Title: Backward chaining inference as a database stored procedure – the experiments on real-world knowledge bases

Author: Tomasz Xięski, Roman Simiński

Citation style: Xięski Tomasz, Simiński Roman. (2018). Backward chaining inference as a database stored procedure – the experiments on real-world knowledge bases. “Journal of Information and Telecommunication” (Vol. 2, NO. 4 (2018), s. 449-464), doi 10.1080/24751839.2018.1479931

JOURNAL OF INFORMATION AND TELECOMMUNICATION
2018, VOL. 2, NO. 4, 449–464
https://doi.org/10.1080/24751839.2018.1479931

Backward chaining inference as a database stored procedure – the experiments on real-world knowledge bases

Tomasz Xięski and Roman Simiński

Institute of Computer Science, University of Silesia, Sosnowiec, Poland

ABSTRACT

In this work, two approaches of backward chaining inference implementation were compared. The first approach uses a classical, goal-driven inference running on the client device – the algorithm implemented within the `KBExpertLib` library was used. Inference was performed on a rule base buffered in memory structures. The second approach involves implementing inference as a stored procedure, run in the environment of the database server – an original, previously not published algorithm was introduced. Experiments were conducted on real-world knowledge bases with a relatively large number of rules. Experiments were prepared so that one could evaluate the pessimistic complexity of the inference algorithm. This work also includes a detailed description of the classical backward inference algorithm – the outline of the algorithm is presented as a block diagram and in the form of pseudo-code. Moreover, a recursive version of backward chaining is discussed.

ARTICLE HISTORY
Received 30 November 2017
Accepted 20 May 2018

KEYWORDS
Expert systems; knowledge bases; backward chaining inference; databases

1. Introduction

Knowledge-based systems are still popular and practically used tools for solving ill-structured problems. Rules are among the most popular forms of representing knowledge in the field of intelligent information systems, regardless of the development of different knowledge representations. Forward and backward chaining inference algorithms are also popular in the real-world applications (Akerkar & Sajja, 2010). The number of applications which utilize rule bases and methods of inference grows, but unfortunately the number of tools for building knowledge-based systems increases much more slowly (Sajja & Akerkar, 2010). The well-known systems, such as JESS (2016), CLIPS (2016), DROOLS (2016) or EXSYS (2016), are usually described as the tools for implementing domain knowledge-based systems. What is more, commercial expert system development tools have been extended to offer web-based development capabilities.

A selected part of our research focused on the development of new methods and tools for building knowledge-based systems is presented in this work. In our previous work (Simiński & Nowak-Brzezińska, 2016), we introduced the `KBExplorer` system – a

CONTACT Tomasz Xięski tomasz.xieski@us.edu.pl Institute of Computer Science, University of Silesia, Będzińska 39, 41-200 Sosnowiec, Poland

WWW application which allows the user to create, edit and share rule-based knowledge bases. We also introduced (in a separate paper) the inference engine, which is provided by the `KBExpertLib` – a software library which allows programmers to implement domain knowledge-based systems using the Java programming language (Simiński, 2016). Next part of the realized project is the `KBExploratorDesktop` (Nowak-Brzezińska & Simiński, 2015) – a desktop application which allows to analyse knowledge bases created by the `KBExplorer`.`KBExploratorDesktop` internally uses the `KBExpertLib` library and is implemented as a standard JavaFX GUI program. The prototype version of `KBExplorer` and the demo version of `KBExploratorDesktop` are available online at http://kbexplorer.ii.us.edu.pl. Figure 1 presents the main software modules of the proposed distributed expert system shell. The system is still under development – enhanced versions of the software are in the test phase.

The migration of information systems from the classic desktop software to the web application can be observed as a permanent trend. This trend also applies to the knowledge-based systems. The 'webalization' of information systems causes many practical and implementation problems and challenges, but we can also identify in this field a number of interesting research problems. In this paper, we present a modified goal-driven inference algorithm for web knowledge-based systems.

The motives behind this article are research and implementation works concerning the application of proposed tools for a weak hardware configuration – like in mobile and embedded devices or obsolete (but still frequently used) computers. The `KBExpertLib` library provides inference algorithms implemented in Java and working on the internal device's resources. The inference performed on such devices may be ineffective from the user's point of view, and for the large rule bases may be totally impossible. We propose a different approach: a client device sends the initial inference information to the server side over the internet and receives inference results. The comparison of time efficiency of these two approaches is the main goal of the article. This article presents new (and not published before) implementation issues, focused on the backward chaining inference. Presented considerations are the continuation of the previous, forward inference dedicated studies (Xięski & Simiński, 2017). This article is a part of a wider project

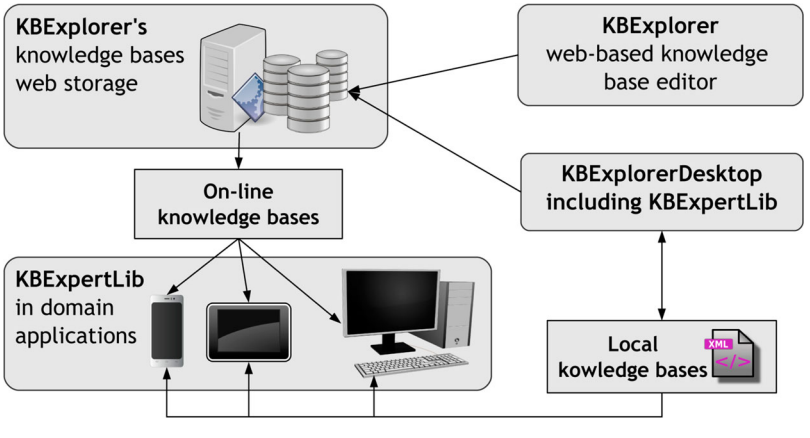

Figure 1. Main components of the distributed expert system shell.

involving both research and implementation works – in this work we analyse two different approaches mentioned earlier.

