Brief Report on the Advanced Use of Prolog for Data Warehouses

François Pinet

INRAE, UR TSCF, Université Clermont Auvergne, 63178 Aubière, France; francois.pinet@inrae.fr

Abstract: Data warehouses have demonstrated their applicability in numerous application fields such as agriculture, the environment and health. This paper proposes a general framework for defining a data warehouse and its aggregations using logic programming. The objective is to show that data managers can easily express, in Prolog, traditional data warehouse queries and combine data aggregation operations with other advanced Prolog features. It is shown that this language provides advanced features to aggregate information in an in-memory database. This paper targets data managers; it shows them the direct writing of data warehouse queries in Prolog using an easily understandable syntax. The queries are not necessarily in an optimal form from a processing point of view, but a data manager can easily use or write them.

Keywords: data warehouse; prolog; data aggregation

1. Introduction

As mentioned in [1], the proposals of [2] and [3] established operational bases for reasoning from first-order logic formulas. This work favored the advent of logic programming and its most emblematic language; namely, Prolog [4]. Rule-based reasoning systems have remained important in the field of artificial intelligence until today.

This paper extends the work of [1] by testing several advanced uses of Prolog for developing data warehouses. A data warehouse is a specific type of database used to integrate, accumulate and analyze data [5,6]. Information from different databases is loaded into a data warehouse for combined analyses. These data are organized in analysis dimensions (time dimension, space dimension, descriptive dimensions, etc.). Indicators are calculated by aggregating a measure according to these dimensions.

In Prolog, the data used for the reasoning are generally all loaded in the memory [1]. The coupling between Prolog and databases has been carefully studied; the objective was to show how the features offered by Prolog could be used with large volumes of data. Prolog integrates specific functionalities that can be interesting for processing data (recursive queries, functions on graphs, constraint solvers, natural language processing, etc.).

Today, with the increase in computer RAM and the advent of in-memory databases, Prolog has become a good candidate for reasoning in databases. Based on this observation, this paper shows how to implement in-memory data warehouses in Prolog and focuses on their main function, data aggregation [6]. The objective was to propose a simple method to model this type of query by directly exploiting the existing functionalities of Prolog. This paper is a brief report that opens the way for the use of Prolog for data warehouse queries. This paper targets data managers; it shows them the direct writing of data warehouse queries in Prolog using an easily understandable syntax. The queries are not necessarily in an optimal form from a processing point of view, but a data manager could easily use or write them. More generally, the use and the adaptation of computer-based languages for specific application fields have been a prolific research topic over the years. For example, domain-specific modeling can be related to the use of Prolog for natural processing [7], C++ for design pattern definitions [8], OCL for spatial relation constraints [9], Java for add-on developments [10] and UML for serious games [11,12].
The paper is organized as follows. Section 2 presents the main existing contributions related to data warehouses and Prolog. Section 3 provides a case study. Section 4 shows the fundamental concepts for representing and querying data warehouses with Prolog. Section 5 compares the proposed Prolog-based queries with the SQL syntax. Sections 6–9 provide more advanced queries and illustrate the advantages of using Prolog in data warehouse queries. Section 10 is the conclusion, indicating future work.

2. Related Work

Prolog is based on first-order logic, which is a formalism to represent knowledge by logic formulas. The syntax of first-order logic includes logical symbols such as universal and existential quantifiers, variables, predicates, conjunctions, disjunctions and implications [13]. For example, the following first-order logic modeling represents that several humans are drivers: \( \exists X (\text{human}(X) \land \text{driver}(X)) \). The following expression models that in every country \( X \), there is an inhabitant \( Y \) who lives in \( X \): \( \forall X (\exists Y (\text{human}(Y) \land \text{live_in}(Y, X)) \leftarrow \text{country}(X)) \).

