Static Integration of SQL Queries in C++ Programs

Maciej Sysak, Bartosz Zieliński, Piotr Kruszyński, Ścibór Sobieski, and Paweł Maślanka

Department of Computer Science,
Faculty of Physics and Applied Informatics,
University of Łódź,
ul. Pomorska nr 149/153, 90-236 Łódź, Poland
{maciej.sysak,bzielinski,piotr,scibor,pmaslan}@uni.lodz.pl

Abstract. Contemporary frameworks offer essentially two methods of accessing data in relational databases. The one using plain SQL requires writing a lot of boilerplate code and storing the SQL in a string, which is error prone and denies the benefits of static query analysis. The other one enforces the use of an additional (usually object oriented) abstraction layer which incurs additional runtime costs and hinders the use of advanced SQL capabilities. In this paper we present a working implementation of a radically different approach. Our tool uses the database engine to analyze the native SQL queries prepared by the user, and generates all the necessary classes representing query responses, single result rows and database connections. The use of native queries allows to utilize advanced and highly optimized SQL features. On the other hand, the use of the generated classes ensures that data access happens in a statically checked, type-safe way.

1 Introduction

There are well known difficulties in accessing data stored in relational databases from application programs, especially if good object orientation is required:

1. A query is formulated in SQL, which is stored in the program code as string. Such strings are opaque to the compiler, which defers the error discovery until the query is actually sent to database during program execution. In case of C/C++ this situation is further aggravated by the low quality of runtime exception mechanisms. Also, writing the query as a string requires the programmer to properly escape all the special characters which might appear in the query (such as quotes). This garbles the query string stored in the program code and increases the error rate even more.
2. Iteration through the query result and binding prepared statements parameters requires a lot of unpleasant boilerplate code which is not type-safe — the number of columns in the result set, their types and names cannot be

checked by the compiler. It is also the task of a programmer to perform all the necessary type conversions between SQL and host language types.

A number of solutions is known and used in practice. Those fond of SQL can use the embedded SQL precompilers supplied for all major databases and languages. Unfortunately this solution has some disadvantages. For one thing, it mixes the SQL code with host language code and some auxiliary glue syntax. This confuses the syntax coloring tools — a seemingly minor inconvenience, which nevertheless might negatively influence the programmer's productivity. Also, it requires the coder to learn the aforementioned glue syntax and the actual implementations do not handle the type conversions as smoothly as one might expect. Finally, the precompilation stage introduces issues with some compilers.

A more common approach is to use ORM frameworks such as *Hibernate* or *Java Persistence Api*. The obvious advantages are that they are fully object-oriented. In particular, queries return objects, which can be made persistent in order to simplify the data modification management in program code. The frameworks often decrease the amount of boilerplate code development by advanced code generation features which makes the programmer task less error prone. As an additional boon the use of their own object oriented query language (such as HQL or JPQL) permits developing database vendor agnostic code. Additionally, many of the ORM frameworks (like JPA or Hibernate) allow the transparent data caching where the updates and inserts can be collected locally and actually sent to database as the single batch at the end of transaction. Similarly, the queries can first examine the cache for the presence of requested data.

Unfortunately the ORM frameworks have also well known drawbacks. A common and not entirely unjustified charge against ORM's is that the quality of generated SQL statements sent to database is very low and it is hard to get the performance right (see e.g. [17]). Less obviously, the same features mentioned above as beneficial can also be considered as a downsides from another point of view. Special query language isolates the developer from the particulars of the given SQL dialect so much that it precludes the use of advantageous features of the dialect. Moreover, some more advanced elements of SQL standard might not be implemented in the custom query language of the framework (consider e.g. window clauses of analytic functions). Finally, the developer is required to learn one more query language only superficially resembling SQL. Let us add that the OQL (HQL, JPQL, etc.) queries are passed to the library functions as strings, which, just like in case of SQL queries, delays the query syntax and semantics checking to the time of program execution. This is why many ORM frameworks include query building DSL's (Domain Specific Languages) utilizing the host language syntax — well known examples include LINQ and Java EE Criteria API. This takes care of compile time syntax checking but still leaves semantics checking to the run time.