The first approach uses a classical, goal-driven inference running on the client device. An algorithm implemented within the `KBExpertLib` is used. The content of a particular knowledge base is retrieved from a local XML file or a relational database and is stored in the device's RAM – Figure 2 illustrates the discussed solution. Inference is performed on a rule base (buffered in memory structures), and the effectiveness of this process depends on the hardware configuration of the local machine and will vary due to the size of the knowledge base. This type of inference uses local resources (memory, processor) of the client's device and may consume a significant amount of energy (possibly supplied with a battery).

The second approach assumes that the inference process is being realized fully on the server side. A dedicated PHP implementation was previously analysed (Simiński & Manaj, 2015). In this work, utilization of stored procedures (within the database server) is considered. A client device uses Rest API services, sends the goal and facts to the server and receives information about the goal confirmation (and optional details) – Figure 3 illustrates the presented approach. The utilization of a server-side implementation minimizes the network traffic, as only a single request is necessary. The usage of database server's stored procedures ensures independence from the used programming tools – only a connection to the database server and a simple API is required. The main research goal of this paper was the experimental evaluation of the backward chaining inference algorithm implementation as a stored procedure and a comparison (of such implementation) with inference performed on preloaded knowledge bases (on local devices). Considered approaches are quite different from the implementation point of view.

Note that this is an extended version of our (Simiński & Xięski, 2017) INISTA 2017 paper. Additions and changes include:

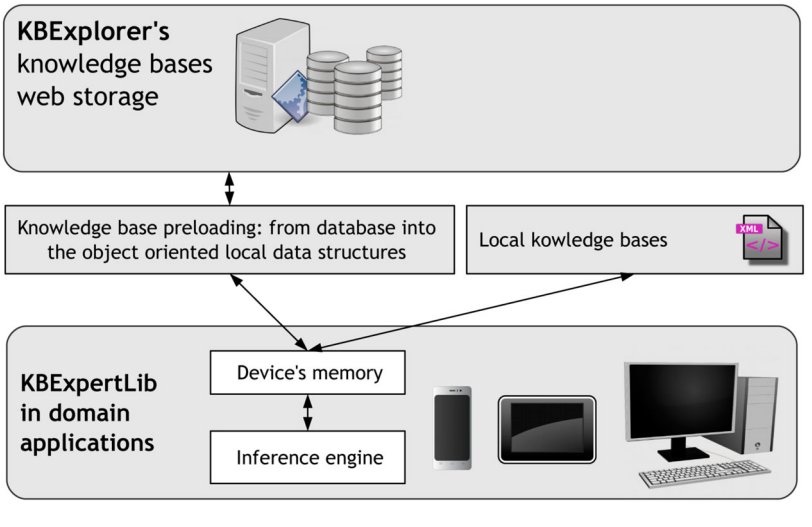

Figure 2. First approach: inference as a local process.

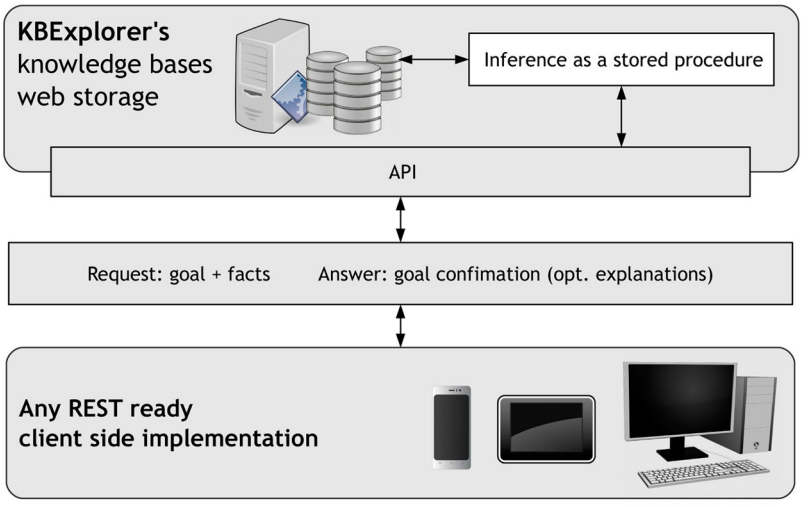

Figure 3. Second approach: inference as a remote process.

- a much more detailed description of the classical backward inference algorithm (see Section 2.2) with a newly created block diagram and revised pseudo-code, which is hard to find in the domain literature,
- an extended description of the KBExpertLib (see Section 3.1),
- new experiment number 3 (in Section 4) focused on the available database storage engines with a detailed analysis,
- a revised and extended version of the abstract, introduction and conclusions.

The organization of this paper is as follows. The next section presents background information – the formal knowledge base model, classical backward chaining inference algorithm and related works. Section 3 outlines the proposed methods and tools – the backward chaining inference algorithm as a stored procedure is introduced. Section 4 presents the experiments and their results. Finally, the conclusions section summarizes the paper.

2. Background information and related works

This section presents the formal description of a knowledge base, the backward chaining inference algorithm, a brief review of software tools related to this work and a short description of our own software implementation.