To make a parallel with the field of databases, Prolog can be used to model both data and the rules to reason with them. As with relational databases, it is based on the closed-world assumption that draws negative conclusions in the case of a lack of positive information [14]. The absence of information in a logic program implies that this information is false. Prolog allows a logic program to be defined by one or several rules, e.g., \( a_0 \leftarrow a_1 \land \ldots \land a_n \). As reminded in [14], this type of rule is equivalent to \( a_0 \lor \neg a_1 \lor \ldots \lor \neg a_n \), where \( a_0, \ldots, a_n \) are formulas. All variables in a formula are universally quantified over the whole formula; the atomic formula \( a_0 \) is the head of the clause. For example, in the context of databases, \( a_i \) can be \( \text{relation}(t_1, \ldots, t_p) \), where each \( t \) is a constant or a variable. A clause with an empty body and without variables can be viewed as a tuple of a relational database. A query \( q \) (i.e., a goal) can be written as \( \leftarrow b_1, \ldots, b_m \). The logical meaning of a query can be explained by referring to the equivalent universally quantified formula [14]: \( \forall X_1 \ldots \forall X_n \neg (b_1 \land \ldots \land b_m) \), where \( X_i \) is the variable that occurs in \( (b_1 \land \ldots \land b_m) \). It is equivalent to \( \neg \exists X_1 \ldots \exists X_n (b_1 \land \ldots \land b_m) \). For query processing, Prolog implements a top-down evaluation of the rules. Intuitively speaking, to process a query \( Q \), the system tries to unify each \( b_i \) with the head of the rules \( a_0 \leftarrow a_1 \land \ldots \land a_n \). If this unification is possible, a variable instantiation is propagated into the body of the rules. One subquery (i.e., a subgoal) proceeds for each \( a_i \) in \( a_1, \ldots, a_n \) in a left to right order. A query succeeds for a given variable instantiation depending on whether the unification succeeds. For example, suppose the following Prolog program:

\[
\begin{align*}
\text{p}(V) & \leftarrow \text{s}(V), \text{t}(V). \\
\text{s}(a). \\
\text{s}(b). \\
\text{t}(b).
\end{align*}
\]

\( V \) is a variable; \( a \) and \( b \) are constant. \( \leftarrow \) is the implication. The query \( \text{p}(X) \) will return \( \text{p}(b) \) as the result, as it is the only case where a variable unification succeeds. Figure 1 shows the Prolog evaluation tree of the query for the variable instantiation \( X = V = a \). This instantiation fails because \( \text{t}(a) \) is not in the program (and this information cannot be deduced from the program by reasoning). Consequently, \( \text{p}(a) \leftarrow \text{s}(a), \text{t}(a) \) fails. In the closed-world assumption, the absence of information implies it is false. The rule body is evaluated in a left to right order: \( \text{s}(X) \), then \( \text{t}(X) \). Figure 2 shows that \( \text{p}(b) \leftarrow \text{s}(b), \text{t}(b) \) succeeds. Thus, \( \text{p}(b) \) is the result of the query.
Prolog allows the definition of recursive rules (containing the same predicates in the head and the body of the rules). This mechanism can easily be exploited to calculate the transitive closure in a graph. The following rules define the transitive closure in a directed graph (see the example in [15]):

\[
\text{connected}(N, N) \ . \\
\text{connected}(N1, N2) \ :- \ \text{edge}(N1, L), \text{connected}(L, N2) \ .
\]

We considered that the direct links between the different vertices of the graph were modeled by the “edge” predicate; for example: edge(a, b); edge(b, c); and edge(a, e). The “connected” predicate could be used to determine the transitive closure. The query “connected(X, Y)” would compute all the results.