More generally, ORM's often impose an active, object oriented view of the data in which objects representing entities manage their own updates leading to an impedance mismatch with the relational model ([13], see eg. [18] for a mathematical treatment of this mismatch). This object oriented view, despite the claims to the contrary [9] the authors (like some others [8]) do not perceive as the most natural way of conceptualizing the database application.

Another fundamental flaw of most of the existing ORM frameworks is that they perform during runtime many of the activities, such as translating OQL to SQL, which can be done during compilation as well. The impact on the performance is hard to assess, but might be significant in some applications. More important, however, is the loss of the benefits of static code analysis.

The dbQcc framework presented in this paper takes a completely different approach. We start with the plain, native SQL query as the source of all needed information. The external tool we created uses the analyzed query to generate classes representing the result row, and the query response. The application code uses the generated classes together with the featured database connection library to access data in a statically checked, type-safe way. The only query used in program runtime is the native one explicitly specified by the programmer in the beginning — we do not modify the query. Note that this is the opposite approach to the one taken by the ORM frameworks, which produces SQL queries from the annotated class definitions. Also it is worthy to emphasize that the class generating tool checks both the syntactic and semantic correctness of the query as well. Hence the successful class creation guarantees the lack of query syntax and data model mismatch errors in the runtime.

1.1 Comparison with Previous Work

Most of the existing approaches to supporting compile time verification of correctness of SQL statements involves introducing special syntax (often an internal DSL [10]) instead of using the native SQL like in our approach — see e.g. [11] for a native C++ example making use of template metaprogramming. An interesting exception is [12] in which a verifier of dynamically constructed SQL is presented. Another example of internal DSL is [14] in which a C# library is introduced which allows to construct SQL strings correct by construction (similarly as XML libraries construct correct XML files). Both ([11] and [14]) utilize an external tool, which examines the database schema and generates the appropriate access classes from table metadata. Note that in our approach we generate the access classes for each statement, rather than for each table. Also note that unlike our framework, which is meant to support the execution of verified queries, the tool presented in [12] only verifies the SQL statements, and does not include any provisions for type safe interface between SQL and the host program. Similarly, neither of the frameworks [11] or [14] supports actual execution of the queries generated.

The ORM frameworks utilize their own query languages for more complicated tasks. This introduces the same problems as dynamic SQL statements. Safe *Query Objects* framework [7] allows to construct compile time checked JDOQL queries (for *Java Data Objects* [1]). The framework is interesting, because unlike e.g., JDO's own JDO Typesafe, it allows to specify filter conditions not with special objects which represent them, but with native Java boolean expressions. The actual queries are generated by the external tool from the bytecode.

A somewhat different approach to generation of SQL queries correct by construction was taken by the authors of [16], in which the SQL queries are created from the specification in term rewriting system [6]. The language of specification includes a base sublanguage, which is made to resemble SQL — modulo some quirks imposed by the Maude's own syntax. The system allows to extend the SQL-like base with the ability to factorize common SQL fragments or to introduce some higher level features like named joins. In effect it can be seen as an SQL metaprogramming system. The framework outputs actual vendor specific queries, which can then be fed to a system like the one developed in this paper.

A system which partially inspired our framework and the approach of which bears the most similarity to ours is *Web4j* Java Web Application Framework [5]. Web4j uses a custom file format containing specially tagged SQL statements. At runtime the statements can be fetched by identifier and then processed, after supplying, if applicable, the parameter values. The iteration through the query result is enabled through the method which accepts the list of arguments consisting of the `SomeType.class` object of the desired class of rows, the identifier of the query, and the query parameters if any. The method returns a list of `SomeType` objects. The object construction process utilizes reflection to determine which constructor to use, matching the constructor with the same number of arguments that there are columns in the rows returned by the query. Note that, while the framework allows, as an option, to check the validity of the query at application startup by preparing the query (for the databases that support it), but, similarly as the matching of row columns to constructor arguments, it happens during runtime, and does not, unlike our solution, offer the compile time type safety for the queries.

When speaking about SQL metaprogramming it is worthwhile to mention *Metagen* [15] — a tool for generating database schemas from vendor independent descriptions in a special language.

Let us note that while object relational mappers isolate the user from some vendor specific database capabilities, presenting instead the generic interface, the support for advanced features in this generic interface is already considerable and is rising fast, see eg. [19], which enriches Hibernates HQL with recursive queries.

