# Integrating and Querying Source Code of Programs Working on a Database

Carlos Garcia-Alvarado
University of Houston
Dept. of Computer Science
Houston, TX, USA

Carlos Ordonez
University of Houston
Dept. of Computer Science
Houston, TX, USA

## ABSTRACT

Programs and a database's schema contain complex data and control dependencies that make modifying the schema along with multiple portions of the source code difficult to change. In this paper, we address the problem of exploring and analyzing those dependencies that exist between a program and a database's schema using keyword search techniques inside a database management system (DBMS). As a result, we present `QDPC`, a novel system that allows the integration and flexible querying within a DBMS of source code and a database's schema. The integration focuses on obtaining the approximate matches that exist between source files (classes, function and variable names) and the database's schema (table names and column names), and then storing them in summarization tables inside a DBMS. These summarization tables are then analyzed with SQL queries to find matches that are related to a set of keywords provided by the user. It is possible to perform additional analysis of the discovered matches by computing aggregations over the obtained matches, and to perform sophisticated analysis by computing OLAP cubes. In our experiments, we show that we obtain an efficient integration and allow complex analysis of the dependencies inside the DBMS. Furthermore, we show that searching for data dependencies and building OLAP cubes can be obtained in an efficient manner. Our system opens up the possibility of using the keyword search for software engineering applications.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; H.2.4 [**Database Management**]: Systems—*Relational databases*

## General Terms

Algorithms, Experimentation, Languages

## Keywords

DBMS, Source Code, Software Maintainability, Integration

## 1. INTRODUCTION


http://doi.acm.org/10.1145/2254736.2254747

A variety of programs normally surround a database management system (DBMS) because they must access the data stored within the DBMS. Therefore, these applications exhibit a close relation with all the data sets stored in the DBMS in terms of their metadata (e.g. file name), classes, methods, variables, and SQL queries. This problem is evident when dealing with source code and a database's maintainability. As a result, a deep analysis of these data dependencies and querying capabilities of common elements that exist across all sources (joint querying) is required for application profiling, speeding up application development, estimating maintainability costs, and application debugging [9, 3]. Even though analyzing data dependency is a deeply studied problem within the compilers and software engineering community [2, 6, 15], we defend the idea that it is possible to solve this data and control dependency problem in a simpler manner by using keyword search techniques and allowing a joint exploration of these external sources. In order to do so, we focus on integrating both sources by finding those relationships that exist between a set of source code files (e.g. C++, Java, SQL scripts) and the physical model of a database.

To exemplify this, a brief scenario is described. In Figure 1, a central DBMS containing information about water quality of wells in the State of Texas and a set of surrounding source code files are shown. From these files, only a portion of them relates to the DBMS due to their file name, class name, variables, methods (procedures of functions) or SQL queries. From these two sources, we are interested in knowing which pieces of code and tables or columns are related directly (or indirectly) to assert the impact of a source code modification. Some queries that we are interested in answering are: "Which is the class that is more dependent on the database?", "Which are the most queried tables?", and "Is there a data dependency between a column in the database with a particular method?".

A number of challenges arise in solving the complex task of relating the content of source code files (including their metadata) and the physical design of a set of databases, which include database names, table names, and column names. Some of these problems include, but are not limited to: finding a conceptual representation of the data and control dependencies between a source code section and a portion of the database, efficient extraction of the existing matches between embedded SQL queries and the potential data and control dependencies given by class names, variable names and file metadata, and efficient and informative querying of the resulting matches.

**Table 1: Example of matches between $S$ and $P$ in the context of water quality.**

| name in $S$ | parent in $S$ | type in $S$ | keyword in $P$ | distance | type in $P$ | loc. in file | file |
|---|---|---|---|---|---|---|---|
| Region | null | table_name | Agency | 0 | metadata | filename | Agency.java |
| Pollutant | null | table_name | Pollutant | 0 | metadata | filename | Pollutant.java |
| Pollutant | null | table_name | Pollutant | 0 | SQL | SELECT * FROM Pollutant | Pollutant.java |
| name | Agency | column_name | name | 0 | method | public string name ( ) | Agency.java |
| ptype | Pollutant | column_name | ntype | 1 | var | String ntype = "" | Pollutant.java |

