Design and Implementation of the CALO Query Manager

Jose-Luis Ambite[1], Vinay K. Chaudhri[2], Richard Fikes[3], Jessica Jenkins[3], Sunil Mishra[2], Maria Muslea[1], Tomas Uribe[2], Guizhen Yang[2]

1. USC Information Sciences Institute, Marina del Rey, CA 90292, USA
2. Artificial Intelligence Center, SRI International, Menlo Park, CA 94087, USA
3. Knowledge Systems Group, Artificial Intelligence Laboratory, Stanford University, Stanford, CA 94305, USA

Abstract. We report on our experience in developing a query-answering system that integrates multiple knowledge sources. The system is based on a novel architecture for combining knowledge sources in which the sources can produce new subgoals as well as ground facts in the search for answers to existing subgoals. The system uses a query planner that takes into account different query-processing capabilities of individual sources and augments them gracefully. A reusable ontology provides a mediated schema that serves as the basis for integration. We have evaluated the system on a suite of test queries in a realistic application to verify the practicality of our approach.

1. Introduction

The problem of integrating and querying information residing in heterogeneous knowledge sources has been a focus of intensive research during the past several years, [1], [2], [3]. In this paper, we report on our effort to build a real system for integrating heterogeneous knowledge sources with different query-answering capabilities. This system is an application of a hybrid reasoning architecture that is distinct from existing systems in two ways: (1) Knowledge sources integrated into our system may return logical formulas representing new subgoals as well as ground facts in response to queries. (2) The limited reasoning capabilities of individual knowledge sources are augmented by using a powerful query planner.

We are conducting this work in the context of CALO (Cognitive Assistant that Learns and Organizes), a multidisciplinary project funded by DARPA to create cognitive software systems that can reason, learn from experience, be told what to do, explain what they are doing, reflect on their experience, and respond robustly to surprises. (See http://www.ai.sri.com/project/CALO). The current project is targeted at developing personalized cognitive assistants (which we will refer to as CALOs) in an office environment where knowledge about email, schedules, people, contact information, and so on is distributed among multiple knowledge sources.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

A CALO must be able to access and reason with this distributed knowledge. We have encapsulated this functionality in a CALO module called Query Manager that serves as the unified access point within a CALO for answering queries. In an office, multiple CALOs will each support a single user. Each copy of CALO has its own copy of the Query Manager. The following three example queries typically arise in an office environment:

1. *Which meetings will have a conflict if the current meeting runs overtime by an hour?* Answering this query requires knowing the ending time of the current meeting, computing the new ending time, and determining which other meetings have been scheduled to pass over the new ending time. In a CALO, a user's schedules are stored in a personal information knowledge source called IRIS. The functionality of computing the new ending time is implemented in a reasoner, called Time Reasoner, that can evaluate simple functions and predicates on time points and intervals.

2. *Who was present in the meeting in conference room EJ228 at 10 a.m. this morning?* A person is considered by a CALO to be present in a meeting if it can recognize that person at the time of the meeting from images provided by the meeting room camera. This knowledge is stated as a rule in a knowledge source called Knowledge Machine (KM) [4]. Given this query, KM uses that rule to produce subgoals for retrieving meeting participants' information from the image analysis results provided by another independent knowledge source called Meeting Ontology Knowledge Base (MOKB). Therefore, in this example, one knowledge source produces new subgoals that are then evaluated by another knowledge source.

3. *List all the people who are mentioned in an article in which Joe is also mentioned?* This query requires first retrieving all the articles that mention Joe, and then for each of those articles, retrieving all the people mentioned in them. This requires selecting instances of a class based on a condition (articles that mention Joe), and then retrieving values of a property (people mentioned in an article) of each of those instances. This information is stored in IRIS. However, IRIS cannot process queries like this one that require successive retrievals from the same class based on different criteria. Therefore, Query Manager must take that limitation into account when developing query plans, and in this case form a plan that involves multiple calls to IRIS.

The examples above illustrate the challenges facing Query Manager in integrating knowledge sources and answering queries of different sorts. However, the intricacy of the internal workings of the system is completely transparent to its users — be they humans interacting with a CALO, or intelligent agents running on behalf of a CALO. Neither do users of Query Manager have to worry about how knowledge is expressed (e.g., as ground facts or axioms) or distributed in the system. Queries only need to be formulated using CALO Ontology, and Query Manager will determine which knowledge sources are required to produce the answers.