Finally, note that our framework strongly supports implementing the domain logic using the transaction script pattern [9], whereas most of the higher level, object oriented frameworks are geared towards either the domain model [9] — e.g., the Hibernate framework, or the table model [9] — e.g. ADO Net.

2 System Architecture and Internals

This section provides a detailed account of our system. The data flow diagram in Figure 1 presents the overview of the system architecture. In particular:

– The sql statements used in the application utilizing the DbQcc framework ought to be placed in the separate uqf file(s) (for each database used) which are essentially SQL files with some metadata placed in specially formatted comments. Subsection 2.1 contains a detailed description of the file format.

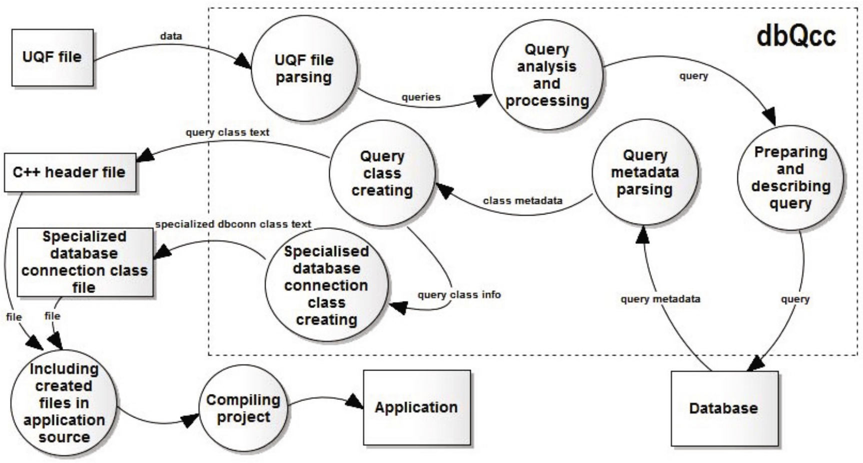
dbQcc
UQF file
data
UQF file parsing
queries
Query analysis and processing
query
Preparing and describing query
query
Database
query metadata
Query metadata parsing
class metadata
Query class creating
query class text
C++ header file
query class info
Specialised database connection class creating
specialized dbconn class text
Specialized database connection class file
file
file
Including created files in application source
Compiling project
Application

Fig. 1. Data flow in the system

– We use a C++ database access abstraction layer responsible for database connection management and the actual query execution at runtime. This abstraction level is not meant to be visible to the user but to serve as an implementation layer for the classes actually used by the programmer. A transparent support for standard database operations in different database engines is provided through the driver mechanism. Currently, only the driver for the PostgreSQL database (based on *libpq* [2]) is available. In order to furnish the required functionalities in an efficient way we use some of the new features introduced in C++11 like shared pointers and move semantics.
– Moving data from the database into the application necessitates converting data from the representation used by the DBMS to the native C++ types. In Subsection 2.2 we describe the mechanisms supporting the conversions.
– The uqf file is processed by the separate tool which can be integrated into compilation toolchain, and which generates for each statement classes representing: query record and query response. Additionally, for each uqf file the database access class is generated. The Subsection 2.3 contains the account of various stages of processing of user queries extracted from uqf file(s).

2.1 User Query File

Database queries used by the application ought to be placed in the separate `.sql` file (see Figure 2), which we will refer to as an `uqf` (user query) file, collecting all queries referring to the particular database. The file employs the special key-value format of comments, which permits adding some auxiliary information necessary for the correct execution of the class generating tool. The project may contain many `uqf` files, each one associated with a different database.

```
/* DBEngine = PostgreSQL
   DBName = dbqcc
   DBHost = ***.uni.lodz.pl
   DBUser = dbqcc_tester
   DBPassword = SQLorNothing
   DBFileName = Blog */
-- PREPARABLE_SELECT = ClientRanking
SELECT c.email, s.name, r.client_rank
FROM (
   SELECT pcnt.*, rank() OVER(
       PARTITION BY pcnt.section_id ORDER BY pcnt.cnt DESC
   ) AS client_rank
   FROM (SELECT count(*) AS cnt, p1.owner_id, p1.section_id
         FROM page p1 GROUP BY p1.owner_id, p1.section_id) pcnt
) r JOIN client c ON (r.owner_id = c.id)
          JOIN section s ON (r.section_id = s.id)
WHERE r.client_rank <= $1 ORDER BY s.name, r.client_rank
```

