the accumulator tables has to be invalidated and recomputed. This process is transparent to the user.

4.3 Computations using Pre-Processed Data

We present a few representative computations and discuss how they are implemented using cumulative data tables.

Agent, Campaign and Client Costs. Using the pre-processed data, the cost of an agent, campaign or client over a time period can be computed resorting to the `agentWork_daily` table, where the total value of the cost is the sum of each day within the time period.

Total Work Time by Agent. The amount of time an Agent worked over a specified time period can be computed using the pre-processed information in the `agentWork_daily` table.

Contact duration. Using the daily pre-processed information `interaction_daily`, computing the average duration of the contacts of a certain type, performed by an agent, becomes more efficient than the previous implementation. Should it be necessary to request the contact duration for a certain campaign, this information is available in the table `interaction_daily_results`.

Invoice of an agent or campaign. Using the table `interaction_daily`, the calculation of the invoicing total for an agent within a time period can be done with the sum of each day in the period, taking the value stored in the `invoice` field. In the same way, the invoice for a campaign can be calculated using the table `interaction_daily_results`.

Total talk-time. The total talk-time (in minutes) for an agent or campaign, within a time period, can be computed using the tables `interaction_daily` and `interaction_daily_result`.

Answered contacts and total number of contacts. The number of daily answered contacts in a campaign can be calculated more efficiently using the table `interaction_daily_results`, which is also used, together with `interactions_daily`, to obtain the total number of contacts performed by an agent.

4.4 Views

With the correct use of indexes in the database fields and the cumulative tables in place, almost all queries were completed in a satisfactory time, given the desired operating conditions.[1] Nevertheless, some situations remained where the performance goals were not being achieved, namely in computing the number of contacts made by an agent and the total costs of the contact center.

In these situations, we chose to implement an SQL view for each one which is then mapped to an ISCO class. This allows the fields of the query to represent the desired result, while avoiding most further processing of the tuple data at the ISCO/application level. Each view is a *filter* for a larger table (possibly performing some computations), so

[1] In the range of 1-3 million events to treat, over a period of one month. The computational resources are very limited, our goal being that a simple workstation-style PC sould be enough to support CC-BA.

that the result is computed by the DBMS instead of retrieving all the records and performing the filter in the application.

To achieve this goal, the view is designed to be similar to the table it applies to, replacing all non relevant fields by similarly named fields with a fixed value of the appropriate range and type (for instance the minimum value in the records). All the fields needed to perform the query are left *unbound*. An additional field with the result of the computation is introduced, in general this is an aggregator such as `sum` or `count`.

The definition of these views in ISCO is made in the same manner as for a regular database table. It extends the table it applies to by adding the computed field. The ISCO Prolog access predicates are limited to the "lookup" ones, i.e. a class which maps to a view is considered `static`.

The following are examples of situations which we dealt with by defining views at the database level:

Number of contacts performed by an agent in a hour: This query cannot be computed by using the accumulator tables since its granularity is smaller than one day (the granularity of the *daily* accumulator tables.) As it was not feasible to use the simple table `interaction` to perform this query, a view was created to perform the computation at the DBMS level.

Processed Records: The total number of processed records of a campaign in a given time interval consists, in SQL terms, in a `count(distinct *)` from the contacts present in the `interaction` table. However, we know that `interaction` is the largest table present in the database with well above one million records per month of activity for a medium sized contact center, and computing this query cannot be directly done over it and remain computationally reasonable.

The introduction of a view decreased the execution time required for answering the query: in our first approach, this query was one of the slowest in the entire application, due to the amount of data that needed to be processed by ISCO. The use of the defined view, which left all the computations in the SQL side improved this situation dramatically.

The SQL code used of this view is shown in figure 3 and the definition of the view in ISCO is presented in figure 4. The ISCO side consists of extending the already defined class "`interaction`" with a new subclass which adds a "regs" field that is calculated in the view SQL code. Note that the SQL view must reproduce the fields which exist in the parent ISCO class.

```
CREATE VIEW "interaction_proc_regs" as
        select
                0 as oid,
                campaign,
                min(agent) as agent,
                min(contact) as contact,
                datetime,
                min(duration) as duration,
                min(result) as result,
                count(distinct contact) as regs
        from
                interaction
        group by campaign, datetime;
```

Fig. 3. Definition of "interaction_proc_regs" view

```
class interaction_proc_regs: interaction.
        regs:   int.
```

Fig. 4. ISCO definition of "interaction_proc_regs" view

When processing the records for 1.000.000 interactions, the times presented are for the first approach (all the calculations done in Prolog) are approximately 10 times higher than when using the defined view and cumlative data.

5 Conclusions and Directions for Future Work

Achieving the kind of data integration which CC-BA provides, coupled with the optimization capabilities to process these potentially very large amounts of data, results in great benefits for the call center management, meaning that the business oriented key performance indicators can be made available promptly, allowing the different manager roles to fine tune the systems to better reach their business goals, not only the operational ones.

One aspect of ISCO that turned out very useful is the ability to work with multiple indexes for database-stored data. For the quantities of data involved this turned out to be a critical aspect, as the same (factual) predicates are being accessed with different instantiation patterns.

We found out the hard way that the ETL process still requires a lot of manual effort and tuning. It would be worthwile to investigate the automation of the process and the design of the internal database, paying particular attention to the unification of several heterogeneous data sources, to the selection of the hierarchical cumulative sums, as well as to the generation of the internal database schema. Some of these aspects are already being worked on, in particular the design of declarative tools to achieve a measure of automation of the ETL process is the object of ongoing work.

On a finishing note, it is worth mentioning that our implementation of CC-BA is able to adequately function inside a virtual machine running Debian Linux with very low resource requirements: a simple Pentium-M based laptop was enough to account for tracking a medium-sized contact center.

References

1. Salvador Abreu. Isco: A practical language for heterogeneous information system construction. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. Prolog Association of Japan.
2. Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.
3. Salvador Abreu and Vitor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Osamu Takata, Masanobu Umeda, Isao Nagasawa, Naoyuki Tamura, Armin Wolf, and Gunnar Schrader, editors, *Declarative Programming for Knowledge Management, 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005, Fukuoka, Japan, October 22-24, 2005. Revised Selected Papers.*, volume 4369 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2006.
4. ISO/IEC JTC1/SC22/WG17. Information technology – Programming languages – Prolog – Part 2: Modules. Technical Report DIS 13211, ISO, 2000.
5. Ger Koole. Performance analysis and optimization in customer contact centers. In *QEST*, pages 2–5. IEEE Computer Society, 2004.
6. Pierre L'Ecuyer. Modeling and optimization problems in contact centers. In *QEST*, pages 145–156. IEEE Computer Society, 2006.
7. Dale Miller. A logical analysis of modules in logic programming. *The Journal of Logic Programming*, 6(1 and 2):79–108, January/March 1989.
8. L. Monteiro and A Porto. Contextual logic programming. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 284–299, Lisbon, 1989. The MIT Press.
9. Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for etl processes. In *DOLAP '02: Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 14–21, New York, NY, USA, 2002. ACM.