CodeQuest: Scalable Source Code Queries with Datalog

Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor

Programming Tools Group,
Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{Elnar.Hajiyev, Mathieu.Verbaere, Oege.de.Moor}@comlab.ox.ac.uk,
http://progtools.comlab.ox.ac.uk/projects/codequest/

Abstract. Source code querying tools allow programmers to explore relations between different parts of the code base. This paper describes such a tool, named *CodeQuest*. It combines two previous proposals, namely the use of logic programming and database systems.

As the query language we use *safe Datalog*, which was originally introduced in the theory of databases. That provides just the right level of expressiveness; in particular recursion is indispensable for source code queries. Safe Datalog is like Prolog, but all queries are guaranteed to terminate, and there is no need for extra-logical annotations.

Our implementation of Datalog maps queries to a relational database system. We are thus able to capitalise on the query optimiser provided by such a system. For recursive queries we implement our own optimisations in the translation from Datalog to SQL. Experiments confirm that this strategy yields an efficient, scalable code querying system.

1 Introduction

Understanding source code is vital to many tasks in software engineering. Source code querying tools are designed to help such understanding, by allowing programmers to explore relations that exist between different parts of the code base. Modern development environments therefore provide querying facilities, but these are usually fixed: one cannot define new relationships that are particular to the project in hand.

It can be very useful, however, to define such project-specific queries, for instance to enforce coding style rules (*e.g.* naming conventions), to check correct usage of an API (*e.g.* no call to a GUI method from an enterprise bean), or to ensure framework-specific rules (*e.g.* in a compiler, every non-abstract AST class must override the *visitChildren* method). Apart from such checking tasks, we might want new ways of navigating beyond the fixed set of relations provided in a development environment. When cleaning up a piece of legacy software, it is for example useful to know what methods are never called (directly or indirectly) from the *main* method. A good querying tool allows the programmer to define all these tasks via simple, concise queries. Note that none of these examples is easily

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 2–27, 2006.

implemented with today's dominant code querying tool, namely *grep*. Built-in querying and navigating facilities of Eclipse, widely used by the IDE users, are limited to a fixed number of certain queries.

The research community has long recognised the need for flexible code queries, and many solutions have been proposed. We shall discuss this previous work in detail in Sect. 6. For now it suffices to say that two crucial ideas have emerged from that earlier research: a logical query language like Prolog to formulate queries, and a relational database to store information about the program.

All these earlier attempts, however, fall short on at least one of three counts: the system is not scalable to industrial-size projects, or the query language is not sufficiently expressive, or the queries require complex annotations to guarantee efficiency. Scalability is typically not achieved because no query optimisation is used, and/or all data is held in main memory. Expressiveness requires recursive queries, to inspect the graph structures (the type hierarchy and the call graph, for example) that are typically found in code queries. Yet the use of recursion in SQL and XQuery is cumbersome, and in Prolog recursion over graphs often leads to non-termination. In Prolog that problem may be solved via tabling plus mode annotations, but such annotations require considerable expertise to get right.

1.1 Contributions

This paper solves all these deficiencies, and it presents a code querying tool that is scalable, expressive and purely declarative. We achieve this through a synthesis of the best ideas of the previous work on code querying. To wit, our contributions are these:

- The identification of *safe Datalog* (a query language originating in database theory) as a suitable source code query language, in the sweet spot between expressiveness and efficient implementation.
- The implementation of Datalog via an optimising compiler to SQL, which is in turn implemented on a relational database system. Our compiler performs a specialised version of the well-known 'magic sets' transformation, which we call 'closure fusion'.
- A method of incrementally updating the database relations when a compilation unit is changed.
- A comprehensive set of experiments, with two different commercial database systems (Microsoft SQL Server and IBM DB2) as a backend for our query compiler, to show the scalability of our approach. We also demonstrate that for this application, a special implementation of recursion outperforms the built-in recursion provided by these database systems.
- Detailed comparison with other state-of-the-art code querying tools, in particular JQuery (an Eclipse plugin tailored for code queries) [2, 24, 34] and XSB (a general optimising compiler for tabled Prolog [3,39]), demonstrating that on small projects our approach is competitive, and on large projects superior.

1.2 Paper Organisation

The paper is organised as follows. First we provide a brief introduction to Datalog; we also present its semantics with an emphasis on the concepts that are important to the implementation of *CodeQuest* (Sect. 2). That implementation is presented in Sect. 3. It is also here that we discuss a number of alternative implementations of recursion, via built-in facilities of the underlying database system, and via a procedural implementation of our own. Next, in Sect. 4, we turn to the tricky problem of incrementally updating the database when a change is made to the source program. Clearly this is crucial to the use of *CodeQuest* in the context of refactoring, where queries are interspersed with frequent changes. The heart of the paper is Sect. 5: there we demonstrate, through careful experiments with a wide variety of queries, that our implementation method yields a truly scalable system. The experiments are conducted with two major database systems to factor out any implementation accidents in our measurements. We also assess the efficiency of incrementally rebuilding the database with a series of refactoring queries. In Sect. 6, we provide a comprehensive account of all the previous work on code queries that has inspired the construction of *CodeQuest*. Finally, we conclude in Sect. 7.

2 Datalog

Datalog is a query language originally put forward in the theory of databases [20]. Syntacticly it is a subset of a logic language Prolog, but has a different evaluation strategy. It also poses certain stratification restrictions on the use of negation and recursion. As a result, in contrast to Prolog, Datalog requires no extra-logical annotations in order to guarantee termination of the queries. At the same time it has the right level of expressiveness for the type of applications discussed above.

Datalog's basic premise is that data is arranged according to relations. For example, the relation *hasName* records names of program elements. Variables are used to find unknown elements; in our syntax, variable names start with a capital letter. So one might use the *hasName* relation as follows:

$$hasName(L, \text{`List'})$$

is a query to find all program elements with the name *List*; the variable L will be instantiated to all program elements that have that name.

Unary relations are used to single out elements of a particular type. So for example, one might write

$$\begin{aligned} &method(M), hasName(M, \text{`add'}),\\ &interface(L), hasName(L, \text{`List'}),\\ &hasChild(L, M) \end{aligned}$$

Here the comma stands for logical 'and'. This query checks that the *List* interface contains a method named *add*. It also illustrates an important issue: a method is a program element, with various attributes, and the name of the method is just

one of those attributes. It is incorrect to write *hasChild*('List', 'add'), because names do not uniquely identify program elements. At present *CodeQuest* does not have a type system, so this incorrect predicate would just evaluate to 'false'.

Above, we have used primitive relations that are built into our version of Datalog only. One can define relations of one's own, for instance to define the notion of subtypes (semi-colon (;) stands for logical 'or', and (:−) for reverse implication):

$$\mathit{hasSubtype}(T, S) :- \mathit{extends}(S, T) \ ; \ \mathit{implements}(S, T).$$

This says that T has a (direct) subtype S when S extends T or S implements T. Of course *CodeQuest* provides many such derived predicates by default, including *hasSubtype*. Unlike primitives such as *extends* or *implements*, these derived predicates are not stored relations, instead they are deduced from the primitives. A full list of all primitive and derived predicates provided in *CodeQuest* can be found on the project web page [5].

In summary, basic Datalog is just a logic programming language, quite similar to Prolog, but without data structures such as lists. The arguments of relations are program elements (typically nodes in the abstract syntax tree) and names. Like other logic programming languages, Datalog is very compact compared to query languages in the SQL tradition. Such conciseness is very important in a code querying tool, as verbosity would defeat interactive use.