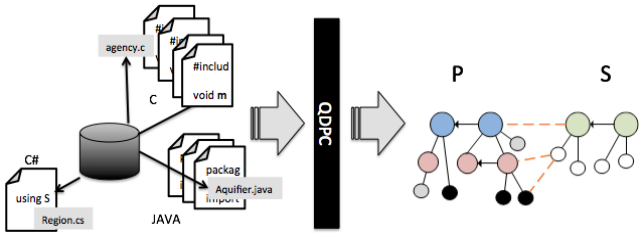



**Figure 1: Integration Scenario. The matches between a program $P$ and a schema $S$ are represented as dotted lines.**

Accounting for all of these challenges, we propose QDPC (Querying Database Programs and Code). This novel system is a collection of algorithms based on traditional SQL queries and database extensibility mechanisms. QDPC focuses on finding a dependency representation in the form of matches, and provides efficient algorithms for extracting and querying all the existing matches between the DBMS and a set of source files using the same central DBMS as the backbone for the entire process. The integration is performed using database extensibility features, such as User-defined functions (UDFs), and the resulting matches are stored inside relational tables for efficient querying using standard SQL queries. The result is a powerful system that allows the user to ask questions that go beyond a simple match extraction, such as finding all the tables that are related to a particular function, all the elements that are related through different dependencies, or OLAP queries.

This paper is organized as follows: Section 2 presents the definitions and notation used throughout the paper. Section 3 describes in detail the algorithms that are part of QDPC for integration and exploration of the sources. In Section 4, performance experiments are shown for the integration and exploration phases. Section 5 presents previous research in the area of source code analysis. Finally, Section 6 presents our conclusions and future work.

## 2. DEFINITIONS

QDPC takes as input a schema $S$ and a program (or source code repository) $P$. $S$ is composed of a set of tables $S = \{T_1, T_2, \dots\}$, where each $T_i$ contains attributes defined as $T_i(a_{i1}, a_{i2}, ...)$, and each $T$ and $a$ is mapped ($\mapsto$) to a keyword $k$. A program with $n$ files is defined as $P = \{v_1, v_2, \dots\}$, in which each $v_i$ is a keyword that represents metadata (e.g. filename), classes, methods (procedures or functions), variables or SQL queries. Furthermore, there exist directed relationships between tables $T$ of the form $T_i(a_{i1}, \dots) \rightarrow T_j(a_{j1}, \dots)$ given by the primary/foreign key referential integrity constraints, and control dependency relationships between classes and methods. There are additional undirected relationships (due to memberships) between classes, methods, variables, and queries, in which a class can have methods, variables and queries, and a method can contain some variables and queries. In Figure 1, we exemplify the relationships between all the elements of $S$ and $P$. Table 1 shows in detail several matches present in Figure 1.

QDPC also takes a query $Q$ which is composed of a set of keywords $Q = \{k_1, k_2, \dots\}$ in order to perform a uniform search in both sources. The searches are performed over a set of approximate matches between $P$ and $S$ mapped to a keyword. An approximate match of a keyword $k$ is obtained based on an edit distance function edit(k,k'), where the edit distance represents the minimum number of insertions, deletions, or substitutions required to turn $k$ into $k'$. Therefore, we consider an approximate match if and only if $\texttt{edit}(k, k') \leq \lceil \beta * |k| \rceil$, where $\beta$ is a real number in the interval [0,1]. Intuitively, the value for $\beta$ cannot be larger than 0.5 because it will accept many unrelated keywords as valid approximations. In the particular case in which exact matches are the only ones desired, the value of $\beta$ equals zero. The value of $\beta$ should be tuned based on the programming style in the source code problem. Because exact matches are clearly the more important ones, a small value of $\beta$ is normally desired (e.g. 0.10). Finally, the result of a query $Q$ is all the graphs, $G$, containing all elements (e.g. tables, columns, class, methods, variables, SQL queries) that are required to satisfy all the approximate matches between $P$ and $S$. The representation of all these elements in relational terms will be given in Section 3.

