Analysing the Entire Wikipedia History with Database Supported Haskell

George Giorgidze[1], Torsten Grust[1], Iassen Halatchliyski[2], and Michael Kummer[3]

[1] University of Tübingen, Germany
[2] Knowledge Media Research Center, Tübingen, Germany
[3] Centre for European Economic Research, Mannheim, Germany

Abstract. In this paper we report on our experience of using Database Supported Haskell (DSH) for analysing the entire Wikipedia history. DSH is a novel high-level database query facility allowing for the formulation and efficient execution of queries on nested and ordered collections of data. DSH grew out of a research project on the integration of database querying capabilities into high-level, general-purpose programming languages. It is an emerging trend that querying facilities embedded in general-purpose programming languages are gradually replacing lower-level database languages such as SQL as preferred facilities for querying large-scale database-resident data. We relate this new approach to the current practice which integrates database queries into analysts' workflows in a rather ad hoc fashion. This paper would interest early technology adopters interested in new database query languages and practitioners working on large-scale data analysis.

1 Introduction

Relational database systems provide scalable and efficient query processing capabilities for complex data analysis tasks. Despite these capabilities, database systems often do little more than to hold and reproduce data items for further processing and analysis with the help of a programming language. In this typical setup, the lion share of data analysis tasks thus happens outside the realm of the database system. This is partly because, for involved analyses, relational database systems require the mastering of advanced features of specialised database query languages, such as SQL. This requirement represents a barrier to many practitioners who need to analyse large-scale data. In this paper, we report on what this means for social scientists interested in Wikipedia data.

Unfortunately, transferring database-resident data into the runtime heap of a programming language is not always an option (for example, if the data size is larger than the available main memory). Even if the transfer is possible, it may be wasteful if the final result of the computation is small. One approach addressing the aforementioned problems is to use relational database systems and their query processing capabilities as *coprocessors for programming languages* [4]. This approach entails the automatic translation of the data-intensive parts of a given

program into relational queries (SQL statements, say) that can be executed by a database engine. In effect, the relational database system transparently participates in program evaluation. Most importantly, data-intensive computations are performed by the database engine and thus close to the data.

Under this approach, SQL is regarded as a lower-level target language into which more familiar and expressive source languages are translated. It is an emerging trend to rely on higher-level querying facilities for the analysis of large-scale database-resident data. Perhaps the most widely used example of such a query facility is LINQ [10] which seamlessly adds advanced query constructs to the programming languages of Microsoft's .NET framework.

In this paper we report on our experience of using *Database Supported Haskell* (DSH) [4] for analysing the entire history of Wikipedia. DSH is a novel high-level database query facility allowing for the formulation and efficient execution of queries over database-resident collections of data. Support for nested and ordered data structures as well as powerful programming abstractions is what distinguishes DSH from other language-integrated database query facilities.

DSH grew out of a research project on the tight integration of database querying capabilities into high-level, general-purpose programming languages. While the project tackles foundational issues of the embedding and compilation of rich query languages [7, 4, 5], here we report on our efforts to team up with colleagues of the social sciences to address topics in large-scale Wikipedia analysis. The use cases of the following sections are taken from current studies on the collaborative construction of knowledge [8, 9].

This paper would interest early technology adopters interested in new database query languages and practitioners working on large-scale data analysis. The remainder of the paper is structured as follows. In Section 2, we describe an earlier study performed by the third author on a small subset of the Wikipedia data. The data analysis tasks of this study were implemented in terms of a (what we assume to be typical) ad hoc embedding of SQL queries into R scripts. In Section 3, we describe how we scaled the study to the entire Wikipedia data using DSH. In Section 4, we outline future work and conclude the paper.

2 A Bilingual Approach Based on SQL and R

In a study by the third author of the present paper the development of new knowledge in Wikipedia was analysed at the level of knowledge domains [8]. The study was based on 4,733 articles and 4,679 authors in the equally large adjacent knowledge domains of physiology and pharmacology. The *boundary spanners* were shown to be a highly relevant group of authors for the whole knowledge-creating community: these authors work on integrative boundary articles between domains and also on the central articles within a single domain.