2.1. Knowledge base

The following formal description of a knowledge base is assumed in this work: a knowledge base is a pair $KB = (R,F)$ where R is a non-empty finite set of rules and F is a finite set of facts. $R = \{r_1,r_2,\ldots,r_n\}$, each rule $r \in R$ will have a form of Horn's clause: $r : p_1 \wedge p_2 \wedge \cdots \wedge p_m \rightarrow c$, where m is the number of literals in the conditional part of rule r, $m \geq 0$, p_i the ith literal in the conditional part of rule r, $i = 1m$, and c the literal of the decisional part of rule r.

For each rule $r \in R$ we define the following functions: $concl(r)$ – the value of this function is the conclusion literal of rule r: $concl(r) = c$; $cond(r)$ – the value of this function is the set of conditional literals of rule r: $cond(r) = \{p_1, p_2, \ldots p_m\}$, $literals(r)$ – the value of this function is the set of all literals of rule r: $literals(r) = cond(r) \cup \{concl(r)\}$. We will also consider the *facts* as clauses without any conditional literals. The set of all such clauses f will be called *set of facts* and will be denoted by F: $F = \{f : \forall_{f \in F} cond(f) = \emptyset \wedge f = concl(f)\}$. The rule $r \in R$ is *fireable* if each condition appearing in the premise of rule r is a fact: $fireable(r)$ *iff* $\forall_{c \in cond(r)} : c \in F$. Each fireable rule can be activated, the conclusion of activated rule is added to facts set F $-activate(r) : F = F \cup \{concl(r)\}$.

In this work, rule's literals will be denoted as pairs of attributes and their values. Let A be a non-empty finite set of conditional and decision attributes.[1] For every symbolic attribute $a \in A$ the set V_a will be denoted as the set of values of attribute a. Attribute $a \in A$ may be simultaneously a conditional and decision attribute. Also a conclusion of a particular rule r_i can be a condition in another rule r_j. It means that rules r_i and r_j are connected, and it is possible that inference chains may occur. The literals of the rules from R are considered as attribute-value pair (a, v), where $a \in A$ and$v \in V_a$.

2.2. Backward chaining inference

The inference algorithm selects some applicable rules to infer new facts and/or confirm established goals. When rules are examined by the inference engine, new facts are added to the fact base if its current content satisfies the conditions in the rules. The strategy of backward chaining is started from a goal and ends with a set of facts that leads to the given goal, and therefore, it is also known as a *goal-driven* strategy of the inference engine.

Backward chaining inference can be considered as a bottom-up procedure which starts with a main goal and queries the fact base about information which may satisfy the conditions contained in the rules. We basically go through the rules in the knowledge base looking for conclusions which match the query and if we find them, we can create new queries (adding new facts if necessary). Its complexity can be linear or less (taking into account the size of the knowledge base), depending on the implementation. The main idea and general description of a classical backward chaining inference algorithm have been repeatedly published, for example, in Grzymala-Busse (2012) and Ligeza (2006). But it is hard to find a detailed, step-by-step algorithm and for this reason we present a more detailed description in the form of a block diagram and pseudo-code.

Figure 4 illustrates a general approach to the backward inference method. Initially the algorithm checks whether the goal g is in the fact set ($g \in F$). If our goal is a fact, the inference is completed and its goal g is confirmed. Otherwise,[2] the rules matching the goal literal g are selected from whole rule base (later in this work these rules will be called *applicable rules*). Thus the rule set R is examined. If the selected rule set R is not empty, the algorithm looks for a fireable rule r. The first applicable rule can be selected or a more sophisticated rule selection strategy can be used.

If a fireable rule r was found, the conclusion of this rule is added to the fact set F. The algorithm considered in this work terminates when a fireable rule was found – the information about goal confirmation is stored in the RAM. In a more general case, the algorithm can consider other applicable rules using the backtracking strategy. Applicable rules are a

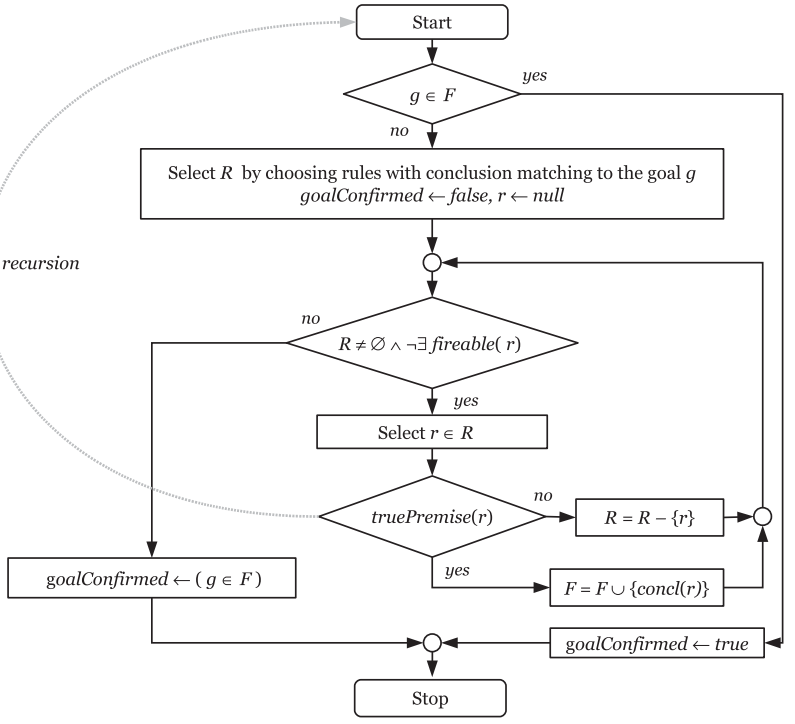

Figure 4. Backward inference – the general algorithm.

set of partial candidates for a goal confirmation, thus all the possible solutions for a determined goal can be obtained.