2. Query Manager Architecture

The Query Manager architecture is shown in Figure 1. It is based on an object-oriented modular architecture for hybrid reasoning, called the JTP architecture [5], which supports rapid development of reasoners and reasoning systems. The architecture considers each knowledge source to be a *reasoner*.

A CALO component (called a *client*) can obtain answers to a query and explanations for those answers by conducting a *query answering dialogue* with the Query Manager. A *query* includes a *query pattern* that is a conjunctive sentence expressed in KIF (Knowledge Interchange Format) [6] whose free variables [1] are considered to be ***query variables***. A *query answer* provides *bindings* of constants to some of these query variables such that the *query pattern instance* produced by applying the bindings to the query pattern and considering the remaining query variables in the query pattern to be existentially quantified is entailed by the knowledge sources in the Query Manager. Query Manager can produce multiple answers to a query and be recalled to provide additional answers when needed.

When queries arrive at the Query Manager, they first undergo syntactic validation and then are sent to the Asking Control Reasoner, which embodies two reasoning methods: iterative deepening and model elimination. The Asking Control Reasoner sends queries to the Asking Control Dispatcher, which calls two reasoners and a dispatcher in sequence: Rule Expansion Reasoner, Query Planner, and Assigned Goal Dispatcher. Each reasoner and dispatcher accepts as input a goal in the form of a query pattern and produces a *reasoning step iterator* as output. A reasoning step is a partial or complete proof of a goal, and a reasoning step iterator is a construct that when pulsed provides a reasoning step as output. The Rule Expansion Reasoner applies rules to its input goal, and provides reasoning steps that contain new subgoals that can be used to produce answers for the input goal. The Assigned Goal Dispatcher dispatches (groups of) subgoals to different reasoners per the query plan produced by Query Planner. Note that each knowledge source is integrated into the Query Manager by implementing an asking reasoner per the JTP reasoner interface specification (see Section 3 for more details) that encapsulates the knowledge source and the specific details of how it answers queries. We refer to such reasoners as *reasoning system adapters*.[2]

Ideally, there would not be a separate rule expansion reasoner in Query Manager. Any reasoner should be allowed to infer new subgoals that are not in the original query plan, and the query plan would then be revised to include the new subgoals. But such a control structure would require dynamic query planning, which is outside the scope of the current phase of the project. We worked around this problem by storing rules (i.e., Horn Clauses) in a separate reasoner (i.e., the Rule Expansion Reasoner) that is invoked before a query plan is generated. This solution is not completely general because it is not always possible to anticipate all such rules. Extending the architecture to support dynamic query planning is the subject of future work.

The design of Query Manager is based on a shared ontology called the CALO Ontology that serves as the *mediated schema* for all reasoners. All the reasoners are expected to implement (parts of) the CALO Ontology. Developing such an ontology is usually the biggest challenge in implementing systems that access heterogeneous knowledge sources. We bootstrapped this process by reusing a large shallow ontology that we had developed in a previous project [7]. Clearly, this ontology was inadequate for the office domain that was the focus of the current project and needed to be extended. Our strategy for extending the ontology was to set up a collaborative process between the developers of the knowledge sources and the knowledge engineers maintaining the ontology. This strategy is not always possible in a typical project that integrates heterogeneous sources, but we had the advantage that the developers of the knowledge sources were also members of the team. For instance, development of CALO Office Ontology was a result of collaboration between IRIS developers and the knowledge engineers, and CALO Meeting Ontology a result of collaboration between MOKB developers and the knowledge engineers. This approach, however, may not generalize to other systems or may be expensive to implement if there is no control on the schemas used by individual knowledge sources. In these cases, a lot of translation work must be done.

[1] A “free variable” in a KIF sentence is a variable that occurs in the sentence outside the scope of any enclosing quantifier.

[2] For convenience in exposition, from now on we will use the terms *reasoners* and *knowledge sources* interchangeably.

