

Generating Highly Customizable SQL Parsers

Sagar Sunkle, Martin Kuhlemann, Norbert Siegmund,
Marko Rosenmüller, Gunter Saake
School of Computer Science
University of Magdeburg
39106 Magdeburg, Germany
{ssunkle,mkuhlema,nsiegmun,rosenmue,saake}@ovgu.de

ABSTRACT

Database technology and the Structured Query Language (SQL) have grown enormously in recent years. Applications from different domains have different requirements for using database technology and SQL. The major problem of current standards of SQL is complexity and unmanageability. In this paper we present an approach based on software product line engineering which can be used to create customizable SQL parsers and consequently different SQL dialects. We give an overview of how SQL can be decomposed in terms of *features* and how different features can be composed to create tailor-made parsers for SQL.

General Terms

Design, Languages

Keywords

Tailor-made Data Management, Embedded Systems, Feature-oriented Programming

1. INTRODUCTION

Since its modest beginnings in the 70's, database technology has exploded into every area of computing. It is an integral part of any modern application where different kinds of data need to be stored and manipulated. All major database technology vendors have adopted a very general approach, combining functionality from diverse areas in one database product [8]. Likewise, *Structured Query Language (SQL)*, the basis for interaction between database technology and its user, has grown enormously in its size and complexity [6, 13]. Starting with the standard selection-projection-join queries and aggregation, SQL now contains a number of additional constructs pertaining to new areas of computing to which database technology has been introduced [8]. All major database vendors conform to ISO/ANSI SQL standards at differing levels, maintaining the syntax suitable for their products, thus further increasing the complexity of learning and using SQL. Requirements like performance, tuning, configurability, etc., differ from domain to domain, thus needing a different

treatment of database technology in different applications. Characteristics of SQL like data access, query execution speed, and memory requirements differ between application domains like data warehouses and embedded systems. Various researchers have shown the need for configurability in many areas of DBMS [6, 13, 17]. We therefore need the ability to select only the functionality we need in database products in general and SQL in particular.

Software product line engineering (SPLE) is a software engineering approach that considers aforementioned issues of products of similar kind made for a specific market segment and differing in features. A set of such products is called a *software product line (SPL)* [18]. In SPLE, products are identified in terms of features which are user-visible characteristics of a product [12, 9]. Decomposing SQL in terms of features, that can be composed to obtain different SQL dialects, can be beneficial and insightful not only in managing features of SQL itself but also in database technology of embedded and real time systems. A customizable SQL and a customizable database management system is a better option than using conventional database products for embedded systems. Embedded and real time systems have different characteristics and performance requirements than the general computing systems due to restricted hardware and frequent use of certain kinds of queries such as projection and aggregation [6]. Embedded systems like various hardware appliances, peer-to-peer, and stream based architectures for embedded devices use shared information resources and require declarative query processing for resource discovery, caching and archiving, etc. [13]. Query processing for sensor networks requires different semantics of queries as well as additional features than provided in SQL standards [14]. A feature decomposition of SQL can be used to create a 'scaled down' version of SQL appropriate for such applications, by establishing a product line architecture for SQL variants.

In this paper, we propose that SPL research is capable of providing answers to the problems of managing features in database products such as SQL engines.

We present an approach for a customizable SQL parser based on product line concepts and a decomposition of SQL into features. We show how different features of SQL are obtained and how these features can be composed to create different SQL parsers.

2. BACKGROUND

2.1 Structured Query Language

SQL is a database query language used for formulating statements that are processed by a database management system to create and maintain a database [21]. The most commonly used SQL query is the SELECT statement which can retrieve data from one or more tables in a database. This data can be restricted using conditional statements in the WHERE clause. SELECT can group related data using the GROUP BY clause and restrict the grouped data with the HAVING clause. It can order or sort the data based on different columns using the ORDER BY clause [15]. SQL contains many statements to create and manipulate database objects. Since its first standardization in 1986, more and more functionality is being included in SQL in each subsequent standard. The latest edition of the SQL standard, referred to as SQL:2003, supports diverse functionality such as call level interfacing, foreign-data wrappers, embedding SQL in Java, business intelligence and data warehousing functions, support for XML, new data types, etc.