Recursion. Code queries naturally need to express properties of tree structures, such as the inheritance hierarchy and the abstract syntax tree. They also need to express properties of graphs, such as the call graph, which may be cyclic. For these reasons, it is important that the query language supports recursion. To illustrate, here is a definition of direct or indirect subtypes:

$$\begin{aligned} \mathit{hasSubtypePlus}(T, S) :- \ & \mathit{hasSubtype}(T, S) \ ; \\ & \mathit{hasSubtype}(T, \mathit{MID}), \mathit{hasSubtypePlus}(\mathit{MID}, S). \end{aligned}$$

Now seasoned logic programmers will recognise that such definitions pose a potential problem: in Prolog we have to be very careful about variable bindings and possible cycles to guarantee termination. For efficiency, we also need to worry about overlapping recursive calls. For example, the above would not be an adequate program in XSB, a state-of-the-art version of Prolog [3, 39]. Instead, we would have to distinguish between whether T is known or S is known at the time of query evaluation. Furthermore, we would have to annotate the predicate to indicate that its evaluation must be *tabled* to avoid inefficiency due to overlapping recursive calls. JQuery [2, 24, 34], the code querying system that is the main inspiration for *CodeQuest*, similarly requires the developer to think about whether T or S is known during query evaluation.

CodeQuest foregoes all such extra-logical annotations: one simple definition of a recursive relation suffices. We believe this is an essential property of a code querying language, as the queries should be really easy to write, and not require any understanding of the evaluation mechanism. Termination is never an issue, as all recursions in *CodeQuest* terminate, due to certain restrictions explained below.

Semantics. Datalog relations that are defined with recursive rules have a least-fixpoint semantics: they denote the smallest relation that satisfies the given implication. To illustrate, the above clause for *hasSubtypePlus* defines it to be the least relation X that satisfies

$$X \supseteq \mathit{hasSubtype} \cup (\mathit{hasSubtype} \circ X)$$

where $(\circ)$ stands for sequential relational composition (*i.e.* $(a, c) \in (R \circ S)$ iff $\exists b : (a, b) \in R \wedge (b, c) \in S$). The existence of such a smallest solution X is guaranteed in our version of Datalog because we do not allow the use of negation in a recursive cycle. Formally, that class of Datalog programs is said to be *stratified*; interested readers may wish to consult [9] for a comprehensive survey.

It follows that we can reason about relations in Datalog using the relational calculus and the Knaster-Tarski fixpoint theorem [29, 11, 18]: all our recursions correspond to monotonic mappings between relations (f is monotonic if $X \subseteq Y$ implies $f(X) \subseteq f(Y)$). For ease of reference, we quote that theorem here:

Theorem 1 (Knaster-Tarski). *Let f be a monotonic function on (tuples of) relations. Then there exists a relation R such that $R = f(R)$ and for all relations X we have*

$$f(X) \subseteq X \quad \textit{implies} \quad R \subseteq X$$

The relation R is said to be the least fixpoint *of f.*

In particular, the theorem implies that we can compute least fixpoints by iterating from the empty relation: to find the R in the theorem, we compute $\emptyset, f(\emptyset), f(f(\emptyset)), \ldots$ until nothing changes. Because our relations range over a finite universe (all program elements), and we insist that all variables in the left-hand side of a clause are used at least once positively (that is not under a negation) on the right-hand side, such convergence is guaranteed to occur in a finite number of steps. Together with the restriction to stratified programs, this means we handle the so-called *safe* Datalog programs. *CodeQuest* does not place any further restrictions on the use of recursion in Datalog.

Closure fusion. Another very simple consequence of Knaster-Tarski, which we have found to be effective as an optimisation in *CodeQuest*, is *closure fusion*. The reflexive transitive closure R^* of a relation R is defined to be the least fixpoint of

$$X \mapsto \mathit{id} \cup (R \circ X)$$

where *id* is the identity relation.

Theorem 2 (closure fusion). *The relation $R^* \circ S$ is the least fixpoint of*

$$X \mapsto S \cup (R \circ X)$$

Furthermore, $S \circ R^*$ is the least fixpoint of

$$X \mapsto S \cup (X \circ R)$$

In words, this says that instead of first computing R^* (via exhaustive iteration) and then composing with S, we can start the iteration with S. As we shall see, this saves a lot of work during query evaluation. Due to the strictly declarative nature of Datalog, we can do the optimisation automatically, while compiling the use of recursive queries.

To illustrate closure fusion, suppose that we wish to find all types in a project that are subtypes of the *List* interface:

$$listImpl(X) :- type(L), hasName(L, \text{`List'}), hasSubtypePlus(L, X).$$

A naïve evaluation of this query by fixpoint iteration would compute the full *hasSubtypePlus* relation. That is not necessary, however. Applying the second form of the above theorem with $R = hasSubtype^*$ and

$$S(L, X) :- type(L), hasName(L, \text{`List'}), hasSubtype(L, X).$$

we obtain the result

$$\begin{aligned} listImpl(X) &:- hasSubtypePlus'(L, X). \\ hasSubtypePlus'(L, X) &:- type(L), hasName(L, \text{`List'}), hasSubtype(L, X). \\ hasSubtypePlus'(L, X) &:- hasSubtypePlus'(L, MID), hasSubtype(MID, X). \end{aligned}$$

Readers who are familiar with the deductive database literature will recognise this as a special case of the so-called *magic sets* transformation [12]. In the very specialised context of *CodeQuest*, it appears closure fusion on its own is sufficient to achieve good performance.

3 *CodeQuest* Implementation

CodeQuest consists of two parts: an implementation of Datalog on top of a relational database management system (RDBMS), and an Eclipse [1] plugin for querying Java code via that Datalog implementation. We describe these two components separately.

3.1 Datalog Implementation

Our implementation of Datalog divides relations into those that are stored in the database on disk, and those that are computed via queries. When we are given a particular query, the relevant rules are compiled into equivalent SQL. The basics of such a translation are well understood [30, 27]; somewhat surprisingly, these works do not include careful performance experiments. Details of the translation that we employ can be found in [23].

The most interesting issue is the implementation of recursion. As noted in the previous section, we restrict ourselves to *safe* Datalog programs, and that implies we can compute solutions to recursive equations by exhaustive iteration.

Modern database systems allow the direct expression of recursive SQL queries via so-called Common Table Expressions (CTEs), as described in the SQL-99

standard. This is one of the implementations available in *CodeQuest*. A major disadvantage, however, is that most database systems impose the additional restriction that only bag (multiset) operations may be used inside the recursion: one cannot employ set union, for example. That implies the semantics of CTEs do not quite coincide with our intended semantics of Datalog. In particular, while in our semantics, all recursions define a finite relation, the corresponding CTE may fail to terminate because there are an infinite number of duplicates in the resulting relation. We shall see a concrete example of that phenomenon later on, when we write queries over the call graph of a program.

It follows that it is desirable to provide an alternative implementation of recursion. Suppose we have a recursive rule of the form:

$$result :- f(result).$$

where $f(R)$ is some combination of R with other relations. We can then find a least fixpoint with the following naive algorithm:

$result = \emptyset$;
do {
  $oldresult = result$;
  $result = f(oldresult)$;
}
while ($result \neq oldresult$)

All modern database systems allow us to express this kind of computation in a procedural scripting variant of SQL. Furthermore such scripts get directly executed on the database server; they are sometimes called *stored procedures*. We shall refer to this implementation as *Proc1* in what follows. We stress once more that because of our restriction to *safe* Datalog, *Proc1* always terminates, in contrast to the CTE implementation. In our experiments, *Proc1* is also sometimes faster than CTEs.

The above method of computing least fixpoints is of course grossly inefficient. If we know that $f(R)$ distributes over arbitrary unions of relations, significant improvements are possible. A sufficient requirement for such distribution is that $f(R)$ uses R only once in each disjunct. Such recursions are called linear, and in our experience most recursions in code queries satisfy that criterion. The following semi-naïve algorithm uses a worklist to improve performance when f distributes over arbitrary unions:

$result = f(\emptyset)$;
$todo = result$;
while ($todo \neq \emptyset$)
{
  $todo = f(todo) - result$;
  $result = result \cup todo$;
}

This algorithm, expressed as a stored procedure, will be referred to as *Proc2*. One might expect *Proc2* to outperform *Proc1*, but as we shall see, this depends

on the characteristics of the underlying database system. Of course many more interesting fixpoint finding algorithms could be devised, and undoubtedly they would help to improve performance. In this paper, however, our aim is to assess the feasibility of implementing Datalog on top of a database system. We therefore restrict ourselves to the comparison of just these three variants: *CTE*, *Proc1* and *Proc2*.