In terms of formalization, the basic operations of relational algebra can easily be expressed in the form of a conjunctive query; i.e., a rule. Consider the Prolog rule of \( r_1(X, Z) \ :- r_2(X, Y), r_3(Y, Z, c) \) with \( X, Y, Z \) variables; \( c \) is a constant. This rule corresponds with: (1) an equi-join operation between relations \( r_2 \) and \( r_3 \) because \( r_2(X, Y) \) and \( r_3(Y, Z, c) \) share a common variable (namely, \( Y \)); (2) a selection operation (the last attribute of the relation \( r_3 \) must be equal to the constant \( c \)); and (3) a projection operation of the attributes \( X \) and \( Z \), if one sees \( r_1 \) as the relation resulting from the query.

Prolog has a strong theoretical basis related to first-order logic, but it also incorporates several very practical features that are needed to write code. It integrates several operations that are not related to first-order logic; for example, it is possible to define input/output operations in rules or queries in order to write or read data streams. This type of operation is executed when it is reached in the execution tree. Data structures such as lists of elements can also be used to facilitate the concrete development of programs.

The use of Prolog to aggregate data was briefly discussed in an example presented in [16]. The example given in [16] is short and does not deal with all cases of aggregations. Based on the information provided in [16], the short communication presented in [1] introduces the idea of a more complete query pattern to represent the aggregations of data warehouses and shows applications for geo-referenced data.

Datalog can be viewed as an alternative to Prolog for databases [17]. Datalog provides a query language for deductive databases. Prolog proposes a top-down evaluation of the rules (from the head of the rules to their body) whereas Datalog usually implements a bottom-up evaluation (from the body of the rules to their head); the latter method is considered to be more suitable for data batch processing. The Datalog reasoning process
can be optimized by methods rewriting the rules at the run-time (see the methods of magic sets in [18]). The Datalog Educational System is an advanced implementation of Datalog that proposes extensions for computing aggregation operations (such as GROUP BY functions) [19,20].

The contribution of this paper was to show how to natively use the Prolog language to implement in-memory data warehouses. In the present paper, the first idea presented in [1] was extended to illustrate more advanced queries that highlighted the advantages of using Prolog for creating data warehouses. The coupling between data aggregations and advanced Prolog features was shown (recursive integrity constraint checking, a numeric constraint solver, graph-based calculations and data format conversions). In the present paper, SWI-Prolog was used, which was the current reference version for this language [21].

3. Case Study Example

We illustrated our proposal with an example. Figure 3 shows a multidimensional logical model presenting the facts and the dimensions of a data warehouse. The fact contained a measure attribute that could be aggregated through dimension levels. The fact class presented the measure “sale goals” linked to salespersons and products (e.g., products such as cars and trucks). Each salesperson had a sales goal that she/he had to reach for each product she/he could sell. In the fact relation, the “salespersonID” and “productID” foreign keys came from the salesperson and product relation. A salesperson could have a line manager (“salespersonManagerID” attribute) who was another salesperson. The product could be aggregated in the product types (e.g., the product categories). The attributes “productTypeRate” and “productDuration” are explained and used in Section 7.

![Figure 3. Sale multidimensional logical model.](image)

Below is an example of an instance of this multidimensional model in Prolog. The attribute ordering was the same in Prolog and in the logical model of Figure 3. In a traditional relational approach, this database would be in a fourth normal form.

```
salesperson('idsp01', 'Peter', 'idsp02').
salesperson('idsp02', 'James', 'idsp03').
salesperson('idsp03', 'Bill', 'none').
salesperson('idsp04', 'John', 'idsp03').
product('idprdtP1', 'GTFC', 1).
product('idprdtP2', 'DDAR', 1).
product('idprdtP3', 'X11', 2).
product('idprdtP4', 'X12', 3).
product('idprdtP5', 'WFG', 3).
```
product('idprdtP6', 'FGHY', 4).
product('idprdtP7', 'CFVG', 5).
product('idprdtP8', 'X13', 6).
productType(1, 0.05, 5).
productType(2, 0.045, 5).
productType(3, 0.055, 6).
productType(4, 0.05, 4).
productType(5, 0.07, 5).
productType(6, 0.035, 6).
fact('idsp01', 'idprdtP1', 100000).
fact('idsp01', 'idprdtP3', 550000).
fact('idsp01', 'idprdtP5', 500000).
fact('idsp02', 'idprdtP2', 300000).
fact('idsp03', 'idprdtP2', 500000).
fact('idsp04', 'idprdtP4', 100000).
fact('idsp04', 'idprdtP5', 350000).