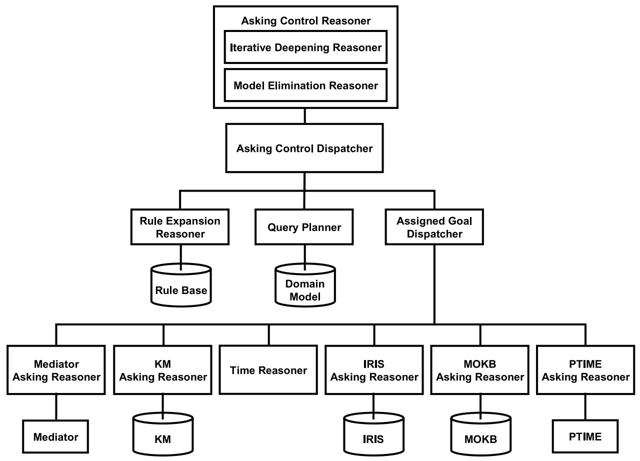

Figure 1. Query Manager Architecture

In addition to the knowledge sources that we have introduced and shown in Figure 1 (KM, MOKB, Time Reasoner, and IRIS), two other reasoners are currently integrated into Query Manager: Mediator and PTIME. Mediator is a data integration system, developed independently by ISI, which extracts useful information from the Web [8]. For example, it is able to access the Web sites of Office Depot and CDW to extract product and pricing information for laptops. In its current deployment in the CALO project, Mediator is interfaced with about two dozen Web sites in the office equipment domain. PTIME is a general-purpose constraint reasoning system. It is currently used for scheduling meetings: given the constraints of the meetings to be scheduled, and a user's preferences and calendar information, it can suggest alternative meeting times in the presence of schedule conflicts.

The current implementation of Query Manager is limited to only conjunctive queries. This is not an inherent limitation of the Query Manager architecture. The use of model elimination in Asking Control Reasoner permits acceptance of queries in full first-order logic. If we use a query language more expressive than conjunctive queries, then Query Planner will also need to be extended (a subject for future research). We also note that the use of iterative deepening provides termination guarantees in the presence of recursive rules, thanks to controlled search capabilities (this subject is not the main focus of this paper).

A big advantage of using the proposed architecture is that the query manager can be dynamically re-configured to use or not use a particular reasoner. For example, one can start up the query manager with only a subset of reasoners. Such dynamic flexibility is not available if one annotates individual relations with the reasoners that should be used for evaluating them.

3. Query Manager Components

3.1 Reasoner Interface

In the Query Manager architecture, the reasoner interface is needed to specify a protocol for integrating knowledge sources into Query Manager. This interface is based on an abstraction called a *reasoning step*. A reasoning step consists of the following elements:

1. A *claim*, which is a sentence that this reasoning step justifies
2. A set *of premises* on which the justification relies, each of which is a sentence
3. A set of *child reasoning steps* on whose claims this justification relies
4. A set of *variable bindings*
5. An *atomic justification* for the claim of this reasoning step given its premises and the claims of its children

An atomic justification can be either "Axiom" or an inference rule. Reasoning steps justified as Axiom do not have any premises or children, while an inference rule serves to infer the reasoning step's claim from its premises and the claims of its child reasoning steps.

Query Manager interacts with a reasoner through the interface by sending the reasoner a goal to prove. The reasoner responds by returning reasoning steps that are either *partial* or *complete proofs* of the goal. A complete proof is a reasoning step that has no premises and no descendant reasoning steps that have any premises. A *partial proof* is a reasoning step that has at least one unproved premise, either directly in itself or indirectly in a descendant reasoning step. Query Manager can derive a complete proof from a partial proof by proving these premises. Note that any reasoning step whose atomic justification is Axiom is a proof. To facilitate returning of multiple reasoning steps corresponding to different proofs of a goal, a reasoner actually returns a *reasoning step iterator*. Query Manager accesses the reasoning steps generated by a reasoner via the reasoning step iterator returned. The reasoner determines the order in which the reasoning steps are accessed via the iterator, and Query Manager determines when to retrieve reasoning steps from the iterator. This provides a framework for streaming answers and in which reasoners can generate answers as needed.