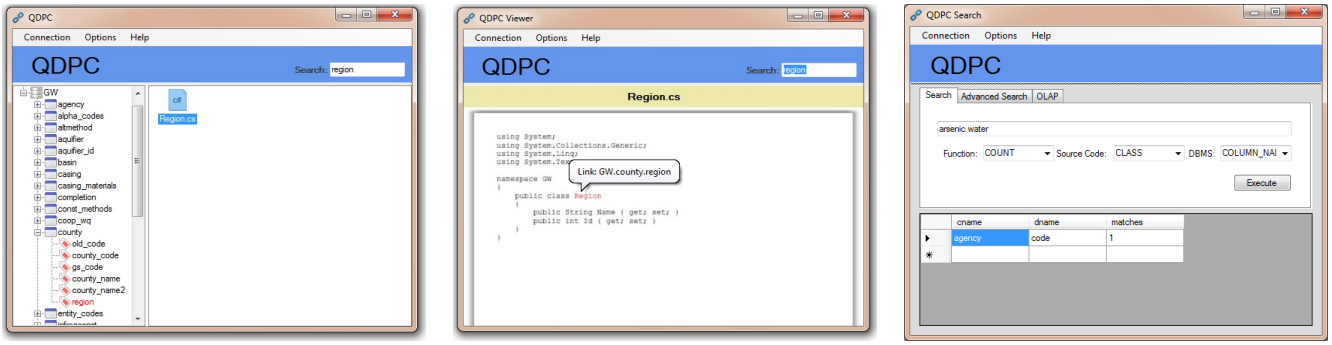
For example, let $S_1$ have two tables $T_1(a_{11}, a_{12}, a_{13})$ and $T_2(a_{21}, a_{22})$, where there $T_1(a_{12}) \rightarrow T_2(a_{21})$; and $P_1$ has a single class:

```
v1:class
begin
      v2:var
      v3:var
end
```

Given that $T_2 \mapsto k_1$ and $a_{12} \mapsto k_2$, and $\texttt{edit}(k_1, v_1) \leq \lceil \beta * |k_1| \rceil$ and $\texttt{edit}(k_2, v_2) \leq \lceil \beta * |k_2| \rceil$, then $(k_1, v_1)$ and $(k_2, v_2)$ represent a match between $S_1$ and $P_1$. If a query is given involving keywords $k_1$ and $k_2$, then the resulting graph is $G_1 = \{T_2, T_1, a_{12}, v_1, v_2\}$.

## 3. INTEGRATION AND QUERYING OF PROGRAMS AND SCHEMAS

QDPC is a system that allows joint keyword searches and complex analysis of the resulting matches to be performed. As a result, QDPC is composed of an integration phase and an extraction phase. Our system exploits a relational database as a backbone for efficient extraction and retrieval of matches. In other words, the entire process of managing both sources is performed within the DBMS. Therefore, both modules




a) Browsing Source Code and Schema. b) Source Code View. c) Assisted/Advanced querying.

**Figure 2: QDPC.**

rely on optimizations that exploit either SQL or UDFs, depending on the required task.

## 3.1 Integration

The integration is a refinement of an earlier idea presented in [8] for managing structured and unstructured sources. However, the analysis was limited to simple queries. The integration phase can be summarized as:

- Preprocessing
  1. Analyze source code files to extract all $v$.
  2. Extract control dependencies.
- Integration.
  1. Obtain unique keywords.
  2. Search for approximate matches in the preprocessed source code.
  3. Apply indexes to the resulting tables.

The notion of achieving the integration relies on a two-step process. The first step preprocesses all the source code files to extract all the metadata, class names, method names, and variables, as well as the calls that exist between them. This will generate three tables: a source code table (*P_document*) which contains an identifier for each $v \in P$ and the location in the file, *P_mcall* (caller method, method) which has the control dependencies between methods, and *P_ccall* (caller method, method) which has the control dependencies between classes. Notice that the $P$ prefix indicates that a table contains information regarding a program $P$. The integration step will rely on three main tables *M_predicate* (keyword id, keyword), *S_database* (keyword, location, parent) and *M_document* (keyword id, edit distance, type, parent method, parent class, location in file), in which *M_document* will hold all the approximate matches to the source code files. Similar to the $P$ prefix, the $S$ and $M$ prefixes relate to the schema and matching results, respectively.