Fig. 2. User query SQL file sample containing parametrized query

The uqf file starts with the information about the database engine vendor which is used by the class generator to choose the appropriate driver (if available). Next, the file contains the connection data — it is utilized by the generator to connect to the database server during query analysis phase. For security reasons one may omit some of the authorization information — the missing data can be supplied at the generator's execution time. In addition, the authorization data provide default values in the generated database connection class.

The last section of uqf file contains a sequence of SQL statements present in the application (intended to be executed in the database the uqf file is associated with). Queries are to be separated with empty lines. Each statement should start with a specially formatted SQL comment specifying exact statement type (PREPARABLE_SELECT in the query in Figure 2) and assigning a unique (within the file) statement identifier (ClientRanking in the query in Figure 2), which must be a valid C++ type name. This identifier will be utilized as the statement class name. The statement type declares whether it is a SELECT query, DML statement or a stored subprogram as well as whether the statement should be prepared prior to the first execution, and thus stored in prepared form in the database server for the duration of the session. Preparing statements may bring significant performance benefits, specifically for frequently executed and particularly complex statements [4] — their parsing, rewriting and creation of their execution plans happen only once, when they are prepared.

After the statement header follows the statement in the native form. There is no need to escape special characters, the statement will be converted to a valid C++ string placed in the query class body during class generation. The query may be parametrized, where the parameters are denoted by consecutive numbers prefixed with the dollar character (see Figure 2). Note that the uqf file is a valid

```
template <typename T> class ValueConverter{};

template <> class ValueConverter<uint32_t> {
public:
   static bool fromDB(char * rawValue, uint32_t & value) {
       uint32_t tmp;
       memcpy(&tmp, rawValue, sizeof(tmp));
       value = ntohl(tmp);
       return true;
   }
   static bool toDB(const uint32_t & value, std::vector<char> & rawValue,
                    int & rawValueFormat) {
       rawValue.resize(sizeof(uint32_t));
       uint32_t tmp = htonl(value);
       memcpy(&rawValue[0], &tmp, rawValue.size());
       rawValueFormat = DBManagerBase::BINARY_FORMAT;
       return true;
   }
};
```

Fig. 3. Value converters for uint32_t (PostgreSQL oid)

SQL file and any editor with SQL syntax coloring capabilities can be used for simple editing.

2.2 Data Conversion Mechanism

Data access layers for C++ of some of the databases (PostgreSQL in particular) allow to exchange data both in binary and text formats. Using the text format is not applicable for all types and clearly less efficient (both computationally and in terms of compactness of representation) than binary format. Data conversion to C++ types is necessary for performing operations on data regardless of format used for communication with database. To facilitate data conversion we provide a mechanism based on C++ templates. The `ValueConverter` class template (see Figure 3) has its specializations for all supported data types. For an unsupported data type (one not having a template specialization) compilation time error occurs. Note that the `libpq` library representations of database integer types are not completely determined, e.g, oid type employs C/C++ `unsigned int` which has no strict definition of size and endianness in language standards. Therefore, our conversion mechanism provides converters (template specializations) for all C++11 standard fixed size integer types (e.g., `uint32_t`) and the compiler matches the specialization for an appropriate type (e.g, `uint32_t` for `unsigned int` on the majority of 32-bit platforms). The endianness problem is solved in a platform-specific way, taking into account that PostgreSQL DBMS provides binary data (for integer representations) in network byte order.

2.3 Query Analysis and Processing

The operation of the query analyzer module is based on using the existing database to verify both the query syntax and the validity of references to schema objects, and, when the query is valid, to obtain the metadata associated with the query. The metadata is used later to generate appropriate classes representing the SQL statement, and, if applicable, the result rows. Hence, the statement analysis stage starts with the query analyzer module connecting with the database server on which the analyzed statements are supposed to be eventually executed.