We first show an example of how this reasoning interface is used for interfacing Query Planner with Query Manager. Suppose Query Planner receives the following query as input: $(b \wedge d \wedge a \wedge c)$, where b, d, a, and c are positive, atomic literals, and decomposes this query into subgroups in two different ways. Query Planner will produce a reasoning step iterator. Each time Query Manaer invokes this iterator a reasoning step will be returned corresponding to one way of decomposing the given query. For example, the following illustrates the two reasoning steps generated:

```
Query Planner Reasoning Step 1:
Claim: (b ∧ d ∧ a ∧ c)
Premise 1: b
Premise 2: (d ∧ a ∧ c)
Justification: {Premise 1, Premise 2; AND introduction}

Query Planner Reasoning Step 2:
Claim: (b ∧ d ∧ a ∧ c)
Premise 1: (b ∧ d)
Premise 2: (a ∧ c)
Justification: {Premise 1, Premise 2; AND introduction}
```

Note that in order to complete the proof of the claim in each reasoning step, Query Manager needs to find proofs of the two premises in each reasoning step.

3.2 Query Planner

The key capabilities of Query Planner are *subgoal grouping* and *subgoal ordering*. As an example of subgoal grouping, consider a query (b ∧ d ∧ a ∧ c) that could be broken into two groups, one consisting of (b ∧ d) to be evaluated at one reasoner and the other (a ∧ c) at another reasoner. For an example of subgoal ordering, the groups in the above query must be evaluated in the order (a ∧ c) followed by (b ∧ d).

The subgoal grouping and ordering produced by Query Planner satisfies the following requirements:

- Satisfaction of binding pattern restrictions [9] of the predicates accepted by each reasoner.
- Modeling of predicate completeness and overlap, that is, whether a reasoner can provide the complete extension of a predicate or just a subset. Query Planner uses this information to minimize the number of reasoners accessed.
- Satisfaction of the query processing capabilities of different reasoners.
- Grouping of the subgoals in the query into maximal subgroups that can be handled by each reasoner, satisfying both the binding pattern restrictions and the reasoner query answering capabilities.

The information about binding constraints, predicate completeness, and query answering capabilities is encoded in what is called a *capability model* in Query Manager (the figure below shows a fragment of the capability model relevant for the query of Section 2). The query planning algorithm in Query Manager consists of the following phases:

1. **Satisfaction of binding constraints**. It constructs a dependency graph to satisfy the binding pattern restrictions of the predicates in the query. Since the dependency graph is a partial order, there may be many different groupings, depending on how we process the dependency graph. In our implementation, we output either a single group or all the groups.
2. **Completeness reasoning**. A predicate appearing in multiple reasoners may have one of two semantics:
   a. Incomplete (Open-world): Each reasoner may contribute different bindings to the predicate. Query Manager needs to query all the reasoners and union the results. For example, if two sources know about the predicate (Laptop-Computer X), then each reasoner may provide different laptops.
   b. Complete (Closed-world): Each reasoner has a replica of the predicate. Query Manager needs to query only one of the reasoners (preferably the most cost-effective one). For example, a predicate like ISI-Employee(X) may correspond to the directory listing at ISI and have all ISI employees; there is no need to go to any other sources.
3. **Selection assignments.** Each (in) equality predicate (=, >, <, ≥, ≤, ≠) is assigned to the first group where it can go according to the dependency graph in order to filter intermediate results as soon as possible.
4. **Enforcement of reasoner query answering capabilities**. Currently, the only query restriction is that IRIS cannot execute joins across classes. An analysis step further decomposes queries to IRIS into subgroups such that each group is a single-class query.

To clarify the behavior of Query Planner, consider the following example query: *Which meetings will have a conflict if the CALO Test meeting runs overtime by an hour?* This query can be formally expressed using KIF syntax as follows (note that names preceded by "?" represent variables, and all the relation names are drawn from the CALO ontology):

```
(and  (CurrentCaloUser ?user)
      (is-calendar-attendee ?user ?meeting)
      (Event-Entry ?meeting)
      (calendar-summary ?meeting "CALO Test")
       (has-end-date ?meeting ?end-date)
      (time-add ?end-date "PT60M" ?new-end-date)
      (Event-Entry ?affected-meeting)
      (time-inside-event  ?new-end-date  ?affected-meeting)
      (is-calendar-attendee ?user ?affected-meeting))
```