The vast scope of SQL's functionality has led many researchers to advocate the usage of a 'scaled down' version of SQL, especially for embedded systems [6, 13, 8]. Embedded systems have many hardware limitations such as small RAM, small stable storage, and high data read/write ratio. Also the applications where embedded systems are used, e.g., healthcare and bank cash cards, need only a small set of queries like select, project, views, and aggregations. A standard called *Structured Card Query Language (SCQL)* by ISO considers inter-industry commands for use in smart cards [11] with restricted functionality of SQL. Some database systems and SQL engines, distinguished as 'tiny', have been proposed to address this issue, e.g., the TinyDB¹ database management system for sensor networks. TinyDB contains TinySQL language for querying sensor networks. TinySQL has a limited functionality compared to SQL such as single table in FROM clause, no column alias in SELECT clause, and sensor networks specific query constructs such as epoch duration and sample period clause.

While the standardization process shows how SQL has increased in size and complexity in terms of features provided, efforts for 'scaled down' versions indicate a need to control and manipulate features of SQL.

¹<http://telegraph.cs.berkeley.edu/tinydb/>

2.2 Software Product Line Engineering

SPLE aims at developing software applications by identifying and building reusable assets in the domain engineering and implementing mass customization in the application engineering [18]. The feature modeling activity applied to systems in a domain to *capture commonalities and variabilities* in terms of features is known as *feature-oriented decomposition*. Feature-oriented decomposition is carried out in the analysis phase of domain engineering. A feature is any end-user-visible, distinguishable and functional or non-functional characteristic of a concept that is relevant to some stakeholder [9, 10]. In modeling the features of SQL (specifically SQL:2003), we take the view of features as the end-user-visible and distinguishable characteristics of SQL. *Feature diagrams* [12, 9] are used to model features in a hierarchical manner as a tree. The root of the tree represents a concept and the child nodes represent features. The hierarchical structure of a feature diagram indicates that there is a parent child relationship between the feature nodes. A feature diagram contains various types of features such as *mandatory*, *optional*, *alternative* features, *AND* features, and *OR* features. A *feature instance* is a description of different feature combinations obtained by including the concept node of the feature diagram and traversing the diagram from the concept. Depending on the type of the feature node, the node becomes part of the instance description [9].

2.3 Feature-Oriented Programming

Feature-oriented programming (FOP) [19] is the *study of feature modularity* and how to use it in *program synthesis* [2]. It treats features as first-class entities in design and implementation of a product line. A complex program is obtained by adding features as incremental details to a simple program. The basic ideas of FOP were first implemented in GenVoca [4] and *Algebraic Hierarchical Equations for Application Design (AHEAD)* [5]. AHEAD uses the *Jak* language, which is a superset of the Java language, for feature-oriented programming. In AHEAD, a programming language and language extensions are defined in terms of a base grammar and extension grammars. Grammars specific to both the language and the language extensions are described using the *Bali* grammar specification language. Bali can be used to specify sub-grammars that can be shared and reused. As a grammar specification language, Bali allows specifying extra constructs for BNF specification such as labels to name the production rules in a grammar. Grammars written in Bali can be composed to specify language extensions and language combinations². A Bali grammar can import definitions for non-terminals from other grammars. Syntax extension and

²<http://www.cs.utexas.edu/users/schwartz/>

the corresponding semantic actions are implemented separately using the Javacc³ parser generator and the Jak language respectively. Features are treated as collaborations among classes and composed using *Jampack* and *Mixin* tools [5].

3. CUSTOMIZABILITY FOR SQL

Customizability for SQL means that only the needed functionality, such as only some specific SQL statement types, is present in the SQL engine. The concepts of features and product lines can be applied to the SQL language so that composition of different features leads to different parsers for SQL. We base our approach for creating a customizable parser for SQL:2003 on the idea of composing sub-grammars to add extra functionality to a language from the Bali approach as stated above. For a customizable SQL parser, we consider LL(k) grammars which are a type of context free grammars. Terminal symbols are the elementary symbols of the language defined by such a grammar while the nonterminal symbols are set of strings of terminals also known as syntactic variables [1]. Terminals and nonterminals are used in production rules of the grammar to define substitution sequences. From a features and feature diagrams perspective, the process of feature-oriented decomposition and feature implementation is realized in two broad stages [20]. In the first stage, the specification of SQL:2003 is modeled in terms of feature diagrams. Given the feature diagram of a particular construct of SQL:2003 that needs to be customized, we need to create the LL(k) grammar that captures the feature instance description. This ensures that the specific feature which we want to add to the original specification is included as well. In the second stage, having obtained LL(k) grammars for the base and extension features separately, we compose them to obtain the LL(k) grammar which contains the syntax for both the base and extension features. With a parser generator, we obtain a parser which can effectively parse the base as well as extension specification for a given SQL construct.