Because our aim is a proof of concept, we have to ensure that our results do not depend on the peculiarities of one relational database management system. For that reason, we provide two backends for *CodeQuest*, one that targets Microsoft SQL Server 2005, and the other IBM DB2 v8.2. Our use of these systems is somewhat naïve, and no attempt has been made to tune their performance. It is very likely that an expert would be able to significantly improve performance by careful selection of the system parameters.

3.2 Querying Java Code

It is our aim to compare *CodeQuest* to JQuery, the leading code querying system for Java. For that reason, we have striven to make the *CodeQuest* frontend as similar as possible to JQuery, to ensure the experiments yield an accurate comparison. For the same reason, the information we extract from Java source and store in the database is the same with the information that JQuery collects. For elements, it consists exhaustively of packages, compilation units, classes, interfaces, all class members and method parameters. As for relational facts, we store hasChild, calls, fields reads/writes, extends, implements and returns relationships.

All these facts are not computed by *CodeQuest*: they are simply read off the relevant data structures in Eclipse, after Eclipse has processed a Java compilation unit. In what follows, the process of collecting information, and storing it in the database is called *parsing*. It is not to be confused with the translation from strings into syntax trees that happens in the Eclipse Java compiler. Naturally parsing is expensive (we shall determine exactly how expensive in Sect. 5), so in the next section we shall consider how *CodeQuest* achieves its parsing incrementally, making appropriate changes to the database relations when a compilation unit is modified.

We are currently working on the implementation of a robust user interface of our plugin for a neat integration within Eclipse. We also wish to develop a similar add-in for Visual Studio.

4 Incremental Database Update

Source code queries are typically performed for software development tasks within an interactive development environment, where frequent changes of the source code occur. Hence, the database of code facts needs be kept up-to-date with the source code developers are working on. Developers cannot afford, however, a reparsing of their entire project between successive modifications and

queries. A querying tool, embedded in a development environment, must provide an incremental update mechanism.

Yet such a feature is inherently similar to the tough problem of incremental compilation. Keeping the database in a consistent state, by specifying strong conditions for which the update of some facts must occur, is a complex task. To illustrate, consider a Java project with two packages *a* and *b*. Package *a* contains a class *A* and package *b* a class *B* declared with the code:

```
package b;
import a.A;
public class B {
    A theField;
}
```

At this stage, the type of *theField* is the class *a.A*. If we introduce a new class *A* in the package *b*, although no previously existing file has changed, the type of *theField* is now bound to *b.A*, and the relationship in the database should be updated accordingly.

Conveniently, Eclipse provides an auto-build feature that triggers a background incremental compilation of a project after each resource modification on that project. Eclipse tries to recompile as few compilation units as possible, but keeps the project in a consistent compiled state.

We leverage the auto-build feature of Eclipse to incrementally update the database when the developer modifies a Java resource. On notification by the Eclipse platform, we remove from the database all facts related to compilation units that are being deleted or recompiled. The cleaning is performed by deleting all compilation unit nodes and their children. These are computed using an *ad hoc* stored procedure generated by *CodeQuest* from the following query:

$$
\begin{aligned}
\mathit{hasChildPlus}(T, S) \;:-\;& \mathit{hasChild}(T, S)\;; \\
& \mathit{hasChild}(T, \mathit{MID}), \mathit{hasChildPlus}(\mathit{MID}, S). \\
\mathit{nodesToDelete}(N) \;:-\;& \mathit{compilationUnitsToDelete}(N)\;; \\
& \mathit{compilationUnitsToDelete}(C), \mathit{hasChildPlus}(C, N).
\end{aligned}
$$

All primitive relations, where one of these deleted children is involved, are also deleted, as well as empty packages. Then, *CodeQuest* simply reparses and stores facts about the compilation units that have been recompiled by Eclipse.

One might argue that compilation units provide too coarse a level of granularity for reparsing. Indeed, in principle one might attempt to do this at the level of class members, say, but keeping track of the relevant dependencies is likely to be complex. Furthermore, object-oriented programs have rather small compilation units. For the projects used in our experiments, the average number of lines of code per compilation unit varies from 81 to 233 lines per unit (see Table 1). That level of granularity, although pretty coarse, has proved to be very workable for our experiments with a series of refactoring queries discussed in the following section.

5 Experiments

In order to determine the performance characteristics – the usability, efficiency and scalability properties of the *CodeQuest* system, we have performed a number of experiments. We compare *CodeQuest* with two alternative approaches, namely JQuery (a mature code querying system by Kris de Volder *et al.* [34, 24]), and XSB which is an optimising implementation of Prolog.

The experiments can be divided into four categories:

- **General queries:** these are generally useful queries, of the kind one might wish to run on any project. They include both recursive and non-recursive queries. We shall use them to compare all three systems.
- **Project specific queries:** some examples of queries that are more specific and specialised for a particular project. It is our contention that such queries, relating to style rules and API conventions, are often desirable and necessitate a flexible code querying system beyond the capabilities of today's IDEs.
- **Program understanding:** program understanding is the most common use of source code querying system. It typically requires a series of queries to be run; here we take a series inspired by previous work on querying systems.
- **Refactoring:** this is the process of restructuring software to improve its design but maintain the same functionality. Typically it involves a series of queries to be executed and the appropriate changes applied to the source. This experiment illustrates that our method of keeping the database up-to-date (described in Sect. 4) is effective.

5.1 Experimental Setup

In our experiments we are going to compare the three versions of *CodeQuest* (*CTE*, *Proc1* and *Proc2*) on two different database systems (MS SQL and DB2), with the well known source code querying tool JQuery. To rule out the possibility that JQuery's performance problems are due to the fact that it was written in Java, we also compare against XSB, a state of the art optimising compiler for tabled Prolog that is written in C. We have not written an interface between XSB and Eclipse, however. Instead we modified the *CodeQuest* plugin to write its facts to a text file that is then read in by the XSB interpreter. In summary, there are eight different systems to compare: six versions of *CodeQuest* itself, plus JQuery and XSB.

For our experiments, we shall use four open-source Java applications of different size. The chosen projects range from very small one-man software projects to huge industrial multi-team projects with many developers around the world involved. Characteristics of the projects are summarised in the Table 1.

Most experiments were run on a Pentium IV 3.2GHz/HT machine with 1GB of memory running Windows XP. The XSB numbers, however, were obtained under Debian GNU/Linux with a quad Xeon 3.2Ghz CPU and 4GB of memory, as we encountered memory violations with XSB when trying to load a large number of facts on a machine with a lesser specification. The reader should

therefore bear in mind that our experimental setup is giving an advantage to XSB; as we shall see, that only strengthens our conclusions about scalability.

5.2 Running Experiments

Initial parsing. Before the queries can be run on a project it is parsed into a database and the time required is shown in Table 2. For all four projects, the time taken to build the relations in MSSQL is 5 to 7 times as much as it takes to compile them in Eclipse. The factor does *not* increase with the size of the project. For DB2, the situation is similar, but the factor is slightly higher (11 to 14). While this is a significant cost, it should be stressed that such complete builds are rare. When changes are applied to the program, the database is updated incrementally and usually there is no need for complete reparsing of the project. We shall measure the cost of such incremental updates when discussing queries for refactoring. We note that the cost of parsing in JQuery is very similar to that for *CodeQuest*, somewhere in between the MSSQL and DB2 versions. However, JQuery is not able to parse Eclipse. We do not provide parsing times for XSB, because as pointed out above, there we load facts indirectly, via a text file produced with a modification of the *CodeQuest* Eclipse plugin.

The high initial parsing cost of code querying systems is only justified if subsequent queries evaluate faster, and that is what we investigate next.