The first task during the integration will focus on obtaining the unique keywords of the database's schema and finding all the approximate matches that exist between these unique keywords and every $v$ in $P$. Therefore, this process is bounded by the number of unique keywords and the number of files. During this integration phase, a batch of keywords is tested at a time. As a result, if the number of unique keywords in $S$ is small, the algorithm is only

```
using System;
namespace GW
{
   public class Alpha_codes
   {
      int _alpa_code;
      public int GET_ALPHA_CODE ()
      {
         return _alpa_code;
      }
      int _county_code;
      public int GET_COUNTY_CODE ()
      {
         return _county_code;
      }
      //...
   }
}
```

**Figure 3: Approximate matching.**

bounded by the number of discovered elements $v$ in the source code repository $P$. In order to find approximate matches, the Approximate Boyer-Moore algorithm (ABM) was used [14]. The rationale behind this is that by using this algorithm, it is possible to capture valid text patterns within the desired edit distance. For example, in Figure 3, the Alpha_code class, the _alpa_code, and _county_code variables are found. In addition, methods GET_ALPHA_CODE and GET_COUNTY_CODE are also identified. In our approach, only the declaration of a variable was taken into account. Once this discovery phase has been finished, indexes are applied to the summary tables containing the approximate matches. Indexes were also added to the unique keywords, source code, and metadata summary tables.

Matches between the database and the source code files are discovered as follows: a set of all the elements from the database ($T$ and $a$) is queried from the catalog in order to extract all of the unique keywords. This set of keywords will be stored in two summary tables. *M_predicate* will contain all of the unique keywords (obtained with a UNION query) and *S_database*, which will contain the location of the elements in the database. The keywords in *M_predicate* are then matched between every element in the database and then for every keyword $k$ in the code repository $P$. The resulting matches in the source code are stored in a *M_document* table. Table *M_predicate* is pruned to only hold those keywords that are represented in the matches. This matching is performed through database extensibility features, such as using user-defined functions (UDFs), in order to keep in-

main memory all of the data structures used to obtain an approximate distance (using edit distance) for every concept and every given string coming from the database or collection of source code files. This matching step is the bottleneck of the keyword matching. However, to ease the processing, the matching is performed in batches. In other words, a set of keywords $k$ mapped from $S$ is scanned simultaneously to avoid multiple passes over the $P\_document$ table and reduce I/Os.

Tables $S\_database$, $M\_document$, and $M\_predicate$ (summary tables) are created to avoid repeated searches on a particular data source and they include the location of the concept in the database and in the source repository. By using these summary tables, it is possible to obtain the matches (stored as a view called $M\_table$) by joining these three tables, in where there is a hash join on the matching keywords between $M\_predicate$ and $S\_database$, and a hash join on the id of $v$ present in $M\_predicate$ and $S\_database$.

This query is quite efficient due to the keyword indexes. An example of the resulting matches is shown in Table 1. Furthermore, the data type is considered irrelevant when matching metadata, but this remains something to be explored in the future.

The complexity of the integration is given by the number of unique keywords to search and the number of files in $P$. Therefore, the integration is on the order of $O(rn)$, where $r$ is the number of unique keywords and $n$ is the number of files.

## 3.2 Querying

The exploration is obtained only by analyzing the summarization tables obtained in the previous phase (mainly the approximate matches in the $M\_table$ view). Analytical queries are obtained mostly from the predicate table and the approximate match table as shown in Equation 1. These analytical queries are efficient one-pass aggregations in SQL using functions such as SUM, COUNT, MAX and MIN.

$$\pi_F(\sigma_Q(M\_predicate) \bowtie (M\_document)) \quad (1)$$

These queries are able to answer questions such as the number of matches that are associated with a particular $v$, $T$ or $a$, as well as which matches are present if the aggregation ($\pi_F$) is removed.

Furthermore, we are able to take these aggregations one step further and compute OLAP cubes from these matches by generating aggregations with different dimensions that include all the granularity levels inside the DBMS and within the source code. An OLAP query that exploits the summarization tables based on the CUBE operator from SQL SERVER is shown in Figure 4.

This query can answer complex analytical queries such as "Which column $a$ has the highest number of dependencies associated with $v$?"