The present system implementation provides a prototype plugin for the PostgreSQL database engine based on the native *libpq* [2] database access library for C which provides advanced functionalities to obtain rich metadata describing the SQL statement and its parameters. Query analyzing module operation, described in detail below, is largely depending on those *libpq* metadata capabilities.

```
class ClientRankingRecord: public QueryRecord {
public:
   DBValue<std::string> client_email, section_name;
   DBValue<long long int> client_rank;
   ClientRankingRecord(QueryResult * queryResult, int rowNumber) :
      QueryRecord(queryResult, rowNumber),
      client_email(queryResult, rowNumber, 0),
      section_name(queryResult, rowNumber, 1),
      client_rank(queryResult, rowNumber, 2) {}
};
```

Fig. 4. Generated query row class example

The SQL statement extracted from `uqf` file is not executed on the database server. Instead, it is sent to the database as a prepared statement. In the majority of relational database engines (and in particular in PostgreSQL) the prepared statement is parsed, and the query plan is generated (with some slots reserved for filling by parameters, if any) and stored in the database, ready for execution (perhaps multiple times) until explicitly closed. Most relevant for our purpose is that one does not need to execute the prepared statement nor supply the values of the parameters in order to be able to receive all the available statement metadata (in particular we can work with the database containing empty tables). After preparing the statement, such as the one depicted in Figure 2, it suffices to send the request *describe prepared*. In reply we receive the information about:

– names and types of columns of the result set (in case of queries). For columns defined by simple table column reference we also get the source table name,
– types of parameters, in case the statement was parametrized.

The query analyzer module uses this metadata together with the statement itself and the statement identifier extracted from the query header in the `uqf` file to

```
class ClientRanking: public QueryResult{
  static constexpr const char * queryName = "ClientRanking";
  static constexpr const char * query = "SELECT [...]";
  static bool prepared;
public:
  long long int param1;
  ClientRanking(DBManagerBase * dbManager,
                const long long int & _param1): param1(_param1){
     const char * paramValues[1];
     int paramLengths[1]; int paramFormats[1];
     std::vector<char> param1RawValue;
     ValueConverter<long long int>::toDB(
           param1, param1RawValue, paramFormats[0]);
     paramValues[0] = &param1RawValue[0];
     paramLengths[0] = param1RawValue.size();
     if(!prepared){
       auto result = dbManager->prepareStatement(
              queryName,query,1,nullptr);
       if(result->bad()) throw std::runtime_error(dbManager->getError());
       prepared = true;
     }
     dbResult = dbManager->executePreparedStatement(
          queryName,1,paramValues,paramLengths,paramFormats,
          DBManagerBase::BINARY_FORMAT);
     if(dbResult->bad()) throw std::runtime_error(dbManager->getError());
  }
  std::shared_ptr<ClientRankingRecord> operator[](int rowNumber){
     if(rowNumber >= dbResult->getRowsNumber()) return nullptr;
     return std::make_shared<ClientRankingRecord>(this,rowNumber);
  }//[...]
};
```

Fig. 5. Generated query response class example

assemble, using the class builder internal tool, header files holding definitions of the classes associated with the statement, such as:

– (In case of SELECT statement) the class representing a single result set row, with a field (and possibly accessor methods) of appropriate type for each result set column (e.g., see Figure 4). The field and accessor names are based on column names, qualified with table name, if applicable (with some necessary conversions, like substituting underscores for spaces). Fields are constructed using the `DBValue` class template which uses the mechanism described in Subsection 2.2 to convert data received from the database. Because `DBValue<T>` overloads the conversion operator to type `T` it follows that it preserves the `const T&` semantics. The class provides also conversion operators supporting structural type equivalence of database records with the same types of columns (c.f. [11]). In our approach the type of a query result set row class identifies the query, but the conversion operators permit

transparent combination or substitution of rows from different queries, provided that they have the same number and types of columns.
- The class representing the query result, which also encapsulates the statement itself (e.g., see Figure 5). This class also provides the functionality to execute the statement (and supply the arguments if applicable), and in the case of the SELECT statement also to iterate through the result rows. The statement, if declared as preparable, is prepared during the construction of the first object of a given class. In this case, executions refer to the parsed statement stored in the database server.

```
class DBManagerBlog : public DBManagerBase {
    DBManagerBlog(){};
public:
    typedef const std::string & csr;
    static std::shared_ptr<DBManagerBlog> create(csr host, csr dbName,
        csr user, csr password, csr errorMessage) {/* [...] */}
    // [...]
    std::shared_ptr<ClientRanking>
    SELECT_ClientRanking(const long long & param1) {
        return std::make_shared<ClientRanking>(this, param1);
    }
};
```