General queries. We start by considering three example queries, that represent typical usage of a code querying tool. They are not specific to a particular project.

The first query is checking a common style rule, namely that there are no declarations of non-final public fields. When such fields occur, we want to return both the field F and the enclosing type T. As a Datalog clause, this query might read as follows:

$$\begin{aligned} \mathit{query}1(T, F) :- \; & \mathit{type}(T), \mathit{hasChild}(T, F), \mathit{field}(F), \\ & \mathit{hasStrModifier}(F, \text{'public'}), \mathbf{not}(\mathit{hasStrModifier}(F, \text{'final'})). \end{aligned}$$

The above query is non-recursive. A little more interesting is the second example. Here, we wish to determine all methods M that write a field of a particular

Table 1. Summary information on benchmark Java projects

Application	Description	Number of java files	Source LOC	Source Classes
`Jakarta Regexp`	Java Regular Expression package	14	3265	14
`JFreeChart`	Java library for generating charts	721	92916	641
`abc` +Polyglot	extensible AspectJ compiler + framework	1644	133496	1260
`Eclipse`	Open Source Java IDE	12197	1607982	10338

Table 2. Required parsing time for the Java projects (hh:mm:ss)

Application	Compile	Relation parsing (MSSQL/DB2/JQuery)	Ratio (parse/compile) (MSSQL/DB2/JQuery)
`Jakarta Regexp`	00:00:01	00:00:07/00:00:12/00:00:06	07/12/06
`JFreeChart`	00:00:15	00:01:29/00:03:25/00:02:35	06/14/10
`abc` (+Polyglot)	00:00:28	00:02:41/00:06:12/00:04:45	06/13/10
`Eclipse`	00:09:23	00:44:45/01:34:46/—:—:—	05/11/—

type, say T. In fact, fields whose type is a subtype of T qualify as well. We therefore specify:

$$\begin{aligned} query2(M, T) :- \; & method(M), writes(M, F), hasType(F, FT), \\ & hasSubtypeStar(T, FT). \end{aligned}$$

Here the main relation of interest is $hasSubtypeStar(T, FT)$, which relates a type T to its subtype FT. It is defined as:

$$\begin{aligned} hasSubtypeStar(T, T) &:- type(T). \\ hasSubtypeStar(T, S) &:- hasSubtypePlus(T, S). \end{aligned}$$

where *hasSubtype* and *hasSubtypePlus* are relations previously discussed in Sect. 2.

The third query is to find all implementations $M2$ of an abstract method $M1$. Naturally Eclipse also provides a way of answering this query, and indeed it is a good example of how those fixed facilities are subsumed by a general code querying system. The query reads:

$$\begin{aligned} query3(M1, M2) :- \; & hasStrModifier(M1, \text{'abstract'}), overrides(M2, M1), \\ & \mathbf{not}(hasStrModifier(M2, \text{'abstract'})). \end{aligned}$$

The definition of *overrides* does also make use of the recursively defined *hasSubtypePlus*:

$$\begin{aligned} overrides(M1, M2) :- \; & strongLikeThis(M1, M2), \\ & hasChild(C1, M1), hasChild(C2, M2), \\ & inheritableMethod(M2), hasSubtypePlus(C2, C1). \end{aligned}$$

In words, we first check that $M1$ has the same signature and visibility as $M2$, since a protected method (say) cannot override a public one. We also check that $M2$ can actually be overridden (so it's not static, for example). When these two conditions are satisfied, we find the containing types of $M1$ and $M2$, and check that one is a subtype of the other.

Let us now consider different systems and their properties. Figure 1 presents the evaluation times of each system for the three queries. For each query, we show eight different ways of evaluating it [systems are listed in the legend of the chart in the same top-down order as the corresponding bars apper in left-right order; in the colour version of this paper, the correspondence is further enhanced

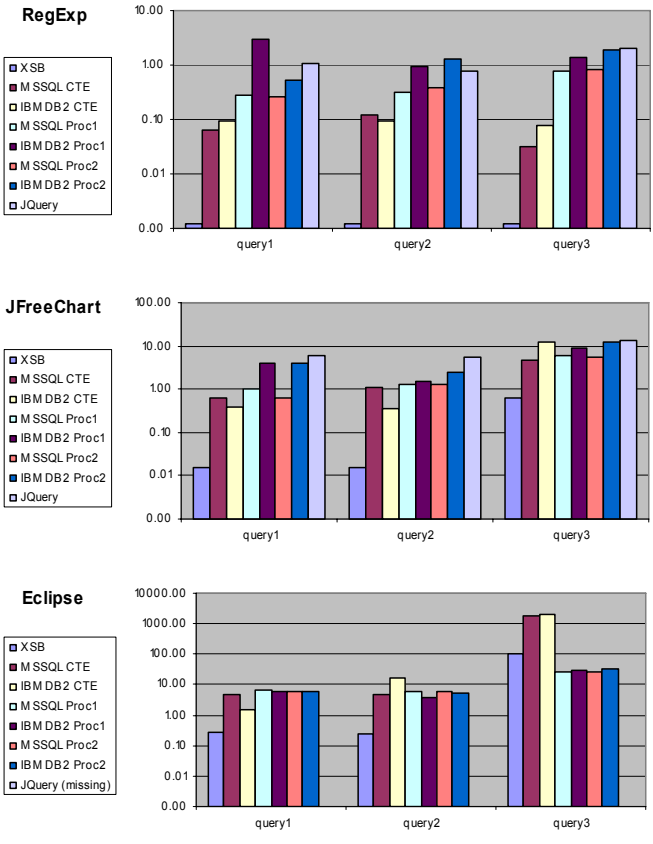

Fig. 1. General queries evaluation times

via colours]. On the vertical axis, we show the time taken in seconds – note that this is log-scale.

CodeQuest vs. JQuery. Most of the *CodeQuest* implementations proved to be more efficient than JQuery, for each of the three queries. The bars for JQuery are missing for the last graph because it was impossible to parse Eclipse with JQuery. It appears, therefore, that while JQuery is very effective for medium size projects, it does not scale to large projects. That is in line with the design goals of JQuery, namely to provide a light-weight, pure Java Eclipse plugin.

CodeQuest vs. XSB. It is natural to wonder whether a more efficient implementation of tabled Prolog such as XSB would yield a system similar to JQuery but with better efficiency characteristics. Our experiments confirm that this is indeed the case, and that Prolog outperforms *CodeQuest*. Notice, however, the exponential growth of time (with respect to the size of the project) required by XSB. Furthermore we have observed that XSB strongly depends on main memory, and for large projects that memory consumption becomes prohibitive (as we

shall see in a query involving the call graph below). It therefore lacks scalability, whereas *CodeQuest* shows much slower growth of time against project size, for each of the queries. It is also important to mention that programs and queries for the XSB system were optimised by hand (distinguishing modes, appropriate use of cut, and tabling), so that their evaluation occurs in the best possible order and excludes all unnecessary computations. Less carefully optimised programs for XSB require considerably more time to execute as will be shown in the following subsection.

CTEs vs. Procs. We now turn to the comparison of the two implementations of recursion that we described in Sect. 3: via a built-in feature of the DBMS, namely Common Table Expressions, or via stored procedures. There is a remarkable difference in evaluation times between these two approaches. *CodeQuest Proc1* and *Proc2* have slightly worse performance than *CodeQuest* CTEs for all non-recursive queries as well as for recursive queries over small code bases. The situation changes significantly, however, with the recursive queries over large amounts of source code. It seems that it is the creation of intermediate tables in the stored procedures approach that causes a certain overhead. But the least fixpoint computation algorithm, implemented using stored procedures, proves to be more efficient, as we see in computationally expensive queries.