Finally, the most complex searches are those that require following the dependencies. These searches answer queries such "What are all the methods associated with a particular column?" In order to find these dependency graphs $G$, we propose the following algorithm: The computation of the graph in $S$ is performed efficiently in main memory due to the reduced number of elements. However, computing the resulting graph in $P$ cannot be performed in main memory. The algorithm for computing graphs in $P$ is based on filtering the approximate match table to find all matches associated to $Q$, and sorting them by their frequencies in ascending order (least frequent first). Then, each of these matches is explored in a breadth-first-search fashion by joining the the $M\_document$ table filtered using $M\_predicate$ with the transition tables $P\_mcall$ and $P\_ccall$ (each explored path is maintained as a tuple in a temporary table). If a valid graph is found, the result is returned to the user. The exploration continues in a recursive manner by joining the resulting table of the previous iteration with the transition tables (those tables that contain the relationships between the methods and classes). The exploration will halt when a certain number of steps has been reached or when certain time threshold has expired (the exploration can fall into infinite loops). The resulting graphs are returned to the user in ascending order based on the number of elements required to satisfy $Q$.

```
SELECT
     CASE
          WHEN GROUPING(a) = 1
          THEN 'ALL'
          ELSE a
     END a,
     CASE
          WHEN GROUPING(v) = 1
          THEN 'ALL'
          ELSE v
     END v,
     SUM(f) AS f
FROM M_TABLE GROUP BY a, v
WITH CUBE;
```

**Figure 4: OLAP cube query.**

# 4. EXPERIMENTS

QDPC is a standalone system developed entirely in C# as two modules: a thin client that uses ODBC to connect to the DBMS and a set of UDFs and Store Procedures that perform the integration and searches (see Figure 2). Experiments were run on an Intel Xeon E3110 server at 3.00 GHz with 750 GB of hard drive and 4 GB of RAM. The server was running an instance of SQL SERVER 2005. All the experiments are the result of an average of 30 runs unless otherwise specified. The times are presented in seconds.

**Table 2: Programs.**

| Description | WP | SE | PJ1 | PJ2 |
|---|---|---|---|---|
| Num. Files | 52 | 44 | 119 | 316 |
| Num. Classes | 52 | 0 | 158 | 460 |
| Avg. File Size (KB) | 1409 | 4787 | 12388 | 11474 |
| Lines of Code (LOC) | 5488 | 5463 | 28180 | 70815 |
| Avg. Num. Variables | 8 | 29 | 19 | 35 |
| Avg. SQL queries | 0 | 1 | 0 | 5 |
| OLTP | N | N | Y | Y |

## 4.1 Programs and Schemas

QDPC was tested using four different source code repositories and their corresponding databases' schemas. The programs and schemas are a program used for capturing information to a database of water quality of wells in the

**Table 3: Schemas.**

| Description | WP | SE | PJ1 | PJ2 |
|---|---|---|---|---|
| Num. Tables | 52 | 25 | 21 | 95 |
| Avg. No. Columns | 7 | 4 | 6 | 5 |
| Max No. Columns | 110 | 12 | 15 | 29 |
| Min No. Columns | 1 | 2 | 2 | 2 |
| Processing Type | OLAP | OLTP | OLTP | OLTP |

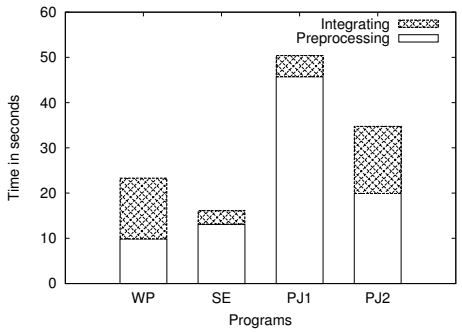



**Figure 5: Preprocessing and Integration.**

State of Texas (WP), the Sphider open source search engine (SE) program, and a couple of management systems (PJ1 and PJ2). Table 2 and Table 3 contain the details of each repository and schema.

## 4.2 Integration

The experiments in Figure 5 show the performance results of preprocessing the source code repositories, obtaining the approximate matches (allowing a proportional edit distance of 10%). Figure 5 shows that looking for the approximate matches and creating the summarization tables uses only a small portion of the time compared to the source code analysis. Despite this larger source code analysis task, the entire integration phase takes less than 1 minute in all the source code repositories.