Traditional data warehouse queries consist of aggregating a measure (e.g., saleGoal in the example) according to the dimension levels (e.g., by productId, salespersonID or productTypeID). Examples of numeric aggregation functions are sum, average and count.

4. General Form for a Data Warehouse Query in Prolog

In order to represent an aggregation of data in Prolog, we used the aggregate operator of Prolog [1]. It was used to calculate the aggregates from the logical predicates. In order to use it to compute aggregation queries in a data warehouse, it was necessary to specify the joins between the relations. To do this, we exploited the link shown in Section 2 between the relational algebra and the conjunctive queries. We proposed the following query pattern expressed in Prolog:

\[
\text{aggregate}(\text{aggregation\_function\_1}, \ldots, \text{aggregation\_function\_N}, \text{Attribute\_1} \hat{\ldots} \text{Attribute\_M}\text{(relation\_1}, \ldots, \text{relation\_O}, \ldots, \text{condition\_1}, \ldots, \text{condition\_P}), \text{aggregation\_result\_1}, \ldots, \text{aggregation\_result\_N}).
\]

\text{aggregation\_function\_i} and \text{aggregation\_result\_i} are, respectively, an aggregation function (e.g., count and sum) and the variable that stores the result of the aggregation obtained using this function. It was possible to use several aggregation functions in the same query. \text{relation\_1}, \ldots, \text{relation\_O} were the relations needed for the aggregation. The joins between the relations were represented in the same manner as traditional conjunctive queries (\text{relation\_1}, \ldots, \text{relation\_O} corresponded with \text{relation\_1} \land \ldots \land \text{relation\_O}). \text{Attribute\_1}, \ldots, \text{Attribute\_M} were the attributes included in \text{relation\_1}, \ldots, \text{relation\_O} that were not used for the aggregation. In SQL queries, the grouping attributes are specified in the GROUP BY clause; in Prolog, the attributes that are not used for the grouping are specified. Thus, by default, the Prolog aggregate operator grouped together all the attributes present in \text{relation\_1}, \ldots, \text{relation\_O}. The \text{Attribute\_1} \hat{\ldots} notation allowed the exclusion of certain attributes from the grouping. \text{condition\_i} was used to specify the conditions (such as the relational algebra selection).

We provide here a few examples of basic data warehouse queries. The following Prolog expressions produced the sum of the sales goals by the salespersons.

\[
\text{aggregate(sum(SaleGoal), ProductID} \hat{\ldots} \text{(fact(SalespersonID,ProductID,SaleGoal))}, SalesGoalSumByPerson)\).
\]

The results were:
SalespersonID = idsp01, SalesGoalSumByPerson = 1150000 ;
SalespersonID = idsp02, SalesGoalSumByPerson = 300000 ;
SalespersonID = idsp03, SalesGoalSumByPerson = 500000 ;
SalespersonID = idsp04, SalesGoalSumByPerson = 450000.