As mentioned earlier, the rule is fireable if each condition appearing in the premise of rule *r* is a fact. However, in the case of the backward inference very often the fact set is empty and only the inference goal must be specified. Therefore, backward inference engines pose the ability to determine facts during the inference process. Generally, there are two sources of facts:

- The knowledge base itself – the missing facts can be determined by reasoning.
- The environment of the knowledge base system – the missing facts can be acquired from human users, sensors, networks, databases or external systems.

The backward inference algorithm is a natural way of confirming the truth about the condition appearing in the rule's premise. The considered condition can be treated as a subgoal and the inference engine is able to confirm or reject this subgoal using the backward chaining algorithm. The process of subgoal confirmation may be realized iteratively or recursively. The recursive algorithm is a natural, clear and understandable way of presenting the backward chaining method, but unfortunately is hard to illustrate recursion in detail on block diagrams. For example, in Figure 4 a grey arrow indicates the process of subgoals confirmation. A more detailed description of the analysed inference method is presented in the form of pseudo-code in Algorithm 1. The line marked with {*} performs the recursive call of inference for a particular subgoal *w*.

When the recursive call of the backward inference rejects the subgoal, the second mentioned method of missing facts gathering is performed – the environment of the knowledge-based system is asked about the truth of the currently considered condition. If on the basis of the environment response, the truth about the condition cannot be determined, the algorithm terminates.

Later in this work we will consider recursive Algorithm 1 implemented in the Java programming language and an iterative version of the backward inference implemented as a stored procedure.

2.3. Related works

The knowledge-based systems were typically developed as desktop applications. Meanwhile, web applications have grown rapidly and have had a significant impact on the application of a traditional expert system. The detailed discussion and comparison of modern knowledge-based systems building tools go beyond the scope of this study. Selected aspects of such review can be found in Ruiz-Mezcua et al. (2011) and Mathkour et al. (2009) and also in Grove (2000), Duan et al. (2005) and Huntington (2000). In this work only a basic overview is presented. The JESS is a well-known and popular tool. It is the skeleton of expert systems developed by Sandia National Laboratories. JESS is written in Java and it is possible to run code in this language using JESS. It uses a syntax similar to Lisp (JESS, 2016). It is compatible with both Windows and Unix systems. Rules written using JESS are saved in the form of an XML file which must contain a *rule-execution-set* element (Canadas et al., 2010). JESS is a rule engine as well as a scripting language, which provides a console for programming and enables basic input/output operations. JESS is a forward chaining inference engine, it provides mechanisms that 'simulate' backward chaining.

EXSYS Corvid (EXSYS, 2016) is a software tool for building and fielding knowledge automation expert system applications. It is designed to be easy to learn and aimed at non-programmers. The Java-based EXSYS Inference Engine makes it simple to deploy systems on different platforms and to integrate them with external programs. The rules (in the knowledge base) are described simply in English and Algebra. Tree-structured logic diagrams are used to describe individual sections of the process. It's a combination of 'Logic Block' structures (which describe rules) and 'Command Blocks' that provide procedural control on system execution. Corvid's Inference Engine uses both backward and forward chaining algorithms.

Another commercial expert system building tool is XpertRule (XpertRule, 2016), which offers a Knowledge Builder Rules Authoring Studio. The XpertRule KBS interfaces over the Web with a thin client using Microsoft's Active Server Page technology. Applications developed using the Knowledge Builder Rules Authoring Studio can be generated as Java Script/HTML files for deployment as Web applications. The Web Deployment Engine is a JavaScript rules runtime engine which runs within a browser.

Similar concepts share the eXpertise2Go's Rule-Based Expert System, which provides free building and delivery tools that implement expert systems as Java applets, Java applications and Android apps (eXpertise2Go, 2016). Next interesting system is Drools – a Business Rules Management System solution. It provides a core Business Rules Engine, a

web authoring and rules management application (Drools Workbench) and an Eclipse IDE plugin for core development (DROOLS, 2016).

This work introduces another decision support system building tool – the `KBExplorator` system (Simiński & Nowak-Brzezińska, 2016). It is a web application and allows the user to create, edit and share rule knowledge bases. It is also connected with the `KBExpertLib` – a software library, which allows programmers to use different kinds of inference within any software projects implemented in the Java programming language. The `KBExplorer` works on the client side and requires only the usage of a typical modern web browser. Knowledge bases created by the user are stored in a relational database, and may be shared between the registered system's users (Simiński & Xięski, 2015). Moreover, it is also possible to download the knowledge base as an XML file (for further analysis). The `KBExpertLib` is a software library, which allows programmers to implement domain knowledge-based systems using the Java programming language. This library makes it possible to run different kinds of inference (classical and modified forward and backward chaining algorithms) on rule-based knowledge bases stored in the `KBExplorer` database or saved locally in the form of XML files (Xięski & Simiński, 2017) (see Figure 1); the `KBExpertLib` is also considered as a tool for the implementation of systems described in Przybyła-Kasperek & Wakulicz-Deja (2014) and Nowak-Brzezińska (2016).