In Table 4 and Figure 6, we focus on analyzing the effect of the approximate matches and the performance of such exploration. Table 4 shows that even though WP contains the fewest LOC, it has the highest number of unique keywords to search. This table shows that the value of $\beta$ is critical to finding good matches (and avoiding having meaningless matches). As is shown in this table, the values from $\beta$ should be between 0.10 and 0.25. However, when the value of $\beta$ exceeds that range, the number of approximate matches increases by a factor of 4x. Furthermore, Figure 6 shows that the performance of the algorithm is bounded by the number of unique matches. PJ2 has the highest performance time due to the number of LOC. However, WP ranks second due to the number of unique keywords to search, regardless of the size of the program. An interesting finding on this plot is that the performance of the algorithm is not affected by the selected $\beta$ because all of these computations are performed efficiently in batches stored within main memory.

Table 5 shows a breakdown of the performance of the integration phase. As expected, the bottleneck of the algorithm is in the computation of the approximate matches between the collections (Compute *M_document*).

**Table 4: Integration (Approximate Matches Found.) Performance time (in seconds) is shown for $\beta = 0.1$.**

| Program | L | $\beta = 0.10$ | $\beta = 0.25$ | $\beta = 0.50$ | Time |
|---|---|---|---|---|---|
| WP | 375 | 850 | 979 | 4008 | 11 |
| SE | 60 | 1438 | 1478 | 3619 | 4 |
| PJ1 | 88 | 4188 | 4304 | 13411 | 5 |
| PJ2 | 332 | 42217 | 42742 | 143928 | 16 |

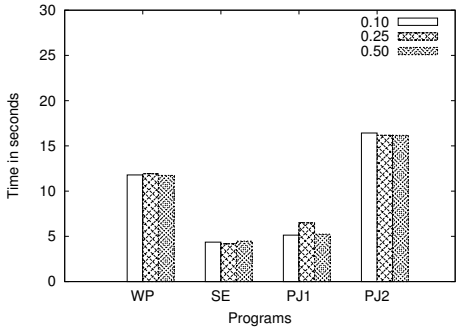



**Figure 6: Integration (Approximate Matches Performance.)**

## 4.3 Querying

The first type of querying to discuss here involves finding the number of matches associated with elements in either $P$ or $S$. Figure 7 shows the result of obtaining the SUM, MAX, MIN and COUNT aggregations in a single pass through the data. Figure 7 presents similar results for all the collections regardless of the original size. This is due to the fact that the summarization is quite efficient because the size of the input table is quite small and the aggregate functions are performed within main memory using a hash aggregate. The aggregation is not necessary as long as only these matches are sought, speeding up the query and resulting in a natural join.

A more complex analysis of aggregations can be computed by generating OLAP cubes. These queries answer analytical questions that focus on finding all the aggregations at different granularity levels. Some of the information obtained in the OLAP cube includes: the matches that are associated with all the methods and all the classes, or all the matches that are associated to all the elements in $P$. Figures 8 and 9 show the result of computing an entire OLAP lattice based on the summary tables. The first plot contains the computation of the lattice in different levels of the hierarchy. Therefore, the "P-type, S-type" cube shows the number of matches associated with a column, table, class, method, and so on. The "P-type, tablename" obtains an OLAP cube indicating the number of matches associated per $v$ with each $T$. The last aggregation generates the OLAP cube showing the number of matches associated with every class and $T$. This plot shows that regardless of the level, the OLAP cube is generated in less than 3 seconds in the smaller programs (see Figure 8) and in less than 17 seconds in the larger ones (see Figure 9). The highest times are associated with the programs with the largest number of approximate matches. In addition, Figure 8 and Figure 9 also show that computing

**Table 5: Integration (QDPC Profiling.)**

| Program | Compute M_predicate | Compute M_document |
|---|---|---|
| WP | 1 | 4 |
| SE | 1 | 2 |
| PJ1 | 1 | 4 |
| PJ2 | 1 | 14 |

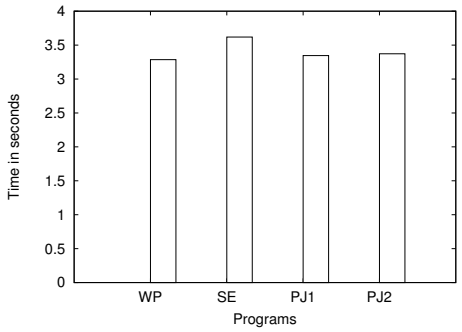



**Figure 7: Querying (Aggregations.)**

a cube in a lower level in the hierarchy is faster due to the early pruning of the match table allowing the exploitation of a hash aggregate.