The fragment of the capability model for the predicates is depicted as follows (note that "f" means an argument can be free whereas "b" denotes that an argument must be bound):

```
(CurrentCaloUser  Person:f)@[KM, IRIS]
(is-calendar-attendee  Event:f  Person:f)@[KM, IRIS]
(Event-Entry  Event:f)@[KM, IRIS]
```

```
(calendar-summary  Event:f  String:f)
(has-end-date  Event:f  dateTime:f)@[KM, IRIS]
(time-add  dateTime:b  timeDuration:b  dateTime:f)@[Time
Reasoner]
(has-start-date  Event:f  dateTime:f)@[KM, IRIS]
(time-lt  dateTime:b  dateTime:b)@[Time Reasoner/IRIS]
(time-gt  dateTime:b  dateTime:b)@[Time Reasoner/IRIS]
```

By default, predicates are deemed incomplete: all relevant reasoners of a predicate must be queried to get all the answers. Given the specification above, Query Planner first orders the predicates according to their binding pattern restrictions, and annotates the given query with the reasoners that can handle each of the predicates. The annotated query is as follows:

```
(and   (CurrentCaloUser ?user)@[KM, IRIS]
       (is-calendar-attendee ?user ?meeting)@[KM, IRIS]
       (Event-Entry ?meeting)@[KM, IRIS]
       (calendar-summary ?meeting “CALO Test”)
                                   @[KM, IRIS]
       (has-end-date ?meeting ?end-date)@[KM, IRIS]
       (time-add ?end-date “PT60M” ?new-end-date)
                                   @[Time Reasoner]
       (Event-Entry ?affected-meeting)@[KM, IRIS]
       (has-start-date ?affected-meeting ?e-start)@[KM, IRIS]
       (has-end-date ?affected-meeting ?e-end)@[KM, IRIS]
       (time-lt ?e-start ?new-end-date)
                                   @[Time Reasoner/ IRIS]
       (time-gt ?e-end ?new-end-date)@[Time Reasoner/IRIS]
       (is-calendar-attendee ?user ?affected-meeting)
                                   @[KM, IRIS])
```

The annotated query represents a conjunction of disjunctive queries. For instance, the statement (CurrentCaloUser ?user)@[KM, IRIS] is equivalent to the following disjunctive query:

```
(or    (CurrentCaloUser ?user)@KM
       (CurrentCaloUser ?user)@IRIS)
```

Query Planner searches the annotated query for maximal groups of predicates that can be executed at the same reasoner while satisfying the binding constraints. During the search, the planner ensures that the predicates sharing object IDs belong to the same reasoner. In general, object IDs are internal to each reasoner and thus cannot be “joined” across different reasoners. For example, consider the following annotated query:

```
(Event-Entry ?meeting)@[KM,IRIS]
(is-calendar-attendee ?user ?meeting)@[KM,IRIS]
```

Here are four ways of combining these predicates:

(Event-Entry ?meeting) @[KM] (is-calendar-attendee ?user ?meeting) @[KM]	(Event-Entry ?meeting) @[KM] (is-calendar-attendee ?user ?meeting) @[IRIS]
(Event-Entry ?meeting) @[IRIS] (is-calendar-attendee ?user ?meetin) @[KM]	(Event-Entry ?meeting)@[IRIS] (is-calendar-attendee ?user ?meeting) @[IRIS]

Observe that variable ?meeting will be bound to object IDs. Therefore, of these plan fragments, only those (two) plans where variable ?meeting is bound by the same reasoner in both literals are valid; the others would simply not produce any answers. This heuristic has a significant impact as the size of the query (in terms of the number of predicates) increases.[3] So after taking into account the assignment of predicates to reasoners and the dependency graph for the query, the query planning algorithm produces the following two query plans. One plan uses only KM to retrieve calendar information:

```
(and  (and   (CurrentCaloUser ?user)
             (is-calendar-attendee ?user ?meeting)
             (Event-Entry ?meeting)
             (calendar-summary ?meeting “CALO Test”)
             (has-end-date ?meeting ?end-date)
             (Event-Entry ?affected-meeting)
             (has-start-date ?affected-meeting ?e-start)
             (has-end-date ?affected-meeting ?e-end)
             (is-calendar-attendee ?user ?affected-meeting))
                                                @KM

      (time-add ?end-date “PT60M” ?new-end-date)
                                   @Time Reasoner

      (and   (time-lt ?e-start ?new-end-date)
             (time-gt ?e-end ?new-end-date))
                                   @Time Reasoner)
```