We add semantic actions to the parser code thus generated using Jak and other feature-oriented programming tools, effectively creating a SQL:2003 preprocessor. In the following sections we explain decomposition of SQL:2003 into features and composition of these features.

3.1 Decomposing SQL:2003

For the feature-oriented decomposition of SQL:2003, we use various SQL:2003 standards ISO/IEC 9075 - (n):2003 which define the SQL language. SQL Framework [16] and SQL Foundation [15] encompass the min-

imum requirements of the SQL. Other parts define extensions.

SQL statements are generally classified by their functionality as data definition language statements, data manipulation language statements, data control language statements, etc. [15]. Therefore, we arranged the top level features for SQL:2003 at different levels of granularity with the basic decomposition guided by the classification of SQL statements by function as found in [15]. The feature diagram for features of SQL:2003 is based on the BNF grammar specification of SQL:2003 and other information given in SQL Foundation.

We use the BNF grammar of SQL for constructing the feature diagrams based on the following assumptions:

- The complete SQL:2003 BNF grammar represents a product line, in which various sub-grammars represent features. Composing these features creates products of this product line, namely different variants of SQL:2003.
- A nonterminal may be considered as a feature only if the nonterminal clearly expresses an SQL construct. Mandatory nonterminals are represented as mandatory features. Optional nonterminals are represented as optional features.
- The choices in the production rule are represented as OR features.
- A terminal symbol is considered as a feature only if it represents a distinguishable characteristic of the feature under consideration apart from the syntax (e.g., DISTINCT and ALL keywords in a SELECT statement signify different features of the SELECT statement).

The grammar given in SQL Foundation is useful in understanding the overall structure of an SQL construct, or what different SQL constructs constitute a larger SQL construct. This approach may also be useful in general to carry out a feature decomposition of any programming language as the grammar establishes the basic building blocks of any programming language. The most coarse-grained decomposition is the decomposition of SQL:2003 into various constituent packages. We have chosen to further decompose SQL Foundation, since it contains the core of SQL:2003. Overall 40 feature diagrams are obtained for SQL Foundation with more than 500 features. Other extension packages of SQL:2003 can be similarly decomposed. Figures 1 and 2 show the features *Query Specification* (representing SELECT statement) and the feature *Table Expression* respectively.

To obtain the sub-grammars corresponding to features, we refer to the feature diagram for the given feature. Based on the feature diagram, we create LL(k) grammars for each feature in the feature instance description using the original SQL:2003 BNF specification for *Query Specification* (cf. Section 7.12 in SQL

³<https://javacc.dev.java.net/>

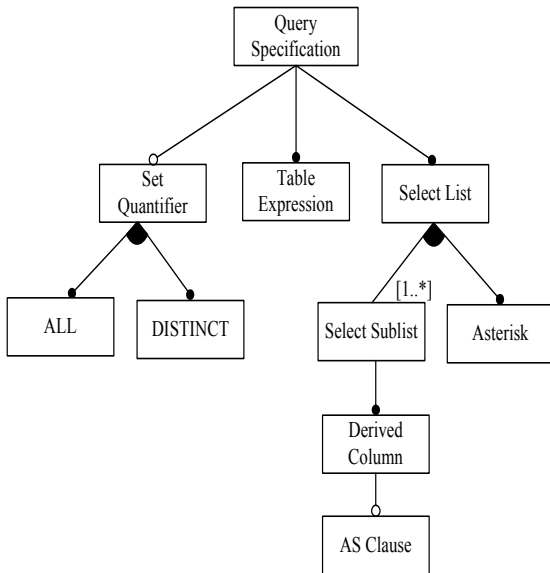


Figure 1: Query Specification Feature Diagram.

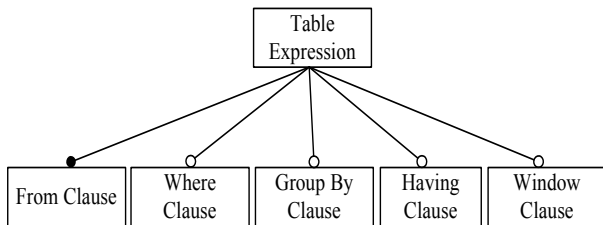


Figure 2: Table Expression Feature Diagram.