Finally, all the elements associated with a particular keyword search are sought, due the dependencies between these. Notice that this type of querying is the most complex because it requires performing a breadth-first traversal of $P$ and $S$ in order to find all the $G$ that cover a $Q$. This search answers queries of the form "Is a particular method dependent on a particular column?". Figure 10 (which is the result of 30 random conjunctive queries with a maximum recursion depth of 5) shows that the exploration is equally efficient when generating the graphs $G$ by taking only a few seconds for the most time-consuming searches (2 and 3 keywords). When the number of keywords increases in $Q$, the performance time of the algorithm decreases due to the limited number of graphs that can satisfy $Q$, resulting in an early termination of the algorithm.

# 5. RELATED WORK

Control and data dependency analysis of programs have been explored for code maintainability, reverse engineering, and compilers. Despite this, the interaction between source code and a database's schema has rarely been explored. In [10] the authors propose the DB-MAIN case tool. This tool automatically extracts data structures and control and data dependencies declared in the source code (including all the explicit and implicit structures and constraints). The result of this tool is a conceptual representation of the data structures and the relationships between them. DB-MAIN proposes three techniques for capturing the dependencies in a program which are based on applying heuristics after a pattern-matching analysis. Unlike our approach, which is based on finding approximate matches, allowing us the flexibility of analyzing a larger variety of languages and pieces of code, DB-MAIN supports only a few languages (such as COBOL). Furthermore, we rely on a DBMS to perform the extraction and exploration of the matches. In the keyword search domain several algorithms have been proposed to manage efficient searches [5]. From this pool of algorithms, the closest similarity is found to be in the DBXplorer system [1], BANKS [4] and DISCOVER [11]. Our graph algorithm is partially based on the DISCOVER algorithm in the exploration style. However, in DBXplorer, DISCOVER and BANKS, the challenge is to find those tables and attributes that are related to PK/FK constraints. This differs from our algorithms, which focus on exploring the database's schema and not the data within. Finally, `QDPC` is the result of ongoing research in the field of data integration and joint exploration between structured and unstructured sources. This approach was originally implemented to integrate semistructured data with structured data, as presented in [13], in which the basic notion of a "link" was introduced. In addition, several algorithms for finding those links were introduced. Later on, we extended these ideas to perform efficient approximate keyword matching inside the DBMS [8]. Finally, in this paper, the ideas presented in [12] for preparing a data set for data mining and the ideas in [7] for OLAP exploration are extended to allow complex analysis of the resulting matches.

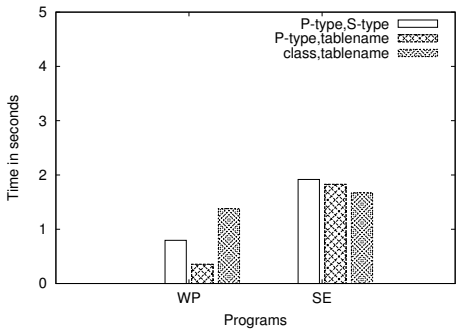



**Figure 8: Querying (OLAP Cube for small projects.)**

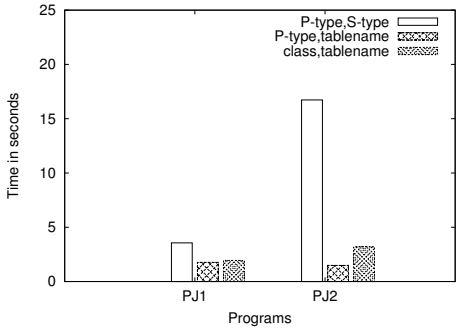



**Figure 9: Querying (OLAP Cube for large projects.)**

# 6. CONCLUSIONS

This paper presents a novel system that allows flexible querying of a source code repository and the schema of a database. In order to do so, our approach relies on an efficient integration phase that summarizes these dependencies and stores them in the database management system. The keyword searches in our approach are performed over these