Here is a query to compute the sum by salesperson excluding the product idprdtP5:
aggregate(sum(SalesGoal),
ProductID\"fact(SalespersonID,ProductID,SalesGoal),
ProductID='idprdtP5',
SalesGoalSumByPerson).
The query below calculated the average of the sales goals by salespersons (combining the count and sum functions).
aggregate((count,sum(SalesGoal)),
SalesGoal\"ProductID\"fact(SalespersonID,Product, SalesGoal)),
(SalesGoalCountByPerson,SalesGoalSumByPerson)),
SalesGoalAvgByPerson is SalesGoalSumByPerson/SalesGoalCountByPerson.
The sum of the sales goals by salesperson and product type (with relation joins) was calculated by:
aggregate(sum(SalesGoal),
ProductID\"SalespersonManagerID"ProductName\"(fact(S,S,PR,SG),
salesperson(S,S,SM),product(PR,PN,PT)),RES).
The results were:
SalespersonID = idsp01, SalespersonName = 'Peter', ProductTypeID = 1,
SalesGoalSumByPersonAndProductType = 100000 ;
SalespersonID = idsp01, SalespersonName = 'Peter', ProductTypeID = 2,
SalesGoalSumByPersonAndProductType = 550000 ;
SalespersonID = idsp01, SalespersonName = 'Peter', ProductTypeID = 3,
SalesGoalSumByPersonAndProductType = 500000 ;
SalespersonID = idsp02, SalespersonName = 'James', ProductTypeID = 1,
SalesGoalSumByPersonAndProductType = 300000 ;
SalespersonID = idsp03, SalespersonName = 'Bill', ProductTypeID = 1,
SalesGoalSumByPersonAndProductType = 500000 ;
SalespersonID = idsp04, SalespersonName = 'John', ProductTypeID = 3,
SalesGoalSumByPersonAndProductType = 450000.

5. Comparison with the SQL Syntax

Note that in the queries of Section 4, term ordering inside the relations was used to identify an attribute. In other words, an attribute was identified thanks to its position in a relation. In the example above, explicit variable names were used, but more concise variable names could also be defined in order to reduce the query verbosity. Thus, the previous query could also be written in a very direct manner:
aggregate(sum(SG),
PR"SM"PN"(fact(S,PR,SG),salesperson(S,SM),product(PR,PN,PT)),RES).
The use of the same variable name inside different relations corresponded with an equi-join operation between these relations.
The equivalent query in SQL is:
SELECT
Fact.SalespersonID, SalespersonName, ProductTypeID, sum(SalesGoal) as SG
FROM Fact, Salesperson, Product
WHERE
Salesperson.SalespersonID=Fact.SalespersonID and
Product.ProductID=Fact.ProductID
GROUP BY
Fact.SalespersonID, SalespersonName, ProductTypeID
The verbosity of the SQL query (240 characters) is higher than the Prolog version (87 characters). This is due to: (1) the equi-join writing in Prolog, which has a very direct manner; and (2) the use of attribute positions inside the relations in Prolog instead of the use of attribute names.
6. Recursive Definition

The data instance in Section 3 shows that there was a hierarchy of salespersons (defined in the salesperson relation). As indicated in Figure 4, idsp02 and idsp03 were managers because they had a successor in the hierarchy.

```
Figure 4. Salesperson hierarchy (see instances in Section 3).
```

Prolog allows the very easy definition of the transitive closure of a graph. Consequently, the rules can define the concept of a manager. The recursive definitions were:
- S2 is the manager of S1, when S2 is defined as the manager of S1 in the salespersons;
- S3 is the manager of S1, when S2 is defined as the manager of S1 in the salespersons and S3 is the manager of S2.

The corresponding Prolog rules were:
```
manager(S1, S2) :- salesperson(S1,_,S2).
manager(S1, S3) :- salesperson(S1,_,-S2), manager(S2, S3).
```

Based on this definition, it was possible to aggregate the sales goals by managers; i.e., for each manager, the sum of all the sales goals of people under her/his responsibility.
```
aggregate(sum(SalesGoal), SalespersonID"ProductID"(fact(SalespersonID,ProductID,SalesGoal),
manager(SalespersonID,SalespersonManagerID)),
SalesGoalSumByManager).
```

Prolog can check integrity constraints [9,22,23]. For example, a cycle is not allowed for managers. An extended version of the manager rules were written in order to check the cycle.
```
manager(S1, S2, _ ) :- salesperson(S1,_,S2).
manager(S1, S3, L) :- salesperson(S1,_,S2), (member(S1,L)-write("cycle detected involving "), writeln(S1); manager(S2, S3, [S1|L])).
```

The Prolog expression “manager (_,_,[ ])” checked if a cycle occurred.