3. Methods and tools

This section presents two approaches for backward chaining inference (illustrated in Figures 1 and 2) described earlier in the introduction.

3.1. Backward chaining inference as Java code

The package `KBExpertLib` is implemented in the Java programming language. It may be used on the server side of WWW application or embedded in the desktop Java applications, also on mobile devices. The `KBExpertLib` is object-oriented library, library's classes are divided into the packages: `kbcore` – the main, essential classes,`kbinfer` – classes providing classical and modified inference algorithms, `kbpartition` – classes allowing decomposition of rule bases, and `kbtools` – additional tool classes.

The `KBExpertLib` provides a backward inference presented by Algorithm 1, the algorithm is implemented as a member function of the `KBBackwardConsoleInferer` class. In this work, we consider inference without interaction with the system's environment and want to analyse the worst possible case in terms of inference time. The`KBBackwardConsoleInferer` is a base class for a specialized derived class, which has to provide a fact confirmation function (Algorithm 1: `environmentConfirmsFact`). Implementation of this function depends on a particular fact confirmation process. In research described later in the article, the implementation of this function is trivial – it always returns a *false* value, because we want to obtain the pessimistic inference algorithm complexity. Implementation of a backward chaining inference algorithm is typical and obvious – much more interesting is the implementation as a stored procedure, described in the next section.

3.2. Backward chaining inference as the stored procedure

This section presents the second of the earlier mentioned approaches of inference realization. This approach works in the database server layer as a stored procedure, which uses native properties of the MySQL engine.

The presented inference algorithm assumes that any selected goal will be confirmed after testing enough subgoals, because we are interested only in the worst-case scenario. What is more, the algorithm was repeated in the experiments sections, for all possible inference goals (distinct rule's conclusions). The pseudo-code of the described approach is presented in Algorithm 2 and the exemplary implementation of the stored procedure is as follows:

```
main_infer:BEGIN
DECLARE rId INT;
DECLARE FK_subgoalAttributeID INT;
DECLARE FK_subgoalValueID INT;
DECLARE subgoalOperator CHAR(3);
DECLARE subgoalRowID BIGINT;
DECLARE subgoalContinousValue VARCHAR(45);
DECLARE exitLoop BIT(1);
DECLARE CONTINUE HANDLER FOR 1329 SET exitLoop = 1;
SET exitLoop = 0; SET kbID = 1;
#Clear temporary tables
TRUNCATE TABLE subgoals;
TRUNCATE TABLE tempfacts;
#Insert the main inference goal to the subgoals table
INSERT INTO subgoals(FK_attributeID, operator, FK_valueID,
continousValue, FK_knowledgeBase) VALUES (3,' ==' , 3, NULL,
kbID);
read_loop: LOOP
#Select first subgoal
SELECT subgoalID, FK_attributeID, operator, FK_valueID,
continousValue INTO subgoalRowID, FK_subgoalAttributeID,
subgoalOperator, FK_subgoalValueID, subgoalContinousValue
FROM subgoals ORDER BY subgoalID DESC LIMIT 1;
IF exitLoop = 1 THEN
LEAVE read_loop;
END IF;
#Insert the current subgoal to the temporary facts table
INSERT INTO tempfacts SELECT NULL, FK_attributeID, operator,
FK_valueID, continousValue, kbID FROM subgoals WHERE
subgoalID = subgoalRowID;
#Delete the current subgoal
DELETE FROM subgoals WHERE subgoalID = subgoalRowID;
#Check if the current subgoal is a fact
IF NOT EXISTS (SELECT * FROM facts WHERE FK_attributeID =
FK_subgoalAttributeID AND operator = subgoalOperator AND
```

```
FK_valueID = FK_subgoalValueID AND continousValue <=>
subgoalContinousValue) THEN
#Get the ID and later on all premises of the rule which
#conclusion in the selected subgoal
SET rId = (SELECT FK_ruleId FROM attributeValue WHERE
FK_attributeID = FK_subgoalAttributeID AND
operator = subgoalOperator AND FK_valueID = FK_subgoalValueID
AND continousValue <=> subgoalContinousValue AND
isConclusion = 1 LIMIT 1);
#If there is no such rule, inference cannot complete
IF rId IS NULL THEN
SELECT "INFERENCE FAILURE";
LEAVE read_loop;
END IF;
#Insert selected premises to the subgoals table,
#excluding those which are facts or were already processed
INSERT INTO subgoals
(FK_attributeID, operator, FK_valueID, continousValue)
SELECT FK_attributeID, operator, FK_valueID, continousValue
FROM attributeValue t1 WHERE FK_ruleId = rId AND
isConclusion IS NULL AND NOT EXISTS
(SELECT 1 FROM tempfacts t2 WHERE t2.FK_attributeID =
t1.FK_attributeID AND t2.operator = t1.operator AND
t2.FK_valueID = t1.FK_valueID AND t2.continousValue <=>
t1.continousValue)AND NOT EXISTS
(SELECT 1 FROM subgoals t3 WHERE t3.FK_attributeID =
t1.FK_attributeID AND t3.operator = t1.operator AND
t3.FK_valueID = t1.FK_valueID AND t3.continousValue <=>
t1.continousValue );
END IF;
END LOOP;
#If all subgoals were confirmed inference is a success
IF exitLoop = 1 THEN
SELECT "INFERENCE SUCCESS";
END IF;
END
```