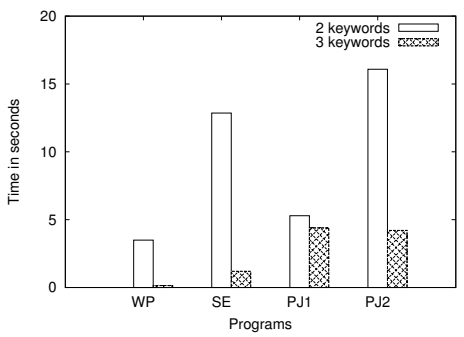



**Figure 10: Querying (Graph Searches.)**

summary structures that allow all the graphs that cover the keywords given by the user to be returned. Additional searches based on aggregation and OLAP cubes can be generated using these summary tables to answer complex analytical queries. Our experiments also showed that integration of several programs varying in the levels of dependency with the database's schema can be obtained quite efficiently (in less than a minute in all the cases). Furthermore, the scalability of the algorithm is observed to be bounded by the number of keywords to be searched for. It was also shown that searching for dependency graphs that cover the keywords of a query can also be obtained in less than 20 seconds in all the cases tested. Moreover, the algorithm also prunes all the undesired matches early (if there are not matches covering all the keywords), resulting in a faster evaluation when no graphs are found. The integration obtained with our system also allowed an efficient evaluation of aggregate functions and an efficient construction of OLAP cubes, which allows such complex queries as, "What are the methods that have the highest number of associated dependencies?" to be answered. Finally, it was shown that a database management system (DBMS) is a good candidate for managing this type of keyword search in the DBMS (the amount of keywords is much less that in the document domain), allowing the integration and exploration of source code.

Future work is required to improve the integration and exploration of source code repositories and a database's schema. A richer analysis of the semantics of the source code (e.g. considering the type of the data) needs to be incorporated. Moreover, a way to deal with ambiguity between the source code repository and the schema (e.g. "id" may be a column of several tables) needs to be found. The representation of complex relationships and interactions between multiple programs and databases remains as an open question. Furthermore, the usage of our match representation to build data mining models (e.g. classification or clustering) to explore the characteristics of the programs, in order to answer more complex questions, needs to be explored. Finally, the authors believe that it is possible to extend and generalize our proposal to jointly query larger semi-structured and structured sources in other domains, as well as to use a DBMS as a backbone for such processing and exploration.

## Acknowledgments

This work was partially supported by National Science Foundation grant IIS 0914861.

## 7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. ICDE*, pages 5–16, 2002.

[2] T.M. Austin and G.S. Sohi. Dynamic dependency analysis of ordinary programs. *Proc. of ACM SIGARCH Computer Architecture News*, 20(2):342–351, 1992.

[3] Z. Bellahsene, A. Bonifati, and E. Rahm. *Schema matching and mapping*. Springer-Verlag, 2011.

[4] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Procs. of ICDE*, pages 431–440, 2002.

[5] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword search on structured and semi-structured data. In *Proc. of ACM SIGMOD*, pages 1005–1010, 2009.

[6] A. Cleve, J. Henrard, and J.L. Hainaut. Data reverse engineering using system dependency graphs. In *Proc. of IEEE Conference on Reverse Engineering*, pages 157–166, 2006.

[7] C. Garcia-Alvarado, Z. Chen, and C. Ordonez. OLAP-based query recommendation. In *Proc. of ACM CIKM*, pages 1353–1356, 2010.

[8] C. Garcia-Alvarado and C. Ordonez. Keyword Search Across Databases and Documents. In *Proc. ACM SIGMOD KEYS Workshop*, 2010.

[9] M. Harman. Why source code analysis and manipulation will always be important. In *Proc. of IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 7–19, 2010.

[10] J. Henrard and J.L. Hainaut. Data dependency elicitation in database reverse engineering. In *Proc of IEEE European Conference on Software Maintenance and Reengineering*, pages 11–19, 2001.

[11] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[12] C. Ordonez. Data set preprocessing and transformation in a database system. *Intelligent Data Analysis (IDA)*, 15(4), 2011.

[13] C. Ordonez, Z. Chen, and J. García-García. Metadata management for federated databases. In *ACM CIMS Workshop*, pages 31–38, 2007.

[14] J. Tarhio and E. Ukkonen. Boyer-Moore approach to approximate string matching. *SWAT*, pages 348–359, 1990.

[15] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proc. of ACM ICSE*, pages 531–540, 2008.