7. Constraint Solver

This section illustrates the use of the CLP(R) solver [24], which is an SWI-Prolog module to handle constraints over real numbers. Suppose that, in the example, the products are sold on credit. The credit rate and credit duration are stored in the relation productType (see Figure 3). All the products with the same type have the same credit rate and duration. It is possible to calculate the sales goal without the credit cost by sales person and product type by directly using a formula such as SalesGoal=SalesWithoutCreditCost*(Rate/(1-(1+Rate)^(-N)))*N. In this case, the CLP(R) solver could determine the value of SalesWithoutCreditCost based on the values of the other bound variables.
```
aggregate(sum(SalesGoal), ProductName"ProductID"(fact(SalespersonID,ProductID,SalesGoal),
product(ProductID,ProductName,ProductTypeID),
productType(ProductTypeID,Rate,N)),SalesGoal),
{SalesGoal=SalesWithoutCreditCost*(Rate/(1-(1+Rate)^(-N)))*N}.
```

The recursive definitions were:
```python
# Possible to calculate the sales goal without the credit cost by sales person and product type by

# Consequently, the rules can define the concept of a manager.

# salesperson hierarchy (see instances in Section 3).

# Prolog allows the very easy definition of the transitive closure of a graph. Consequently, the rules can define the concept of a manager.

# The recursive definitions were:

# S2 is the manager of S1, when S2 is defined as the manager of S1 in the salespersons;

# S3 is the manager of S1, when S2 is defined as the manager of S1 in the salespersons and S3 is the manager of S2.

# The corresponding Prolog rules were:

# manager(S1, S2) :- salesperson(S1,_,S2).

# manager(S1, S3) :- salesperson(S1,_,-S2), manager(S2, S3).

# Based on this definition, it was possible to aggregate the sales goals by managers; i.e.,

# for each manager, the sum of all the sales goals of people under her/his responsibility.

# aggregate(sum(SalesGoal), SalespersonID"ProductID"(fact(SalespersonID,ProductID,SalesGoal),

# manager(SalespersonID,SalespersonManagerID)),

# SalesGoalSumByManager).

# Prolog can check integrity constraints [9,22,23]. For example, a cycle is not allowed for managers.

# An extended version of the manager rules were written in order to check the cycle.

# manager(S1, S2, _ ) :- salesperson(S1,_,S2).

# manager(S1, S3, L) :- salesperson(S1,_,S2), (member(S1,L)-write("cycle detected involving "), writeln(S1); manager(S2, S3, [S1|L])).

# The Prolog expression “manager (_,_,[ ])” checked if a cycle occurred.

# Constraint Solver

# This section illustrates the use of the CLP(R) solver [24], which is an SWI-Prolog module to handle constraints over real numbers. Suppose that, in the example, the products are sold on credit. The credit rate and credit duration are stored in the relation productType (see Figure 3). All the products with the same type have the same credit rate and duration. It is possible to calculate the sales goal without the credit cost by sales person and product type by directly using a formula such as SalesGoal=SalesWithoutCreditCost*(Rate/(1-(1+Rate)^(-N)))*N.

# In this case, the CLP(R) solver could determine the value of SalesWithoutCreditCost based on the values of the other bound variables.

# aggregate(sum(SalesGoal), ProductName"ProductID"(fact(SalespersonID,ProductID,SalesGoal),

# product(ProductID,ProductName,ProductTypeID),

# productType(ProductTypeID,Rate,N)),SalesGoal),

# {SalesGoal=SalesWithoutCreditCost*(Rate/(1-(1+Rate)^(-N)))*N}.
```
The results were:
SalesGoal = 100000, SalespersonID = idsp01, ProductTypeID = 1, Rate = 0.05, N = 5, SalesWithoutCreditCost = 86589.53341261645 ;
SalesGoal = 550000, SalespersonID = idsp01, ProductTypeID = 2, Rate = 0.045, N = 5, SalesWithoutCreditCost = 482897.44188721693 ;
SalesGoal = 500000, SalespersonID = idsp01, ProductTypeID = 3, Rate = 0.055, N = 6, SalesWithoutCreditCost = 416294.19238697476 ;
SalesGoal = 300000, SalespersonID = idsp02, ProductTypeID = 1, Rate = 0.05, N = 5, SalesWithoutCreditCost = 259768.60023784934 ;
SalesGoal = 500000, SalespersonID = idsp03, ProductTypeID = 1, Rate = 0.05, N = 5, SalesWithoutCreditCost = 432947.6670630823 ;
SalesGoal = 450000, SalespersonID = idsp04, ProductTypeID = 3, Rate = 0.055, N = 6, SalesWithoutCreditCost = 374664.7731482773 .