The analysis has been implemented in terms of a mixture of two main technologies, SQL and R. Relevant data was sourced from a MySQL database dump of the German Wikipedia that contained all logged data on articles and authors. Multiple SQL queries were textually embedded into a single script of R code

```
1  library(RODBC)
2  con <- odbcConnect("···")
3  # retrieve articles in the physiology and pharmacology domains
4  phyA <- dbGetQuery(con,"SELECT pd_page_id FROM pagedomains
5                          WHERE (pd_domain = 'physiology')")
6  phaA <- dbGetQuery(con,"SELECT pd_page_id FROM pagedomains
7                          WHERE (pd_domain = 'pharmacology')")
8  allA <- unique(rbind(phyA,phaA))
9  # Establish direct links between the retrieved articles
10 links <- dbGetQuery(con,
11   paste( "SELECT pl_from, page_id FROM pagelinks, page
12             WHERE pl_title = page_title
13               AND page_namespace = 0
14               AND page_is_redirect = 0
15               AND pl_from IN (", toString(allA[,1]), ")
16               AND page_id IN (", toString(allA[,1]), ")", sep = ""))
```

Fig. 1. R script snippet supporting the analysis of boundary-spanning articles and authors in the German Wikipedia.

that drove the analysis. We consider this to be a representative setup. Similar studies have used other host languages (e.g., Perl, Python, and C). Our general observations below remain valid, however. It is inherent to this approach that the analyst is *bilingual*, capable of speaking two languages and also fit to translate logic as well as data formats between the two (SQL and host language).

Figure 1 displays a snippet from the mentioned R script. The code is sprinkled with SQL fragments: the variable assignment `phyA <- dbGetQuery(con,...)` sends the quoted SQL statement to a connected database server for execution and binds the resulting n-ary table to the R variable `phyA`, a data frame (or matrix) of n columns.

A number of issues make this style of data-intensive programming problematic for the analyst. Among the more pressing issues are the following:

(Lack of) Static safety. R does not understand the quoted SQL text and sends it as is. The text may contain syntactically invalid or even harmful SQL code [13]. This is important if fragments of SQL text are spliced at script runtime. The lack of static safety and the possibility of run time failures is particularly problematic for long running programs as a runtime error may require a restart of the entire computation, including those (potentially long-running) parts executed before the failure occurred.

Query text size and number of queries. Note how the `toString()` calls in lines 15 and 16 refer to a formerly computed query result (R variable `allA`). The size of these splices, and thus of the overall generated query text, depends on the queried data and thus must, in principle, be considered arbitrarily large. In effect, intermediate query results (here bound to `allA`) are carried from query to query in textual form. Besides the obvious inefficiency, this approach may

# Articles	SQL+R ⏱ (sec)	DSH ⏱ (sec)
10,000	46.25	1.15
20,000	108.89	2.97
30,000	181.63	3.80
40,000	263.77	4.64
50,000	353.31	5.54

Table 1. Wall-clock execution times for the R and DSH program fragments from Sections 2 and 3.

easily overwhelm the database system's SQL parser or compiler: both `IN` clauses in line 15 and 16 end up containing almost five thousand literals (the aforementioned 4,733 page identifiers, to be more precise).

Alternatively, to avoid the construction of huge queries, carried intermediate results may be used to issue multiple separate but simple queries in an iterative fashion. This mode of data-intensive computation, however, incurs considerable run time overhead associated with the now frequent switches between R's run-time system and the database system [7, 4]. Note that in this latter scenario, the boundary spanner analysis would lead to at least 4,733 of these costly context switches. Scalability clearly suffers.

Computation outside the database realm. The various SQL fragments contribute sizeable intermediate results which are materialised on the R heap and then threaded through the script. R operations (e.g., `unique`: duplicate elimination, `rbind`: union, `[,]`: projection) are used to perform data-intensive computation on the R heap while the database server would have been perfectly able to do the job more efficiently close to the source data. Again, this raises severe scalability and performance issues.