Foundation). These sub-grammars are used later during composition to obtain a grammar that can parse various SQL constructs represented by corresponding features. We represent a grammar and the tokens separately. Accordingly, for each sub-grammar we also create a file containing various tokens used in the grammar.

3.2 Composing SQL:2003 Features

During our work on a customizable parser for SQL:2003, we found that the tool *Balicomposer*, which is used for composing the Bali grammar rules, is restrictive in expressing the complex structure of SQL grammar rules. Bali uses its own notation for describing language and extension grammars which are converted to LL(k) grammars as required by the Javacc parser generator. We instead use the LL(k) grammars with additional options used by the ANTLR⁴ parser generator in our prototype. In order to create an SQL engine, we also need the feature-oriented programming capability already present in the form of Jak language and Bali related tools *Mixin* and *Jampack*.

We use the feature diagrams obtained in the decom-

position to create a feature model from which different features constituting the parser may be selected. Such a parser for SQL:2003 can selectively parse precisely those SQL:2003 statements which are represented as features in feature diagrams under consideration. We explain this with an example.

Suppose that we want to create a parser for the SELECT statement in SQL:2003 represented by the *Query Specification* feature (cf. Figure 1). Specifically we want to implement a feature instance description of {*Query Specification*, *Select List*, *Select Sublist* (with cardinality 1), *Table Expression*} with the *Table Expression* feature instance description, {*Table Expression*, *From Clause*, *Table Reference* (with cardinality 1)}. We would proceed as follows:

1. A feature tree of the SELECT statement presents various features of the statement to the user. Selection of different subfeatures of the SELECT statement is equivalent to creating a feature instance description.
2. To create a parser for these features, we make use of the sub-grammars and token files created during the decomposition. We compose these sub-grammars to one LL(k) grammar. Similarly, corresponding token files are composed to a single token file.
3. Using the ANTLR parser generator, we create the parser with the composed grammar. The parser code generated is specific to the features we selected in the first step. That is, it is capable of parsing precisely the features in the feature instance description for which we created LL(k) grammars.

Thus composing the sub-grammars for the *Query Specification* (cf. Figure 1) feature which represents an SQL SELECT statement, the optional *Set Quantifier* feature of Query Specification and the optional *Where Clause* feature of the *Table Expression* (cf. Figure 2) feature which itself is a mandatory feature of *Query Specification*, gives a grammar which can essentially parse a *SELECT statement with a single column from a single table with optional set quantifier (DISTINCT or ALL) and optional where clause*. This procedure can be extended to other statements of SQL:2003, first mapping features to sub-grammars and then composing them to obtain a customizable parser.

In composing LL(k) grammars we must consider the treatment of nonterminals and tokens. The treatment of nonterminals is most involved. We have provided composition mechanisms for various production rules with the same nonterminal, for composition of optional nonterminals, and for composing complex sequences of nonterminals. Production rules in the grammar may or

⁴<http://www.antlr.org/>

may not contain choices for the same nonterminal, e.g., $A: B \mid C \mid D$ and $A: B$. The composition of production rules labeled with the same nonterminal is carried as follows:

- If the new production contains the old one, then the old production is replaced with the new production, e.g., in composing $A: BC$ with $A: B$, the production B is replaced with BC .
- If the new production is contained in the old one, then the old production is left unmodified, e.g., in composing $A: B$ with $A: BC$, the production BC is retained.
- If the new and old production rules defer, then they are appended as choices, e.g., in composing $A: B$ with $A: C$, productions B and C are appended to obtain $A : B \mid C$.

We compose any optional specification within a production after the corresponding non optional specification. $A: B$ and $A : B[C]$ or $A : B$ and $A : [C]B$ can be composed in that order only. Some production rules in the grammar may contain complex lists as productions. Complex lists are of the form ' $\langle NT \rangle [\langle comma \rangle \langle NT \rangle \dots]$ '. In the composer for the prototype, if features to be composed contain a sublist and a complex list, e.g., $A: B$ and $A: B [“,” B]$ respectively, then these are composed sequentially with the sublist being composed ahead of the complex list.

A feature may require other features for correct composition. Such features constraints are expressed as 'requires' or 'excludes' conditions on features. We use the notion of composition sequence that indicates how various features are included or excluded.