Proc1 vs. Proc2. Proc2 has an optimised algorithm for computing the fixpoint of recursively defined relations. It is therefore equivalent to *Proc1* for non-recursive queries, and it should be more efficient for recursive ones. The downside of the *Proc2* algorithm is that it extensively creates and drops temporary tables. Thus, there is no efficiency gain for recursive queries over small size projects. Somewhat to our surprise, *Proc2* also does worse than *Proc1* on top of DB2, contrary to the situation for MSSQL. In more complex queries, for instance those that involve the call graph (discussed below), *Proc2* pays off even on DB2.

MSSQL vs. IBMDB2. It is clear from the graphs that usually the *CodeQuest* implementation on top of IBM DB2 is less efficient than on top of MS SQL Server. We have found that this may be somewhat sensitive to the exact form of the SQL that is produced by our compiler from Datalog. For instance, in DB2 it is better to avoid generating *not exists* clauses in the code. Furthermore, we note that: 1) we did not resort to the help of a professional database administrator and it is very likely that the database systems we were using could be tuned to increase performance significantly; 2) creation and deletion operations in IBM DB2 are generally more expensive than in MS SQL Server and since they are extensively used in the *Proc2* algorithm, the performance gain through a lesser number of joins was overwhelmed by the performance loss of a bigger number of creation/deletion of temporary tables. Nevertheless, both implementations prove that the concept of building a query system with a RDBMS at its backend is both efficient and scalable.

Project specific queries. While one can spend a lot of time trying to come up with the best possible optimisations for a general query, it is not quite possible

when queries are written frequently and are specific to different projects. In this subsection we want to run exactly such experiments.

Most of the coding style constraints in an object oriented software system are implicit and cannot be enforced by means of the programming language. Therefore it is desirable to run queries to ensure that such constraints are satisfied. *abc* is an AspectJ compiler based on an extensible compiler framework called Polyglot [10,35]. One of the coding style constraints in Polyglot is the following: every concrete AST class (an AST class is one that implements the *Node* interface), that has a child (a field which is also subtype of *Node*) must implement a *visitChildren* method. In order to check whether that constraint holds, we write the following query:

$$
\begin{aligned}
&\mathit{existsVChMethod}(C) :- \mathit{class}(C), \mathit{hasChild}(C, M), \mathit{method}(M),\\
&\qquad\qquad\qquad\quad \mathit{hasName}(M, \text{`visitChildren'}).\\
&\mathit{nodeInterface}(N) \quad :- \mathit{interface}(N), \mathit{hasName}(N, \text{`Node'}).\\
&\mathit{concreteClass}(C) \quad :- \mathit{class}(C), \mathbf{not}(\mathit{hasStrModifier}(C, \text{`abstract'})).\\
&\mathit{query1}(C) :- \mathit{nodeInterface}(N), \mathit{concreteClass}(C),\\
&\qquad\qquad\quad \mathit{hasSubtypePlus}(N, C), \mathit{hasChild}(C, F), \mathit{hasType}(F, T),\\
&\qquad\qquad\quad \mathit{hasSubtypeStar}(N, T), \mathbf{not}(\mathit{existsVChMethod}(C)).
\end{aligned}
$$

The $\mathit{existsVChMethod}(C)$ looks up all the classes that have methods called *visitChildren*. The $\mathit{nodeInterface}(N)$ respectively finds the interface with the name *Node* and $\mathit{concreteClass}(C)$ all the classes that are not abstract. The final part of the query is read as follows: find all concrete classes that are subtypes of type *Node* and have a child (field) of the same type, but there exists no method called *visitChildren* in that class.

The evaluation times of this query are given in Fig. 2(query1). In contrast to the general queries, we did not perform any complex hand-tuning of the Prolog queries. An obvious equivalent of the *CodeQuest* query has been taken.

The next query also applies to *abc* and the Polyglot framework. We would like to find all the methods that are not called (transitively) from *abc*'s *main* method. We expect to receive a list of methods that are defined to be called externally, or perhaps via reflection. Potentially we may encounter dead code here if a function neither reachable from the *main* nor from any of the extending modules.

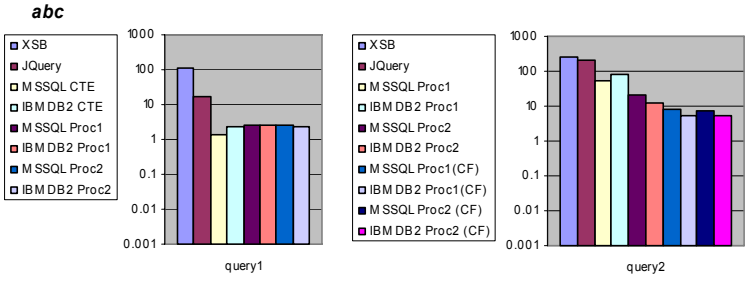

Fig. 2. Project specific queries evaluation times

$$
\begin{aligned}
&\mathit{polyCall}(M1, M2) &&:- \mathit{calls}(M1, M2).\\
&\mathit{polyCall}(M1, M2) &&:- \mathit{calls}(M1, M3), \mathit{overrides}(M2, M3).\\
&\\
&\mathit{polyCallPlus}(X, Y) &&:- \mathit{polyCall}(X, Y).\\
&\mathit{polyCallPlus}(X, Z) &&:- \mathit{polyCallPlus}(X, Y), \mathit{polyCall}(Y, Z).\\
&\\
&\mathit{mainCalls}(\mathit{Dummy}) &&:- \mathit{method}(\mathit{Main}), \mathit{hasName}(\mathit{Main}, \text{`main'}),\\
&&&\quad \mathit{polyCallPlus}(\mathit{Main}, \mathit{Dummy}).\\
&\\
&\mathit{query}2(\mathit{Dummy}) &&:- \mathit{method}(\mathit{Dummy}), \mathbf{not}(\mathit{mainCalls}(\mathit{Dummy})).
\end{aligned}
$$

We were unable to make this query evaluate successfully on systems other than *CodeQuest* with closure fusion (on the DB2 version, it takes 13 seconds). As the main purpose of this paper is to evaluate *CodeQuest* relative to other systems, we decided to run the query on *abc* sources only, excluding Polyglot. Naturally that means we do not catch call chains that occur via Polyglot, so the results of the query will be highly inaccurate.

In the results (Fig. 2(query2)) we have explicitly included query evaluation time for *CodeQuest* with and without the closure fusion optimisation. It is evident that this optimisation is highly effective for this example. Another important detail to mention here is that recursive relations such as *polyCallPlus* may have loops. For example, if method $m1$ (transitively) calls method $m2$ and method $m2$ again (transitively) calls method $m1$. Computation of recursive relations of this kind is almost impossible using Common Table Expressions in SQL. There are various work-arounds to this problem, but none of them is efficient and general. This is the reason why the numbers for the CTEs based implementation of *CodeQuest* are missing for this query. Finally, we note that for the XSB query, we did have to apply some obvious optimisations by hand to make it terminate at all, even when the code base was reduced by excluding Polyglot.

Program understanding. The most typical usage of a source code querying tool is undoubtedly program understanding. In this subsection we give an example of a program exploration scenario that involves a series of queries to be run consecutively as a programmer browses through the source. This scenario was loosely inspired by an earlier paper on JQuery [24].