The above-mentioned implementation assumes that the initial set of facts is already known and the user has set the inference goal to a descriptor expressed internally as a pair of *attributeID*=3 and *valueID*=3 (which is also the conclusion of the first rule in the knowledge base). What is more, two temporary tables (*subgoals* and *tempfacts* storing, respectively, the current set of inference subgoals and a temporary set of facts, which will be valid if the inference succeeds) use internally the *MEMORY Storage Engine*. The *MEMORY Storage Engine* which is a part of the MySQL DBMS ensures that data in such tables will be kept in the server's RAM memory – this may be required due to many insert and delete operations.[3]

The main loop starts by selecting the first subgoal from the *subgoals* table – currently there is only the main inference goal (line 3.2 of Algorithm 2). The subgoal is inserted into the temporary set of facts (represented by the *tempfacts* table) and removed from the *subgoals* table (lines 3.2 and 3.2). If the current subgoal of the inference does not belong to the set of initial facts, the procedure selects the ID of the first (not analysed) rule from the knowledge base whose conclusion corresponds to the subgoal (line 3.2). If such a rule does not exist, the inference ends with failure. This is not the case, and so premises of the selected rule are added to the subgoals table (line ??), excluding those which are in the set of temporary facts (*tempfacts*) or were already processed (are included in the *subgoals* table). If all subgoals are confirmed, the main loop ends and inference is considered as a success. Entries from the *tempfacts* table can now be treated as new facts.

4. Experiments

This section presents the experiments performed on four real-world knowledge bases. The first used knowledge base (`bud4438`) is used by a builder company in Poland and currently consists of 4438 rules. There are 2802 symbolic attributes and 51 numeric. The formed (by domain experts) rules were generally very short and the structure of the knowledge base was flat. The second base (`bud22190`) was generated by duplicating the first one five times with random modifications (because gaining access to large, real-world knowledge bases is very difficult). The third (`eval416`) and fourth (`eval1199`) knowledge bases regarded the topic of effectiveness evaluation of sales representatives and consisted of 416 and 1119 rules. More information about the structure and experimental evaluation of rules partitioning concerning these knowledge bases can be found in Simiński (2017).

The aim of the first experiment was to evaluate the data loading times from a relational database. The results of this experiment (shown in Table 1) were needed to confirm the usefulness of the proposed backward chaining inference algorithm implemented directly on the database and should be interpreted in the context of the second experiment.

Results presented in Table 1 clearly indicate that loading times from the database can be regarded as significant. It is especially visible for the `bud4438` and `bud22190` knowledge bases, because they have a lot of attributes whose definitions (and a list of possible values) are also stored beside the rule set. That is why the authors wanted to check if performing operations directly on the database server can be an alternative to having to wait for the data loading phase to finish in order to perform, for example, inference in the client's application. The memory usage for data structures which hold the rules seems to be reasonable. For 22190 rules the data structures occupy less than 5 MB of memory.

The goal of the second experiment was to compare two (previously described in this work) methods of backward chaining inference – a classical version implemented in the

Table 1. Knowledge base average loading times.

Knowledge base	Rules count	Loading time from database (s)	Memory occupation (KB)
eval416	416	2.092	96
eval1199	1199	5.185	285
bud4438	4438	29.238	1197
bud22190	22190	188.195	4646

Java programming language and one that operates directly on the database server. The results of this experiment are presented in Table 2.

It is obvious that inference realized on objects stored in the RAM of a client's computer will be faster than performing the same process directly in the database layer (even when using proper column indexes). But the results in Table 2 should be interpreted in the context of knowledge base loading times (see Table 1). Then one can observe an advantage of the stored procedure method (in the case of knowledge bases `bud4438` and `bud22190`) compared to running inference on the desktop application. In the case of the Java program, the user has to wait a time period of about 29 (in the case of `bud4438`) or 188 s (in the case of `bud22190`) for the database to load and additionally 0.2 or 2.5 s for the inference process to finish.

When performing inference directly on the database this time is reduced to a bit over 13 or 136 s, respectively. This means that the user will be able to perform inference at least one to two times (directly on the database), whereas the data would still be loading in the desktop application. Of course when the user plans to perform inference multiple times, it is still better to load the data into the desktop application, because it's a one-time operation only. As far as the `eval416` and `eval1119` knowledge bases are concerned, the direct database approach performs worse than the traditional one. This is caused by the structure of these knowledge bases. Although they store less rules, they form a hierarchical structure, and a chain of rules is activated and analysed instead of only a single rule like in the case of the `bud4438` and `bud22190` bases which have a flat structure.

The results from the second experiment can also have practical impacts – a 'smart' software dispatcher, which can choose the best inference scenario according to the current user's device properties – hardware configuration, internet availability, type of power supply – can be made. We are going to include this solution into the `KBExplorer` system and the `KBExpertLib` library in the future.

The aim of the last experiment was to determine the impact of the chosen storage engine (to handle CRUD operations on the data) on the rules' retrieval time (when considering the storage procedure approach). Three most commonly used MySQL storage engines were selected for further research: the *Memory*, *MyISAM* and *InnoDB* approaches (Bell, 2012). This analysis should be considered particularly useful when dealing with a large and hierarchical knowledge base which generates many subgoals in the backward chaining inference process. The results of this experiment are presented in Table 3.

The obtained results clearly indicate that the *Memory* storage engine is the fastest one in all tested cases (regardless of the used knowledge base). It is an obvious conclusion,

Table 2. Analysis of rules' retrieval time (in seconds).