Host language dictates execution granularity and order. Following a typical style, the R script breaks the data-intensive portion of the computation down into parts that lead to intelligible pieces of SQL code. The synchronous execution of these pieces through `dbGetQuery()` prescribes a granularity and order of query evaluation that leaves little room for query optimisation and prevents query scheduling by the database engine.

The small snippet of R code in Figure 1 exhibits all four issues mentioned above. These issues are instances of problems of programming language and database interoperability that have long been identified [3]. The present paper was motivated by the assumption that vibrant areas of data-intensive research, such as the Wikipedia analysis, would particularly benefit from a modern account of database integration.

```
phyA :: Q [Integer]
phyA = [pd_page_idQ p | p ← pageDomains, pd_domainQ p ≡ "physiology"]
phyA :: Q [Integer]
phyA = [pd_page_idQ p | p ← pageDomains, pd_domainQ p ≡ "pharmacology"]
allA :: Q [Integer]
allA = nub (phyA ++ phaA)
links :: Q [(Integer, Integer)]
links = [ tuple (pl_fromQ l, page_idQ p)
        | p ← pages
        , l ← pageLinks
        , pl_titleQ l ≡ page_titleQ p
        , page_namespaceQ p ≡ 0
        , page_is_redirectQ p ≡ 0
        , pl_fromQ l ∈ allA
        , page_idQ p ∈ allA]
```

Fig. 2. DSH definitions corresponding to the R script given in Figure 1.

3 A Unilingual Approach Based on DSH

Figure 2 gives DSH definitions corresponding to the R script from Figure 1. DSH allows for formulation of database executable program fragments using Haskell's list prelude and the monad comprehension notation [5]. The former turns DSH into an expressive database query facility for nested and ordered collections of data, while the later ensures that DSH queries can be expressed concisely in a widely understood and adopted notation.

Unlike the R script, *all* DSH definitions are database-executable. This provides significant performance benefits as shown in Table 1. In this benchmark, the increase in the number of Wikipedia articles imitates the scenario where the total number of articles in the two domains of interest is much larger than in physiology and pharmacology. As for the rest of the evaluation setup, we used: the latest complete German Wikipedia database dump (June 3, 2012) with 3,958,157 entries in the `page` table and 85,969,266 entries in the `pagelinks` table, version 9.1.4 of the PostgreSQL database management system, the Arch Linux distribution with kernel-3.4.4, version 2.15.0 of the R system, version 7.4.1 of the Glasgow Haskell Compiler (GHC), and a host equipped with an Intel Xeon X5570 CPU.

How does DSH-based data analysis fare if compared with the widely deployed ad hoc embedding approach? Have the major issues of bilingual data analysis actually been addressed?

Static safety. DSH's query compiler guarantees that the translation of database-executable program fragments will not fail at run time and the translation will generate valid database-executable SQL queries. If a computation has been tagged with the type constructor Q but cannot actually be performed in-

side the database system DSH will reject the program right from the start. No analyst time is wasted due to late failure at analysis run time.

Query text size and number of queries. With DSH, the number of generated database queries and their query text size does *not* depend on the size of the queried data. In fact, the number of generated SQL queries is predictable: it is statically determined by the *type* of the database-executable fragment. Specifically, the number of generated queries equals the number of list type constructors (i.e., $[\cdot]$) in the type. For details about the compilation technology that provides this essential guarantee the reader may refer to [7, 4].

Computation outside the database realm. DSH allowed us to perform all of our analysis using the underlying relational database system, close to the data. This was instrumental to scale our analysis to the entire Wikipedia history in our follow-up study investigating an economic view of knowledge creation [9].

Host language dictates execution granularity and order. DSH provides the guarantee that values of Q-types are evaluated by the database coprocessor without unnecessary context switches between the host language runtime system and the relational database management system. This allows the database to determine the best possible query evaluation strategy as it has access to the entire query and not just parts of the query.