JFreeChart is a free Java library for generating charts. Suppose a user would like to find out when the graph plots are redrawn. They might start by listing the packages and the classes defined in each one:

$$\mathit{query}1(P, T) :- \mathit{package}(P), \mathit{hasChild}(P, \mathit{CU}), \mathit{hasChild}(\mathit{CU}, T), \mathit{type}(T).$$

The user immediately spots the *plot* package where all kinds of plots are defined. Drawing is a standard operation and will be most likely defined in the supertype of all plots. Thus, he can pick any of the plot-types and search for its supertype:

$$\mathit{query}2(\mathit{SuperT}) :- \mathit{type}(\mathit{PickedType}), \mathit{hasSubtypePlus}(\mathit{SuperT}, \mathit{PickedType}).$$

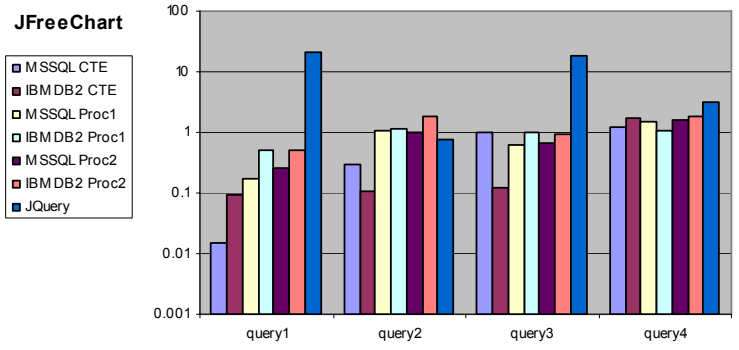

Fig. 3. Program understanding queries evaluation times

where *PickedType* is the type, chosen by the programmer. The result of this query will find an abstract *Plot* class. To list all its methods, the user defines the following query:

$$query3(M) :- hasChild(AbstractPlotType, M), method(M).$$

In the list the user finds an abstract method *draw* and he can finally define a query to spot all calls to this method or any overriding method in an extending class:

$$query4(M2) :- hasChild(C2, M2), polyCalls(M2, DrawM).$$
$$query4(M2) :- hasChild(C2, M2), polyCalls(M2, TargetM), overrides(TargetM, DrawM).$$

Both JQuery and RDBMSs support some form of caching. As the cache warms up it requires typically less time to evaluate subsequent queries. This is especially crucial factor for JQuery since it is known to have strong caching strategies and run much faster on a warm cache. Figure 3 presents the comparison graph for the above scenario for JQuery and *CodeQuest*.

The *CodeQuest* system again shows better results. In retrospect, this is not that surprising, since RDBMSs also traditionally possess caching mechanisms to limit the number of disk I/Os. In addition to that, as described in Sect. 7 further optimisations can be included in the *CodeQuest* system itself.

Refactoring. The following refactoring scenario is inspired by a feature request for JFreeChart [40]. The task is to create an interface for combined plot classes and make it declare methods common to these classes, notably *getSubplots()*. We compare JQuery with the *Proc2* version of *CodeQuest* . We start by writing a query to locate the combined plot classes:

$$classesToRefactor(C) :- class(C), hasName(C, Name), re\_match(\text{'\%Combined\%'}, Name), declaresMethod(C, M), hasName(M, \text{'getSubplots'}).$$

In words, this query looks for a class whose name contains the substring *Combined*, which furthermore declares a method named *getSubplots*. Evaluation of this query yields four elements: *CombinedDomainCategoryPlot*, *CombinedDomainXYPlot*, *CombinedRangeCategoryPlot* and *CombinedRangeXYPlot*.

We perform the first refactoring, by making the four classes implement a new interface *CombinedPlot* that declares a single method *getSubplots()*. This refactoring involves a sequence of operations in Eclipse, in particular the application of built-in refactorings such as *'Extract Interface'* and *'Use Supertype Where Possible'* as well as some minor hand coding.

The next step is to look for other methods than *getSubplots()*, common to the four refactored classes, whose declarations could be pulled up in the new interface. A query for this task reads as follows:

$$
\begin{aligned}
&\mathit{overridingMethod}(M) \;:\!-\; \mathit{overrides}(M, N).\\
&\mathit{declares}(C, S) \;:\!-\; \mathit{class}(C), \mathit{declaresMethod}(C, M),\\
&\qquad\qquad \mathit{hasSignature}(M, S), \mathbf{not}(\mathit{overridingMethod}(M)).
\end{aligned}
$$

$$
\begin{aligned}
\mathit{declarations}(S) \;:\!-\; &\mathit{class}(C1), \mathit{hasName}(C1, \text{'CombinedDomainCategoryPlot'}),\\
&\mathit{class}(C2), \mathit{hasName}(C2, \text{'CombinedDomainXYPlot'}),\\
&\mathit{class}(C3), \mathit{hasName}(C3, \text{'CombinedRangeCategoryPlot'}),\\
&\mathit{class}(C4), \mathit{hasName}(C4, \text{'CombinedRangeXYPlot'}),\\
&\mathit{declares}(C1, S), \mathit{declares}(C2, S),\\
&\mathit{declares}(C3, S), \mathit{declares}(C4, S).
\end{aligned}
$$

In words, we look for signatures of methods that are defined in all four classes of interest, which furthermore do not override some method in a supertype. Of course one might wish to write a more generic query, but as this is a one-off example, there is no need. The query yields two method signatures, ***double*** *getGap()* and ***void*** *setGap(**double**)*, which are related to the logic of the new interface. Hence, we perform a second refactoring to include these declarations in *CombinedPlot*.

This scenario provides a tiny experiment for measuring the efficiency of our incremental update mechanism and compare it to the one implemented in JQuery. An important difference between these two update mechanisms is the following. In *CodeQuest*, the update runs as a background task just after any incremental compilation is performed by Eclipse. In JQuery, the update occurs only when

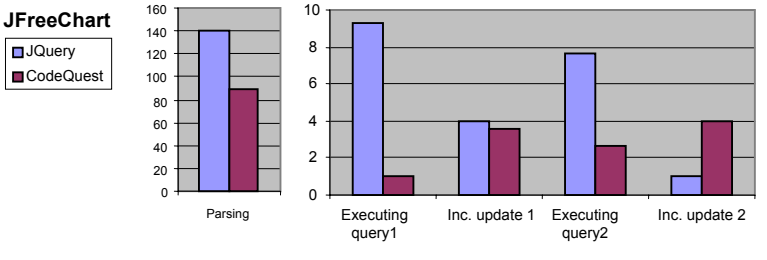

Fig. 4. Query evaluation and incremental update times for the Refactoring example

user explicitly executes the update action. The results are shown in Fig. 4. The sequence of measurements consists of the initial parsing time (which neither system needs to repeat after the first loading of the project), followed by two queries and updates.

In the given scenario the update times of the two systems are comparable. However, this refactoring example requires an update of very few facts. JQuery's performance considerably deteriorates when performing a larger change since it involves the deletion and recreation of many tiny files on the hard drive. For instance, if we apply the *Rename* rafactoring to the *org.jfree.data.general* package, update of 185 files will be required. It takes JQuery longer to update its factbase (over 5 mins) than to reparse the entire project again, whereas *CodeQuest* completes the update within 30 secs.

5.3 Effect of RDBMS Optimiser

Relational database systems not only enable code querying tools to be scalable for projects of virtually any size; another advantage lies in the powerful query optimisers, based on over forty years of intensive research. In order to illustrate the effect of the RDBMS optimiser we ran the project specific queries again, but this time with the optimiser switched off. *query1* (coding style constraints) was evaluated using CTEs based implementation of *CodeQuest* and *query2* (methods not transitively called from *main*) using *Proc2*. The evaluation time of *query1* with and without the internal IBM DB2 SQL optimiser is 2.7 and 5 seconds respectively. The difference becomes even more tangible for *query2*: 3.8 and 404 seconds respectively. Clearly it does not make sense for a code querying tool to try to re-implement all the work on optimisation already done for databases.

5.4 Memory Usage

We conclude this experimental section with a few brief remarks about memory usage. Both JQuery and XSB are memory hungry, and simply crash when there is not sufficient main memory available. Simple experiments confirm this behaviour: indeed, JQuery is unable to parse the source of Eclipse, and in XSB we could load it only under Debian Sid on a machine with 4GB of RAM. This sharply contrasts with the memory behaviour of database systems: these will use main memory where available, but performance is scalable. Because these results are entirely as expected, we do not present detailed numbers.

5.5 Summary

In this section we ran a variety of tests to measure performance of *CodeQuest* and to compare it against other similar systems. *CodeQuest* proved to be at least as efficient as JQuery in all case studies. Furthermore, simple techniques for storing intermediate results in temporary tables instead of recomputing them in every subsequent query could be added to already existent caching mechanisms of RDBMSs which would further leverage their performance. Of course that increased efficiency comes at the price of using a relational database system — there is much

merit in JQuery's lightweight approach, which does not require any additional software components.