and the other uses only IRIS for calendar information:

```
(and  (and   (CurrentCaloUser ?user)
             (is-calendar-attendee ?user ?meeting)
             (Event-Entry ?meeting)
             (calendar-summary ?meeting “CALO Test”)
             (has-end-date ?meeting ?end-date))@IRIS

      (time-add ?end-date “PT60M” ?new-end-date)
                                   @Time Reasoner

      (and   (Event-Entry ?affected-meeting)
             (has-start-date ?affected-meeting ?e-start)
             (has-end-date ?affected-meeting ?e-end)
             (time-lt ?e-start ?new-end-date)
             (time-gt ?e-end ?new-end-date)
             (is-calendar-attendee ?user ?affected-meeting))
                                   @IRIS
```

Observe that both query plans have three groups. Since only Time Reasoner can handle the predicate time-add, it must be evaluated separately. All these groups in each plan must be evaluated in succession so as to satisfy the binding constraints: evaluation of the first group will provide bindings for ?end-date needed by the predicate time-add in the second group, which will in turn provide bindings for the variable ?new-end-date needed by the time comparison operators in the third group.

[3] At this point the query planner is considering only logical optimizations like removing subqueries with nonmatching object IDs. In the future we also plan to include cost-based optimizations based on the expected sizes of different predicates and subqueries.

Query	# Literals	# Expansion	# Plans	# Sources	# Groups	QP Time (ms)
TF1	10	0	1	1	1	42.1
TF2	13	0	2	2 (15)[a]	2	52.1
TF3	5	0	2	2 (17)[b]	2	20.0
TF4	9	0	1	1	1	38.1
TS1	15	0	72	5	585	77.1
TS2	20	0	2	3	4	85.1
TS3	22	0	1	2	3	98.0
TD1	1	3	38	4	150	85.1
TD2	10	6	72	4	314	202.1
TD3	12	1	18	4	47	69.1

a Query Manager called Mediator that in turn called 15 Web information extraction agents, one per Web site.

b In addition to the 15 Web information extraction agents, two other sources were used for retrieving reviews by Mediator.

Table 1. Evaluation of the Query Manager

Finally, Query Planner must still take into account the fact that IRIS can only answer queries about instances of a single class. This, however, does not result in a multiplication of the number of plans. Instead, a further rewrite of the second query plan above suffices. Its final plan is as follows:

```
(and    (and(   CurrentCaloUser ?user)
                (is-calendar-attendee ?user ?meeting))
                                                  @IRIS
        (and    (Event-Entry ?meeting)
                (calendar-summary ?meeting "CALO Test")
                (has-end-date ?meeting ?end-date))
                                                  @IRIS

        (time-add ?end-date "PT60M" ?new-end-date)
                                          @Time Reasoner

        (and    (Event-Entry ?affected-meeting)
                (has-start-date ?affected-meeting ?e-start)
                (has-end-date ?affected-meeting ?e-end)
                (time-lt ?e-start ?new-end-date)
                (time-gt ?e-end ?new-end-date))@IRIS

        (is-calendar-attendee ?user ?affected-meeting)@IRIS)
```

3.3 Reasoning System Adapters

While implementing the reasoning system adapters we had to deal with the problems of ontology mismatches and impedance mismatches. The ontology mismatch problem arises when the ontology used by a source is different from the one used by the query manager. In such cases the reasoning system adapter implements the translation between the two ontologies. The impedance mismatch problem arises when a source does not support logical query interface. In such cases, the reasoning system adapter implements a mapping between the logical specification of a query to the native API of a source. An example native API for a source is based on Java functions. The adapter then must map logical queries into Java functions, and the results of the function into variable bindings in the query.

4. Evaluation

We evaluated the performance of Query Manager in the context of an application of CALO in the office domain. Specifically, we chose a set of queries from the following categories and studied various aspects of answering these queries in Query Manager (for simplicity, we have omitted the KIF representation of these queries and stated them in English without all the details):

1. Task Fulfillment: Purchasing a piece of office equipment such as a laptop, printer, or scanner.

   (TF1) List all vendors from which purchase of flatbed scanners can be made.
   (TF2) Get real-time quotes, including price, vendor information, horizontal and vertical resolution, for flatbed scanners with price less than $200.
   (TF3) Show models of flatbed scanners having the greatest number of positive product reviews.
   (TF4) Show purchase request forms that were filled in for the latest purchase of flatbed scanners.