For example, for the first result, the equation 100,000 = SalesWithoutCreditCost * (0.05/(1-(1+0.05)(−5)))^5 was solved. The result was SalesWithoutCreditCost = 86,589.53341261645.

8. Clique-Based Aggregation

We illustrate here the use of another advanced function in Prolog. This section shows that one can aggregate measures by groups dynamically calculated in the query. More precisely, we provide an example based on the work of [25] that allows graph analyses in Prolog. The proposed functions on graphs can easily be integrated in aggregation queries. For example, the graph of Figure 5 shows the similarity between the product types. There is a link between two product types when there is a significant similarity link between these types; e.g., between a tanker truck and a fuel tank or between a tanker truck and a delivery truck.

Figure 5. Similarity link between product types.

The query below calculates the sum of the sales goals by the group of similar product types and by the salesperson. The types were grouped according to the graph cliques found in Figure 5. Figure 5 contains two cliques (1,2,3) and (3,4,5), which were two complete subgraphs (all vertices were connected in each subgraph) [25]. The graph of Figure 5 was represented inside the query by an adjacency matrix. The predicate clique_find_multi automatically calculated the cliques for this graph.

aggregate(sum(SalesGoal),
ProductID*ProductName*ProductTypeID*
clique_find_multi(10,
[[0,1,1,0,0,0],[0,0,1,1,0,0],[1,0,1,0,0,0],[1,0,1,0,0,0],[0,0,1,1,0,0],[0,0,1,0,0,0]],Clique),
member(ProductTypeID,Clique), fact(SalespersonID,ProductID,SalesGoal),
product(ProductID,ProductName,ProductTypeID)), SalesGoalSumByProductTypeClique).

The results were:
Clique = [1, 2, 3], SalespersonID = idsp01,
SalesGoalSumByProductTypeClique = 1150000 ;
Clique = [1, 2, 3], SalespersonID = idsp02,
SalesGoalSumByProductTypeClique = 300000 ;
Clique = [1, 2, 3], SalespersonID = idsp03,
SalesGoalSumByProductTypeClique = 500000 ;
Clique = [1, 2, 3], SalespersonID = idsp04,
SalesGoalSumByProductTypeClique = 450000 ;
Clique = [3, 4, 5], SalespersonID = idsp01,
SalesGoalSumByProductTypeClique = 500000 ;
Clique = [3, 4, 5], SalespersonID = idsp04,
SalesGoalSumByProductTypeClique = 450000.