4 Further Reading, Conclusions and Future Work

Due to the limited space we are not able to discuss all of DSH's distinguished features in this paper; particularly its ability to handle nested and ordered collections, and its embedding and compilation aspects. For these topics the interested reader may refer to [7, 4, 5]. TryDSH, which is an interactive web-based environment for writing, executing, and inspecting the compilation pipeline of DSH queries, can be accessed from [2]. The source code of DSH is available from [1].

In this paper we reported on how we use DSH for large-scale data analysis of Wikipedia data. We observed that the widely used combination of SQL queries interleaved with data analysis in programming languages such as R does not scale for large data sets such as the entire revision, interlinkage, and page access history of the German Wikipedia. DSH allowed us to analyse this large data set from a high level of abstraction and, at the same time, perform the analysis almost entirely using a relational database management system.

One problem that we encountered while using DSH for the Wikipedia data analysis is that, in some DSH queries, we found it hard to reason about the performance behaviour of the resulting SQL queries. This is partly because of the current somewhat involved automatic translation of high-level DSH constructs to lower-level relational database languages. Although we managed to overcome the aforementioned problems with careful reformulations of the DSH queries, a more systematic solution involving changes in the DSH query compiler and optimiser would be beneficial. This would be yet another step towards our goal of allowing practitioners—who are not necessarily expert programmers or database

engineers—to analyse large-scale data, such as the entire Wikipedia history, using high-level and reusable domain-specific languages and libraries.

This paper focuses on DSH, which allows a relational database management system to be used as a coprocessor for the Haskell programming language. It is worthwhile to mention that the query compilation techniques featured in DSH are also being used to improve database query facilities in other languages such as C# [7], Ruby [6], Links [11] and Scala [12]. We conjecture that the tight integration of general-purpose programming languages with relational database management systems is a trend that will continue.

Acknowledgements

The authors work within an interdisciplinary cluster of researchers developing an infrastructure for the analysis of social networks. This work is funded by a grant from the ScienceCampus Tübingen. We would like to thank Manfred Knobloch, Thorsten Doherr and Frederic Schütz for helping us with various aspects of sourcing and processing a part of the data used for the analysis.

References

1. Database Supported Haskell (DSH). http://hackage.haskell.org/package/DSH.
2. TryDSH. http://dbwiscam.informatik.uni-tuebingen.de/trydsh/.
3. G. Copeland and D. Maier. Making Smalltalk a database system. *ACM SIGMOD Record*, 14(2):316–325, 1984.
4. G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell boards the Ferry: Database-supported program execution for Haskell. In *Revised Selected Papers of IFL 2010, Alphen aan den Rijn, Netherlands*, volume 6647 of *LNCS*. Springer.
5. G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing back monad comprehensions. In *Proc. of the Haskell Symposium 2011, Tokyo, Japan*. ACM.
6. T. Grust and M. Mayr. A deep embedding of queries into Ruby. In *Proc. of ICDE 2012*, Washington (DC), USA. IEEE.
7. T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *Proc. VLDB Endow.*, 3(1-2):162–172, 2010.
8. I. Halatchliyski, J. Moskaliuk, J. Kimmerle, and U. Cress. Who integrates the networks of knowledge in Wikipedia? In *Proc. of WikiSym 2010*. ACM.
9. M. Kummer, M. Saam, I. Halatchliyski, and G. Giorgidze. Centrality and content creation in networks the case of German Wikipedia. Technical Report 12-053, ZEW, Mannheim, Germany, 2012.
10. E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proc. of SIGMOD 2006*. ACM.
11. A. Ulrich. A Ferry-based query backend for the Links programming language. Master's thesis, University of Tübingen, 2011.
12. J. Vogt. Type safe integration of query languages into Scala. Master's thesis, RWTH Aachen University, 2011.
13. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. of PLDI 2007*, San Diego, CA, USA. ACM.