Fig. 6. Generated database connection class

The last stage of query analysis module operation is the generation of specialized database connection classes (for each database server application connects to). The classes inherit from the generic database access driver for the particular DBMS. It extends the driver with methods for executing SQL statements and stored procedures specified in the processed `uqf` file (see, e.g., Figure 6).

After all the queries are analyzed and all necessary classes are generated, the query analyzer module cleans up the database, removing all prepared statements (in case of PostgreSQL it suffices to close the connection, as the prepared statements are associated with the session and die with it).

3 Simple Example of DbQcc Usage

The following code presents basic example usage of our framework:

```
std::string error;
auto dbManager = DBManagerBlog::create("***.uni.lodz.pl",
   "dbqcc", "dbqcc_tester", "SQLorNothing", error);
if(!dbManager) {throw std::runtime_error(error);}
auto ranking = dbManager->SELECT_ClientRanking(10);
for(auto i = 0; i < ranking->getRowCount(); i++) {
   auto row = ranking->getRow(i);
   // [...]
}
```

Classes generated by dbQcc provide simple API for programmer to establish database connection, execute queries described in user queries file and access response data. Our example schema may be treated as an extremely simplified blog's database. It consists of three tables: PAGE, CLIENT and SECTION connected with referential constraints. PAGE stores the title and content of blog pages. Each page is authored by the unique client and belongs to the unique section. In our sample UQF we placed the `ClientRanking` query (Figure 2) defined as preparable, containing one parameter. This select is supposed to return, for each section, n most prolific authors ranked with respect to the number of the authored pages, where n is passed as a parameter. Note that `ClientRanking`, in addition to being a complex query with joins and multiply nested subqueries, makes use of a window (or analytic) functions (the `rank()`), which, despite being in the standard, are not widely supported by ORM's.

We have performed tests to compare our framework with the directly used PostgreSQL API (libpq). The blog schema described above was filled with random data: 100000 rows in the CLIENT table, 200000 rows in the PAGE table, and, finally, 10 rows in the SECTION table. We prepared two equivalent implementations of the same application which executes the query from Figure 2 one hundred times with the sole parameter set to 7500 and collects the result rows, performing all the necessary data conversions. The two implementations, compiled with the same (default) optimization level, utilize, respectively:

– The dbQcc framework (**dbQcc**),
– Directly used PostgreSQL API (**libpq**).

Both implementations utilize statement preparation. Note that statement is prepared only once. The durations of each query execution and data conversion were measured separately, and for each implementation averages and standard deviations were computed. The results are presented in Table 1. Note that the differences in times between query executions in both implementations are less than the standard deviation, and hence it follows that our framework does not add any significant overhead.

Table 1. Execution times for the two alternative implementations of the same application executing query from Figure 2

	dbQcc	**libpq**
Query execution	6.8 ± 0.2	6.7 ± 0.2
Row parsing	0.058 ± 0.007	0.005 ± 0.002

4 Conclusion

In the paper we presented an alternative approach to programming database applications, which is more natural and effective for SQL oriented people (and, potentially, also leads to a better performance). We developed a working tool generating C++ classes from plain native SQL statements. Those generated classes allow us to execute the statements in the application program and to iterate in a semantically correct way through the result sets (in case the statement was a query). Thus, we effectively couple native SQL with a C++ code in a way which is statically checked for correctness, excluding, in particular, the possibility of runtime type mismatch errors. Unlike in the case of embedded SQL, the queries and application program is kept separate, which simplifies application development by programmers with specialized skills — the C++ programmer has no need to see the SQL queries developed by database wizards, and conversely, database specialists are happily separated from C++.

Using plain SQL in its native form makes it possible to write specialized and highly optimized queries even with DBMS vendor specific features. For a simple example see Section 3, which presents a non trivial SQL query, hardly supported by typical approaches, but potentially useful and non-artificial.