By today's standards, considering both parameters of the hardware systems at hand and the size of software projects that require querying, *CodeQuest* is definitely competitive with XSB. The memory based computations of an optimised Prolog program are fast but not scalable. Non-optimised Prolog queries are clearly less efficient than the same queries evaluated with *CodeQuest*.
Today's industrial databases are able to evaluate recursive queries as described in the SQL99 standard. However, it appears that built-in recursion is often less efficient than custom algorithms using stored procedures. Furthermore, in some cases the built-in facilities do not work at all, in particular when an infinite number of duplicate entries might be generated in intermediate results. So, the choice between different implementations of *CodeQuest* with the tested RDBMS comes down to *Proc1* and *Proc2*. Formally *Proc2* is an optimised variant of *Proc1* and should therefore be more preferable. But in practice it requires creating and dropping temporary tables during each iteration step. If a database system has the cost of creation and dropping tables higher than a certain limit, then the optimisation becomes too expensive. In our experiments, *Proc2* is more efficient than *Proc1* in most of the queries when used in MS SQL Server and vice-versa when used in IBM DB2. More generally, code generation strategy (*CTE*, *Proc1* or *Proc2*) is tightly coupled with an internal RDBMS SQL optimiser. As a consequence of that, the choice of the appropriate *CodeQuest* implementation depends not only on the exact type of queries that a user may want to run, but also on the RDBMS and in particular on the SQL optimiser being used to run produced SQL code.

6 Related Work

There is a vast body of work on code queries, and it is not possible to cover all of it in a conference paper. We therefore concentrate on those systems that have provided an immediate inspiration for the design and implementation of *CodeQuest*. First we focus on work from the program maintenance community, then we discuss related research in the program analysis community, and we conclude with some recent developments that are quite different to *CodeQuest*, and could be seen as alternative approaches to address the same problems.

Storing the program in a database. In the software maintenance community, there is a long tradition of systems that store the program in a database. One of the earliest proposals of this kind was Linton's Omega system [32]. He stores 58 relations that represent very detailed information about the program in the INGRES database system. Queries are formulated in the INGRES query language QUEL, which is quite similar to SQL. There is no way to express recursive queries. Linton reports some performance numbers that indicate a poor response time for even simple queries. He notes, however, that future query optimisers ought to do a lot better; our experiments confirm that prediction.

The next milestone in this line of work is the C Information Abstraction system, with the catchy acronym CIA [14]. CIA deviates from Omega in at least two

important ways. First, based on the poor performance results of Omega, CIA only stores certain relations in the database, to reduce its size. Second, it aims for an incremental construction of the database — although the precise mechanism for achieving that is not detailed in [14], and there are no performance experiments to evaluate such incremental database construction. In *CodeQuest* we also store only part of the program, but it is our belief that modern database systems can cope with much larger amounts of data. Like CIA, we provide incremental updating of the database, and the way to achieve that efficiently was described in Sect. 4. CIA does not measure the effects of the optimiser provided by a particular database system, and in fact it is claimed the system is independent of that choice.

Despite their disadvantages, Omega and CIA have had quite an impact on industrial practice, as numerous companies now use a database system as a code repository, *e.g.* [13, 42].

Logic query languages. Both Omega and CIA inherited their query language from the underlying database system. As we have argued in Sect. 2, recursive queries, as provided in a logic programming language, are indispensable. Indeed, the XL C++ Browser [26] was one of the first to realise Prolog provides a nice notation to express typical queries over source code. The implementation is based directly on top of a Prolog system, implying that all the facts are held in main memory. As our experiments show, even with today's vastly increased memory sizes, using a state-of-the-art optimising compiler like XSB, this approach does not scale.

A particularly interesting attempt at overcoming the problem of expressiveness was put forward by Consens *et al.* [15]. Taking the search of graphs as its primary focus, GraphLog presents a query language with just enough power to express properties of paths in graphs, equivalent to a subset of Datalog, with a graphical syntax. In [15] a convincing case is made that the GraphLog queries are easier to write than the Prolog equivalent, and the authors state: "One of our goals is to have such implementations produced automatically by an optimizing GraphLog to Prolog translator." Our experiments show that to attain scalability, a better approach is to map a language like GraphLog to a relational database system.

Also motivated by the apparent inadequacy of relational query languages as found in the database community, Paul and Prakash revisited the notion of relational algebra [36]. Their new relational algebra crucially includes a closure operator, thus allowing one to express the traversals of the type hierarchy, call graph and so on that code queries require. The implementation of this algebra is done on top of the *Refine* system, again with an in-memory representation of the relevant relations [8]. Paul and Prakash report that hand-written queries in the *Refine* language typically evaluate a factor 2 to 10 faster than their declarative formulations. *CodeQuest* takes some of its inspiration from [36], especially in our use of relational algebra to justify optimisations such as closure fusion. Of course the connection between Datalog (our concrete syntax) and relational algebra with a (generalised) closure operator has been very thoroughly explored in the theoretical database community [7].

Also spurred on by the desire to have a convenient, representation-independent query language, Jarzabek proposed PQL, a Program Query Language [25]. PQL contains quite detailed information on the program, to the point where it is possible to formulate queries about the control flow graph. The query syntax is akin to that of SQL, but includes some operators for graph traversal. While SQL syntax undoubtedly has the benefit of being familiar to many developers, we feel that advantage is offset by its verbosity. Jarzabek points out that PQL admits many different implementations, and he describes one in Prolog. If so desired, it should be possible to use *CodeQuest* as a platform for implementing a substantial subset of PQL.

Another logic programming language especially for the purpose of source code queries is ASTLog, proposed by Crew [16]. Unlike PQL, it is entirely focussed on traversing the syntax tree, and there is no provision for graph manipulation. That has the advantage of a very fast implementation, and indeed ASTLog was used within Microsoft on some quite substantial projects.

The immediate source of inspiration for our work was JQuery, a source-code querying plugin for Eclipse [24, 34]. JQuery represents a careful synthesis of all these previous developments. It uses a logic programming language named TyRuBa [4]. This is similar to Prolog, but crucially, it employs tabled resolution for evaluating queries, which avoids many of the pitfalls that lead to non-termination. Furthermore, JQuery has a very nice user interface, where the results of queries can be organised in hierarchical views. Finally, it allows incremental building of the fact base, and storing them on disk during separate runs of the query interpreter. The main differences with *CodeQuest* are that TyRuBa requires mode annotations on predicates, and the completely different evaluation mechanism in *CodeQuest*. As our experiments show, that different evaluation mechanism is more scalable. The increased efficiency comes however at the price of less expressive power, as TyRuBa allows the use of data structures such as lists in queries, whereas *CodeQuest* does not. In JQuery, such data structures are used to good effect in building up the graphical views of query results. We feel this loss of expressiveness is a price worth paying for scalability.

Datalog for program analysis. The idea of using a tabled implementation of Prolog for the purpose of program analysis is a recurring theme in the logic programming community. An early example making the connection is a paper by Reps [38]. It observes that the use of the 'magic sets' transformation [12] (a generalised form of our closure fusion) helps in deriving demand-driven program analyses from specifications in Datalog.

A more recent paper in this tradition is by Dawson *et al.* [17], which gives many examples, and evaluates their use with the XSB system. We note that many of the examples cited there can be expressed in Datalog, without queries that build up data structures. As it is the most mature piece of work in applying logic programming to the realm of program analysis, we decided to use XSB for the experiments reported in Sect. 5. Our focus is not on typical dataflow analyses, but instead on source code queries during the development process.