Knowledge base	Data source	Min.	Max.	Mean	Median
eval416	Database	16.890	17.234	17.016	16.946
eval416	RAM	0.002	0.267	0.079	0.075
eval1119	Database	52.172	53.781	52.844	52.828
eval1119	RAM	0.002	1.377	0.383	0.367
bud4438	Database	13.063	14.969	13.961	13.899
bud4438	RAM	0.001	0.753	0.543	0.478
bud22190	Database	136.000	137.000	136.333	136.000
bud22190	RAM	0.001	2.462	1.567	1.342

Table 3. Analysis of rules' retrieval time based on the selected MySQL storage engine (in seconds).

Knowledge base	MySQL storage engine	Min.	Max.	Mean	Median
eval416	Memory	16.890	17.234	17.016	16.946
	MyISAM	16.943	17.701	17.226	17.172
	InnoDB	19.223	23.469	21.108	20.930
eval1119	Memory	52.172	53.781	52.844	52.828
	MyISAM	53.063	55.219	54.151	54.258
	InnoDB	60.000	64.000	61.667	61.500
bud4438	Memory	13.063	14.969	13.961	13.899
	MyISAM	14.500	16.201	15.511	15.572
	InnoDB	19.591	23.031	21.474	21.469
bud22190	Memory	136.000	137.000	136.333	136.000
	MyISAM	139.000	141.000	140.000	140.000
	InnoDB	172.000	177.000	174.500	174.500

because it uses only RAM as a main storage point, and it is the fastest memory type available. But what if the RAM resources were limited, so that the whole subgoals and facts tables could not be stored in it? This is a valid scenario, especially when taking into account the fact, that multiple users can carry out inference at the same time on the server. In such case, one has to consider using the remaining storage engines such as *MyISAM* or *InnoDB*. The most distinguishing feature of the *InnoDB* engine is its support for database transactions. *MyISAM* tables have generally a smaller footprint than *InnoDB* ones, but this storage engine does not support transactions and table-level locking operations may slow down write operations.

When comparing the results for the *MyISAM* and *Memory* storage engines, the first approach naturally performs worse, but the differences are not that noticeable. One could also observe that the bigger and more complex the knowledge base is, the more time it takes for the *MyISAM* approach to perform CRUD operations. But still the differences compared to the *Memory* engine are from 0.21 to 3.667 s (based on the mean column). So generally the *MyISAM* storage engine may be considered as a valid replacement for the *Memory* one if for some reason the latter cannot be used.

If data safety is the biggest concern, the *InnoDB* storage engine should be taken into account because it has commit, rollback and crash-recovery capabilities. Unfortunately, as the results presented in Table 3 indicate, it also performs worst in the backward chaining inference task. The differences between the previously analysed storage engines for the smaller knowledge bases (`eval416`, `eval1119` and `bud4438`) may be still considered as acceptable (as they are in the range of 4–9 s). However, results for the `bud22190` knowledge base indicate a noticeable difference. The *InnoDB* engine needs 36 more seconds on average to return the outcome of inference. Therefore, for larger and more complex knowledge bases the usage of the *InnoDB* storage engine may not be even applicable.

5. Conclusions

In this work, two approaches for inference implementation, which uses knowledge bases stored in the form of a relational database, were introduced. The first approach requires a priori loading of the knowledge base contents to the RAM of the client's computer. The inference process is performed later on using only data stored in the RAM. The second

approach involves implementing inference as a stored procedure, run in the environment of the database server.

Experiments were conducted on real-world knowledge bases with a relatively large number of rules. Experiments were prepared so that one could evaluate the pessimistic complexity of the inference algorithm. The results confirmed that the inference implemented in object-oriented data structures loaded into memory is effective. The times of inference in the worst-case scenario did not exceed 2.5 s. However, loading the contents of the knowledge base proved to be time-consuming. For small bases these times were about a couple of seconds, but for the largest one they reached over 3 min. Such a case is acceptable in systems where the knowledge base is reloaded rarely, and the waiting time (for data loading) is tolerable from the user's point of view. For applications where there is need for frequent reloading of the knowledge base, this solution is inconvenient and may be cumbersome for the user. This may be the case for knowledge bases which are frequently updated, for example, by programs that use specific algorithms to automatically generate rules.

The inference implemented in the form of a stored procedure runs significantly slower than the solution described previously, which is not a surprising result. However, when comparing the inference times and adding the time of loading data from the knowledge base, the solution using a stored procedure turns out to be faster (in specific conditions). This solution is especially convenient when the knowledge base is updated frequently. Unfortunately, the used storage engine has a significant impact on the rules' retrieval time and should be selected very carefully, taking into account the requirements and possibilities of the owned hardware. If free RAM amount is not an issue, the authors recommend to use the *Memory* storage engine as it is the fastest of the analysed approaches for every tested knowledge base. If RAM resources are limited (in a multi-user scenario, for example) the *MyISAM* storage engine may be considered as a valid replacement for the memory one.

Implementation of the inference in the form of a stored procedure is an interesting solution and will be permanently included in the described `KBExplorer` system and the`KBExpertLib` library. This will allow the programmer implementing a domain expert system to select the desired mode of inference. We consider the implementation of an 'intelligent' inference dispatcher, which will be able to select the inference method, according to the device resources (processor speed, free memory) as well as taking into account the speed of the used internet connection. This solution will be analysed as the next stage of research.