2. Task Setup: Setting up an office task such as arranging a meeting.

   (TS1) List all schedule conflicts if the meeting "CALO Test" goes overtime by an hour.
   (TS2) Show the best times to schedule a one-hour meeting between 9 a.m., May 10, 2005 and 3 p.m., May 12, 2005.
   (TS3) List all existing meetings that will be canceled if the current meeting is scheduled.

3. Task Discussion: Observing a meeting in which participants are discussing a project schedule.

(TD1)	List all participants of a particular meeting.
(TD2)	Show all meeting artifacts that were in focus in a particular meeting.
(TD3)	Show the person who answered the question raised by Joe during a particular meeting.

The current query manager was used by knowledge engineers for evaluating the overall performance of the system, and has not yet been put in front of the end-users of CALO. In its use by the knowledge engineers, there were no problems in formulating the queries and getting the answers. We did face some difficulty in initially explaining the functionality and services offered by the system. For example, one interesting feature of our approach is that one can decide at runtime which reasoners should be used in answering the queries. It was not until the time that the users wanted this functionality that this became a compelling feature. We also anticipate that as the overall capability of CALO evolves there will be increasing requirements for reasoning that can be only satisfied using the proposed architecture. We also plan to investigate the questions that are of interest to end –users and provide a user-interface for formulating those questions.

In the current paper, we have considered only a sampling of questions that the query manager can answer. The set of overall questions answered by it is a function of the questions that the reasoners integrated into it can answer. It is capable of taking any conjunctive query, and decomposing and mapping it into individual sources, and presenting the answers. The current system is limited to only conjunctive queries, and thus, will have trouble with queries that have disjunctions or negations in them. It will also have trouble with queries that require significant re-planning during the query evaluation process.

For the queries above, we consider the following metrics (results are shown in columns of Table 2):

Number of literals in the original KIF encoding of the query (Column 21)

- Number of rule expansions involved in answering the query (Column 3)
- Number of conjunctive query plans generated by Query Planner (Column 4)
- Number of knowledge sources involved in answering the query (Column 5)
- Number of groups in query plans generated for this query (Column 6)
- Total running time taken by Query Planner in answering the question (Column 71)

We conducted our test using a publicly released version of CALO v2.0. Our test machine was an IBM T42 laptop with 1.5 GB of main memory, 1.7 GHz Intel Centrino CPU, and running on Windows XP Professional. Our system was implemented in Java and we used JRE v1.4.2 for our test. To get an idea of the performance of Query Manager in real running environments, we used the measurement of elapsed time. We executed each query multiple times to gather statistically significant results.

First, observe that the running time taken by Query Planner alone in answering these questions ranges from 40 to 200 ms. This level of performance is satisfactory in real running environments since many of these queries take much longer than 200 ms to execute. For instance, TF2 normally takes a couple of minutes because in answering this query Mediator needs to fetch HTML documents from multiple Web sites and then perform data extraction operations on these documents. Therefore, the overhead introduced by Query Planner is not significant.

In contrast to questions in the Task Fulfillment and Task Setup categories, answering questions in the Task Discussion category requires various degrees of rule expansion because of the limited reasoning capability of MOKB, which mainly serves as a persistent triple store and does not support reasoning.

Clearly, the number of query plans increases with the number of rule expansions involved, because each different rule expansion results in a different query plan. This correlation can be validated in questions TD1, TD2, and TD3. The number of knowledge sources involved also has an impact on the number of query plans generated, as can be observed in question TD3. The large number of query plans generated for questions TS1, TD1, TD2, and TD3 is due to a limitation in our current implementation: our domain model is not expressive enough for pruning all query plans that are semantically equivalent. In particular, questions TS1, TD1, TD2, and TD3 used a large number of time comparison operators that are declared to be implemented by several reasoners in their domain models. Even though the "Completeness" reasoning discussed in Section 3.2 can be utilized to address this problem, predicates with operator-like semantics need to be optimized with additional constraints: Ideally, operators should be pushed down as close to the appropriate knowledge sources as possible; randomly selecting a knowledge source for this operator in general does not have good performance guarantees. Currently, we are working on developing a new domain model (in combination with a statistics model) for Query Manager to address this problem.