Very recently Martin *et al.* proposed another PQL (not to be confused with Jarzabek's language discussed above), to find bugs in compiled programs [33,31]. Interestingly, the underlying machinery is that of Datalog, but with a completely different implementation, using BDDs to represent solution sets [43]. Based on their results, we believe that a combination of the implementation technology of *CodeQuest* (a relational database system) and that of PQL (BDDs) could be very powerful: source code queries could be implemented via the database, while queries that require deep semantic analysis might be mapped to the BDD implementation.

Other code query languages. Aspect-oriented programming represents a separate line of work in code queries: here one writes patterns to find all places in a program that belong to a cross-cutting concern. The most popular aspect-oriented programming language, AspectJ, has a sophisticated language of such patterns [28]. In IBM's Concern Manipulation Environment, that pattern language is deployed for interactive code queries, and augmented with further primitives to express more complex relationships [41]. We subscribe to the view that these pattern languages are very convenient for simple queries, but they lack the flexibility needed for sophisticated queries of the kind presented in this paper.

It comes as no surprise that the work on identifying cross-cutting concerns and code querying is converging. For example, several authors are now proposing that a new generation of AspectJ might use a full-blown logic query language instead [22,21]. The results of the present paper seem to suggest Datalog strikes the right balance between expressiveness and efficiency for this application also.

To conclude, we would like to highlight one effort in the convergence of aspects and code queries, namely Magellan. Magellan employs an XML representation of the code, and XML queries based on XQuery [19, 37]. This is natural and attractive, given the hierarchical nature of code; we believe it is particularly suitable for queries over the syntax tree. XQuery is however rather hard to optimise, so it would be difficult to directly employ our strategy of relying on a query optimiser. As the most recent version of Magellan is not yet publicly available, we were unable to include it in our experimental setup. An interesting venue for further research might be to exploit the fact that semi-structured queries can be translated into Datalog, as described by Abiteboul *et al.* [6] (Chapter 6).

7 Conclusion

In this paper, we have demonstrated that Datalog, implemented on top of a modern relational database system, provides just the right balance between expressive power and scalability required for a source code querying system. In particular, recursion allows an elegant expression of queries that traverse the type hierarchy or the call graph. The use of a database system as the backend yields the desired efficiency, even on a very large code base.

Our experiments also indicate that even better performance is within reach. A fairly simple, direct implementation of recursion via stored procedures often

outperforms the built-in facilities for recursion provided in today's database systems. More careful implementation of recursion, especially in conjunction with the query optimiser, is therefore a promising venue for further work.

At present the queries that can be expressed with *CodeQuest* are constrained by the relations that are stored in the database; we have closely followed JQuery in that respect, in order to make the experimental comparison meaningful. It is, in particular, impossible to phrase queries over the control flow of the program. There is however nothing inherently difficult about storing the relevant information. In fact, we plan to make the choice of relations configurable in *CodeQuest*, so the database can be adapted to the kind of query that is desired for a particular project.

We are particularly keen to see *CodeQuest* itself used as an engine for other tools, ranging from different query languages through refactoring, to pointcut languages for aspect-orientation. At present we are in the process of providing *CodeQuest* with a robust user interface; once that is complete, it will be released on the project website [5].

Acknowledgements

Elnar Hajiyev would like to thank Shell corporation for the generous support that facilitated his MSc at Oxford during 2004-5, when this research was started. We would also like to thank Microsoft Research (in particular Dr. Fabien Petitcolas) for its support, including a PhD studentship for Mathieu Verbaere. Finally, this research was partially funded through EPSRC grant EP/C546873/1. Members of the Programming Tools Group at Oxford provided helpful feedback at all stages of this research. We are grateful to Kris de Volder for many interesting discussions related to the topic of this paper.

References

1. *Eclipse*. `http://www.eclipse.org`.
2. *JQuery*. `http://www.cs.ubc.ca/labs/spl/projects/jquery/`.
3. *XSB*. `http://xsb.sourceforge.net/`.
4. *The TyRuBa metaprogramming system*. `http://tyruba.sourceforge.net/`.
5. *CodeQuest*. `http://progtools.comlab.ox.ac.uk/projects/codequest/`.
6. Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.
7. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
8. Leonor Abraido-Fandino. An overview of Refine 2.0. In *Procs. of the Second International Symposium on Knowledge Engineering and Software Engineering*, 1987.
9. Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.
10. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. *abc*: An extensible AspectJ compiler. In *Aspect-Oriented Software Development (AOSD)*, pages 87–98. ACM Press, 2005.

11. Roland Backhouse and Paul Hoogendijk. Elements of a relational theory of datatypes. In Bernhard Möller, Helmut Partsch, and Stephen Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 7–42. Springer Verlag, 1993.
12. François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts*, pages 1–16. ACM, 1986.
13. Cast. Company website at: `http://www.castsoftware.com`.
14. Yih Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.
15. Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 138–156, New York, NY, USA, 1992. ACM Press.
16. Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997.
17. Stephen Dawson, C. R. Ramakrishnan, and David Scott Warren. Practical program analysis using general purpose logic programming systems. In *ACM Symposium on Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.
18. Henk Doornbos, Roland Carl Backhouse, and Jaap van der Woude. A calculational approach to mathematical induction. *Theoretical Computer Science*, 179(1–2):103–135, 1997.
19. Michael Eichberg, Michael Haupt, Mira Mezini, and Thorsten Schäfer. Comprehensive software understanding with sextant. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 315–324, Washington, DC, USA, September 2005. IEEE Computer Society.
20. Hervé Gallaire and Jack Minker. *Logic and Databases.* Plenum Press, New York, 1978.
21. Stefan Hanenberg Günter Kniesel, Tobias Rho. Evolvable pattern implementations need generic aspects. In *Proc. of ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 116–126. June 2004.
22. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-oriented Software Development*, pages 60–69. ACM Press, 2003.
23. Elnar Hajiyev. CodeQuest: Source Code Querying with Datalog. MSc Thesis, Oxford University Computing Laboratory, September 2005. Available at `http://progtools.comlab.ox.ac.uk/projects/codequest/`.
24. Doug Janzen and Kris de Volder. Navigating and querying code without getting lost. In *2nd International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003.
25. Stan Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering*, 24(3):197–215, 1998.
26. Shahram Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Tsuyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, and Richard Helm. Architecture of the XL C++ browser. In *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 369–379. IBM Press, 1992.
27. Karel Ježek and Vladimír Toncar. Experimental deductive database. In *Workshop on Information Systems Modelling*, pages 83–90, 1998.

28. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *European Conference on Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.
29. Bronislaw Knaster. Un théorème sur les fonctions d'ensembles. *Annales de la Societé Polonaise de Mathematique*, 6:133–134, 1928.
30. Kemal Koymen. A datalog interface for SQL (abstract). In *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation*, page 422, New York, NY, USA, 1990. ACM Press.
31. Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2005. ACM Press.
32. Mark A. Linton. Implementing relational views of programs. In Peter B. Henderson, editor, *Software Development Environments (SDE)*, pages 132–140, 1984.
33. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN OOPSLA Conference*, pages 365–383, 2005.
34. Edward McCormick and Kris De Volder. JQuery: finding your way through tangled code. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN OOPSLA conference*, pages 9–10, New York, NY, USA, 2004. ACM Press.
35. Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, 2003.
36. Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996.
37. Magellan Project. Web page at: `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/Magellan/XIRC.html`. 2005.
38. Thomas W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases, ILPS*, pages 163–196, 1993.
39. Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 442–453, New York, NY, USA, 1994. ACM Press.
40. Eric Sword. Create a root *combinedplot* interface. JFreeChart feature request: `http://sourceforge.net/tracker/index.php?func=detail&aid=1234995&group_id=15494&atid=365494`, 2005.
41. Peri Tarr, William Harrison, and Harold Ossher. Pervasive query support in the concern manipulation environment. Technical Report RC23343, IBM Research Division, Thomas J. Watson Research Center, 2004.
42. Michael Thompson. Bluephoenix: Application modernization technology audit. Available at: `http://www.bitpipe.com/detail/RES/1080665824_99.html`., 2004.
43. John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog and binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780, pages 97–118. Springer-Verlag, November 2005.