Notes

1. Decision attributes are attributes that are at least once included in a conclusion of any rule from *R*.
2. When the employed rules representation language allows to use the negation in literals, it is possible to check whether the $\neg g$ is a fact. If $\neg g$ is true, the goal g should be rejected.
3. For a detailed motivation why this storage engine was chosen have a look at the results of the last experiment.

Disclosure statement

No potential conflict of interest was reported by the authors.

Notes on contributors

Tomasz Xięski is a member of the Institute of Informatics at the University of Silesia. He graduated from the Faculty of Computer Science and Materials Science, University of Silesia in Katowice. Professional interests include methods, languages and tools for mobile and desktop programming. Scientific interests focus on methods of artificial intelligence, mainly on data mining, home automation and expert systems.

Roman Simiński is a member of the Institute of Informatics at the University of Silesia. He graduated from the Faculty of Automatic Control, Electronics and Computer Science, Silesian University of Technology in Gliwice. Professional interests include methods, languages and tools for programming and designing information systems. Scientific interests focus on methods of artificial intelligence, mainly on knowledge-based systems, inference processes and knowledge engineering.

References

Akerkar, R., & Sajja, P. (2010). *Knowledge-based systems*. Sudbury: Jones & Bartlett Publishers.

Bell, C. (2012). Expert mysql. Apress. Retrieved from https://github.com/Apress/exp-mysql.

Canadas, J., Palma, J., & Túnez, S. (2010). A tool for MDD of rule-based web applications based on OWL and SWRL. Knowledge Engineering and Software Engineering (KESE6), 1, Karlsruhe, Germany.

CLIPS. (2016). *CLIPS NASA home page*. Retrieved from http://www.siliconvalleyone.com/founder/clips/index.htm

DROOLS. (2016). *DROOLS home page*. Retrieved from https://www.drools.org

Duan, Y., Edwards, J. S., & Xu, M. (2005). Web-based expert systems: Benefits and challenges. *Information & Management*, *42*(6), 799–811.

eXpertise2Go. (2016). *eXpertise2Go home page*. Retrieved from http://expertise2go.com

EXSYS. (2016). *EXSYS home page*. Retrieved from http://www.exsys.com

Grove, R. (2000). Internet-based expert systems. *Expert systems*, *17*(3), 129–135.

Grzymala-Busse, J. W. (2012). *Managing uncertainty in expert systems* (Vol. 143). New York: Springer Science & Business Media.

Huntington, D. (2000). Web-based expert systems are on the way: Java-based web delivery. *PC AI*, *14*(6), 34–36.

JESS. (2016). *JESS information*. Retrieved from http://herzberg.ca.sandia.gov

Ligeza, A. (2006). *Logical foundations for rule-based systems* (Vol. 11). Berlin, Heidelberg: Springer.

Mathkour, H., Al-Turaiki, I., & Touir, A. (2009). The development of a bilingual fuzzy expert system shell. *Journal of King Saud University-Computer and Information Sciences*, *21*, 27–44.

Nowak-Brzezińska, A. (2016). Mining rule-based knowledge bases inspired by rough set theory. *Fundamenta Informaticae*, *148*(1–2), 35–50.

Nowak-Brzezińska, A., & Simiński, R. (2015). Goal-driven inference for web knowledge based system. ISAT'2015: International conference on Information Systems Architecture and Technology, Karpacz, Poland.

Przybyła-Kasperek, M., & Wakulicz-Deja, A. (2014). Global decision-making system with dynamically generated clusters. *Information Sciences*, *270*, 172–191.

Ruiz-Mezcua, B., Garcia-Crespo, A., Lopez-Cuadrado, J., & Gonzalez-Carrasco, I. (2011). An expert system development tool for non ai experts. *Expert Systems with Applications*, *38*(1), 597–609.

Sajja, P. S., & Akerkar, R. (2010). Knowledge-based systems for development. In P. S. Sajja, & R. Akerkar (Eds.), *Advanced knowledge based systems: Model, application & research* (pp. 1–11). Kolhapur,India: Technomathematics Research Foundation.

Simiński, R. (2016). The kbexpertlib software library for java–functionality properties and performance study. *Studia Informatica*, *37*(1), 125–134.

Simiński, R. (2017). The experimental evaluation of rules partitioning conception for knowledge base systems. In Information systems architecture and technology: Proceedings of 37th international

conference on Information Systems Architecture and Technology – ISAT 2016 – part I (pp. 79–89), Karpacz, Poland.

Simiński, R., & Manaj, M. (2015). Implementation of expert subsystem in the web application – selected practical issues. *Studia Informatica*, *36*(1), 131–143.

Simiński, R., & Nowak-Brzezińska, A. (2016). Kbexplorator and kbexpertlib as the tools for building medical decision support systems. In International conference on computational collective intelligence (pp. 494–503), Halkidiki, Greece.

Simiński, R., & Xięski, T. (2015). Physical knowledge base representation for web expert system shell. In International conference: Beyond databases, architectures and structures (pp. 558–570), Ustroń, Poland.

Simiński, R., & Xięski, T. (2017). Backward chaining inference as a database stored procedure – the experiments on real-world knowledge bases. IEEE International conference on innovations in intelligent systems and applications (inista) 2017 (pp. 253–258), Gdynia, Poland.

Xięski, T., & Simiński, R. (2017). A performance study of two inference algorithms for a distributed expert system shell. In International conference: Beyond databases, architectures and structures (pp. 512–526), Ustroń, Poland.

XpertRule. (2016). *Xpert Rule home page*. Retrieved from http://www.xpertrule.com