9. Format Conversion

Different rules can be directly defined in Prolog to convert the data formats. The data of Section 3 were in a relational form; it was possible to convert them into a document-oriented format, for example. In this format, the data could be nested according to the dimensions. The query below converted the data into documents on the product dimension. The predicate named “assert” inserted the documents into the memory.

```
fact(SalespersonID, ProductID, SalesGoal),
product(ProductID,ProductName,ProductTypeID),
productType(ProductTypeID,Rate,Duration),
assert(fact_doc(fact(SalespersonID,SalesGoal,product(ProductID,ProductName,
productType(ProductTypeID,Rate,Duration))))).
```

In this case, the new data inserted were:

```
fact_doc(fact(idsp01,100000,product(idprdtP1,'GTFC',productType(1,0.05,5)))).
fact_doc(fact(idsp01,550000,product(idprdtP3,'X11',productType(2,0.045,5)))).
fact_doc(fact(idsp01,500000,product(idprdtP5,'WFG',productType(3,0.055,6)))).
fact_doc(fact(idsp02,300000,product(idprdtP2,'DDAR',productType(1,0.05,5)))).
fact_doc(fact(idsp02,100000,product(idprdtP4,'X12',productType(3,0.055,6)))).
fact_doc(fact(idsp03,350000,product(idprdtP5,'WFG',productType(3,0.055,6)))).
fact_doc(fact(idsp03,150000,product(idprdtP5,'WFG',productType(3,0.055,6)))).
fact_doc(fact(idsp04,300000,product(idprdtP2,'DDAR',productType(1,0.05,5)))).
fact_doc(fact(idsp03,550000,product(idprdtP5,'WFG',productType(3,0.055,6)))).
fact_doc(fact(idsp04,100000,product(idprdtP4,'X12',productType(3,0.055,6)))).
fact_doc(fact(idsp04,350000,product(idprdtP5,'WFG',productType(3,0.055,6)))).
```

10. Conclusions

Data warehouses have demonstrated their applicability in numerous application fields such as business, health, agriculture and the environment. In the present paper, we proposed a general framework for the definition of a data warehouse and its aggregations in Prolog. We illustrated a few advanced uses of Prolog in this context. Our objective was to show that one can express, in Prolog, the typical queries of data warehouses and that one can easily combine aggregations with other advanced features in Prolog. A main motivation for a data manager is to natively use the advanced features provided by logic programming in addition to the query capabilities. The advantage for the data manager is to handle one single language (Prolog) instead of several technologies (SQL+Java, for example). The relation joins can also be expressed in Prolog in a very direct manner using common variables between the predicates. The attributes of a database modeled in Prolog can have complex structures; for example, the form of a logical formula.
Numerous other capabilities are available in Prolog [21]. This paper illustrates a few of them. Prolog provides very interesting features to aggregate information in its in-memory database. A future study may be to have this approach tested by several data warehouse designers and to compile a survey to evaluate their acceptance of this new technical solution.

The paper focused on the modeling aspect; in future work, it would be interesting to evaluate the execution time performance of Prolog for data warehouses according to different volumes of data. The performance could be compared with other in-memory data management systems according to the datasets usually exploited for data warehouse benchmarks. A future perspective could also be to integrate Prolog into traditional online analytical processing (OLAP) architecture. A data warehouse is just one component that processes aggregation queries and provides results. It can be inserted into a complete OLAP architecture. In this architecture, different software components interact in order to manage all the steps needed by users to integrate, query and visualize the data. A classical OLAP architecture is shown in Figure 6. First, data sources are integrated into a database (for example, a relational database) using an extraction, transformation and loading process. Second, the end-user navigates the data, thanks to an OLAP client (for example, JRubik) using a dedicated human–machine interface. The end-user can trigger OLAP operations such as drill-down and roll-up to change the data aggregation levels. The operations are processed using an OLAP server (for example, Mondrian). This server interacts with the database by sending the aggregation queries to the database and receives the results. A future goal is to use Prolog instead of traditional relational databases (and SQL) to store and query the data. A future study may be to create an interface between OLAP servers and Prolog-based data warehouses, and also to provide the possibilities to model and define complex queries in an OLAP architecture such as the ones presented in this paper.

Figure 6. OLAP architecture.

Funding: This research received no external funding.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The author declares no conflict of interest.

References