Because we utilize mechanisms such as a lazy evaluation of values, the current implementation of dbQcc is not thread safe. This means that the use of generated code in threaded application requires protecting all database operations with a mutex associated with a given database session. Presently we work on effective synchronization of generated code using atomic types and non-blocking algorithms.

The present implementation supports only PostgreSQL, and it will be worthwhile to develop drivers for some other database engines. Despite a well-layered architecture of our system, it might not be entirely trivial, as we assume the availability of certain PostgreSQL features which are not required by the standard and might not be available from other vendors.

Moreover, using the database server to parse SQL statements, while not without merit (it is simple to implement and, if the database is the same as the target one, we can be sure that the query will be accepted also during the application runtime), it has some downsides as well. For one thing, sometimes the database might not exist (even in the form of a schema with empty tables) at program design time. Arguably, it might be better (and more elegant) to parse the statement directly in the tool with the option of supplying the schema description in the specialized format. Moreover, once the tool does the parsing, and hence understands the SQL statement syntax, new possibilities appear, e.g., of extending the SQL syntax, introducing, in particular, special annotations for automated BLOB conversion into specified class objects during query execution.

Finally, as we can create connection objects to many databases and database servers in the same application, it is only natural for the need for the distributed commit mechanisms to appear. Therefore the support for two-phase commit is currently under active development.

Acknowledgements. We would like to thank the reviewers for their helpful remarks and to Łukasz Krawczyk for his help in implementing the tests.

References

1. Java Data Objects, `http://db.apache.org/jdo/`
2. Libpq — C Library, `http://www.postgresql.org/docs/9.3/static/libpq.html`, chapter in [3]

3. PostgreSQL 9.3.2 Documentation, `http://www.postgresql.org/docs/9.3`
4. PREPARE command,
`http://www.postgresql.org/docs/9.3/static/sql-prepare.html`
5. Web4j java web application framework,
`http://www.web4j.com/Java_Web_Application_Framework.jsp`
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
7. Cook, W., Rai, S.: Safe query objects: statically typed objects as remotely executable queries. In: Proceedings of 27th International Conference on Software Engineering, ICSE 2005, pp. 97–106 (May 2005)
8. Date, C.: An Introduction to Database Systems. Addison-Wesley (2003)
9. Fowler, M.: Patterns of Enterprise Application Architecture. A Martin Fowler signature book. Addison-Wesley (2003)
10. Fowler, M.: Domain-Specific Languages. Addison-Wesley Signature Series (Fowler). Pearson Education (2010)
11. Gil, J.Y., Lenz, K.: Simple and safe SQL queries with C++ templates. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE 2007, pp. 13–24. ACM, New York (2007)
12. Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. In: Proceedings of 26th International Conference on Software Engineering, ICSE 2004, pp. 645–654. IEEE (2004)
13. Maier, D.: Representing Database Programs As Objects. In: Advances in Database Programming Languages, pp. 377–386. ACM, New York (1990)
14. McClure, R., Kruger, I.: Sql dom: compile time checking of dynamic sql statements. In: Proceedings of 27th International Conference on Software Engineering, ICSE 2005, pp. 88–96 (May 2005)
15. Pustelnik, J., Sobieski, Ś.: Metagen — the text tool for generating sql database descriptions from ER diagrams (in polish). In: Bazy Danych - Modele, Technologie, Narzędzia, pp. 309–314. WKL Gliwice (2005)
16. Sobieski, S., Zieliński, B.: Using maude rewriting system to modularize and extend sql. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC 2013, pp. 853–858. ACM, New York (2013)
17. Wegrzynowicz, P.: Performance antipatterns of one to many association in hibernate. In: 2013 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 1475–1481 (September 2013)
18. Wiśniewski, P., Burzańska, M., Stencel, K.: The impedance mismatch in light of the unified state model. Fundamenta Informaticae 120(3), 359–374 (2012)
19. Wiśniewski, P., Szumowska, A., Burzańska, M., Boniewicz, A.: Hibernate the recursive queries - defining the recursive queries using hibernate orm. In: ADBIS (2), pp. 190–199 (2011)