In Column 2 of Table 1, the number of literals listed is for literals in the original KIF encoding. In most cases, rule expansion will add literals to a query. If no rule expansion is involved, the numbers in Column 2 are good indications of the size of the query submitted to Query Planner. In these cases (questions TF1-4 and TS1-3), we can observe that the running time of Query Planner is proportional to the size of its input. The seemingly long running times for questions TD1 and TD2 need some further explanation. In the case of question TD1, rule expansion results in two final conjunctive queries, one with 15 literals and the other with 3 literals. Therefore, the total input size to Query Planner is 18 and we can see that the total running time on these 18 literals (85.1 ms) is comparable to that of question TS2 (20 literals, 85.1 ms). In the case of TD2, rule expansion eventually results in two conjunctive queries, each with 23 literals. We can observe that in this case the total running time roughly doubles that of question TS3 (22 literals, 98.0 ms).

5. Related Work

The CALO Query Manager is conceptually a hybrid reasoning system: the reasoning systems may return subgoals. This is the most distinctive aspect of this system in relation to the most previous work on information integration. Approaches to incorporating external reasoners into first-order deductive systems include procedural attachment [10] and theory resolution [11]. The resolution used in the QM might best be viewed as implementing the theory resolution approach, in a model-elimination setting. However, these hybrid reasoning frameworks do not specify how the literals should be grouped together, in what order they should be handled, or which reasoner to try first when there is a choice. More generally, our work addresses query optimization problems (such as grouping and ordering of literals) that are likely to appear in any hybrid reasoning system using heterogeneous sources.

6. Conclusion and Future Work

We have presented a system for integrating heterogeneous knowledge sources with the help of a reusable ontology, a query planner, and a hybrid reasoning architecture. The main advantages of the approach include semantically well-defined interfaces, efficient execution of queries, and the ability to leverage reasoning for answering queries. We have also presented empirical results of evaluating our system that demonstrated its efficiency.

This work can be extended in several ways: (1) query planning needs to be made dynamic to handle partial proofs; (2) query planning needs to take in statistics to achieve better performance; (3) our domain model needs to represent a richer set of source capabilities; (4) Query Manager should be able to work in a multi-CALO environment in which slight divergent ontologies may exist in individual CALOs.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, or the Department of Interior-National Business Center (DOI-NBC).

References

1. Noy, N. and A. Halevy, eds. *ACM SIGMOD Record*. . Vol. 33. 2004.
2. Noy, N.F., A. Doan, and A.Y. Halevy, eds. *AAAI Magazine Special Issue on Semantic Data Integration*. Vol. 26. Spring, 2005.
3. Ouksel, A. and A. Sheth, eds. *ACM SIGMOD Record Special Issue on Semantic Interoperability in Global Information Systems*. Vol. 28. March 1999.
4. Clark, P. and B. Porter. *KM -- The Knowledge Machine: Users Manual*. 1999. The system code and documentation are available at http://www.cs.utexas.edu/users/mfkb/km.html.
5. Fikes, R., J. Jenkins, and G. Frank. *JTP: A System Architecture and Component Library for Hybrid Reasoning,* in *Proc. Seventh World Multiconference on Systemics, Cybernetics, and Informatics.* 2003. Orlando, FL, USA.
6. Genesereth, M.R. and R.E. Fikes. *Knowledge Interchange Format, Version 3.0 Reference Manual.* 1992 (Logic-92-1).
7. Barker, K., B. Porter, and P. Clark. *A Library of Generic Concepts for Composing Knowledge Bases*, in *Proc. 1st Int Conf on Knowledge Capture (K-Cap'01)*. 2001. p. 14--21
8. Thakkar, S., J.L. Ambite, and C.A. Knoblock. *A Data Integration Approach to Automatically Composing and Optimizing Web Services*. in *ICAPS Workshop on Planning and Scheduling for Web and Grid Services*. 2004.
9. Ullman, J. *Principles of Database and Knowledge Base Systems, Volume 2.* 1989.
10. Myers, K. *Hybrid Reasoning Using Universal Attachment.* Artificial Intelligence, 1994. **67**: p. 329-375.
11. Stickel, M. *Automated Deduction by Theory Resolution.* Journal of Automated Reasoning, 1985. **4**: p. 333-355.