4. RELATED WORK

Extensibility of grammars is also subject to current research. Batory et al. provide an extensible Java grammar based on an application of FOP to BNF grammars [3]. Our work on decomposing the SQL grammar was inspired by their work. Initially, we tried to use the Bali language and accompanied tools to decompose the SQL grammar. The language extension approach of Bali clearly separates the syntax extension and the implementation of semantic actions. However, given the complexity of SQL:2003 grammar, we cannot achieve the customizability that we need by using Bali [20]. We therefore created a new compositional approach based on ANTLR grammars.

Bravenboer et al. provide with METABORG an approach for extensible grammars [7]. Their aim is to extend an existing language with new syntax from a different language. For that reason they need a parsing approach that can handle various context free grammars simultaneously because multiple languages may

be combined producing various ambiguities. Since we are decomposing only SQL, a single language, we can use the common approach of employing a separate scanner which would be insufficient for their goal. They use *Syntax Definition Formalism (SDF)* to achieve modularity, although the modularity of SDF grammars can be attributed to scanner-less generalized LR (SGLR) parsing mechanism used in METABORG [7]. The notion of inheritable grammar modules in SDF also occurs in Bali. We provide the mechanisms of adding, removing and modifying the production rules in grammar but not inheritable grammars at this point. Disambiguation mechanism of the METABORG approach consists of priority between productions, reject/prefer mechanisms for derivations, enforcing associativity for operators. Similar disambiguation constructs are also present in ANTLR in terms of syntactic and semantic predicates.

5. CONCLUSIONS

Features of SQL:2003 are user-visible and distinguishable characteristics of SQL:2003. The complete grammar of SQL:2003 can be considered as a product line, where sub-grammars denote features that can be composed to obtain customized parsability. In addition to decomposing SQL by statement classes, it is possible to classify SQL constructs in different ways, e.g., by the schema element they operate on. We propose that different classifications of features lead to the same advantages of using the feature concept. We have created 40 feature diagrams for SQL Foundation representing more than 500 features. Other extension packages of SQL can be similarly decomposed into features. We have created different prototype parsers by composing different features. Currently we are creating an implementation model and a user interface presenting various SQL statements and their features. When a user selects different features, the required parser is created by composing these features. Although SPLE and FOP concepts have been applied to programming language extension as in Bali, our approach is unique in that these concepts have been applied to SQL:2003 for the first time to obtain customizable SQL parsers. Modularity and disambiguation constructs for given type of grammars depend largely on the parsers and parser generators used. Similarly implementation of semantics for given language depends on the generated parser. Some parsers represent production rules in the grammar as methods whereas some other parsers represent these as classes. We surmise that the best approach will depend on ease of implementation of semantics. One of our future aims is to find out what kind of modularity of grammars and what kind of parsing mechanism is most suitable for feature-oriented extension of SQL.

Acknowledgments

Norbert Siegmund and Marko Rosenmüller are funded by German Research Foundation (DFG), Project SA 465/32-1. The presented work is part of the FAME-DBMS project⁵, a cooperation of Universities of Dortmund, Erlangen-Nuremberg, Magdeburg, and Passau, funded by DFG.

6. REFERENCES

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
- [2] D. Batory. A Tutorial on Feature-oriented Programming and Product-lines. In *Proceedings of the 25th International Conference on Software Engineering*, pages 753–754, Washington, DC, USA, 2003. IEEE Computer Society.
- [3] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the International Conference on Software Reuse*, pages 143–153. IEEE Computer Society, 1998.
- [4] D. Batory and S. O’Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.
- [5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.
- [6] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDMBS: Scaling Down Database Techniques for the Smartcard. In *Proceedings of the International Conference on Very Large Data Bases*, pages 11–20. Morgan Kaufmann, 2000.
- [7] M. Bravenboer and E. Visser. Concrete Syntax for Objects: Domain-specific Language Embedding and Assimilation Without Restrictions. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383. ACM Press, 2004.
- [8] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *Proceedings of 26th International Conference on Very Large Data Bases (VLDB’00)*, pages 1–10. Morgan Kaufmann, 2000.
- [9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [10] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
- [11] International Organization for Standardization (ISO). Part 7: Interindustry Commands for Structured Card Query Language (SCQL). In *Identification Cards – Integrated Circuit(s) Cards with Contacts*, ISO/IEC 7816-7, 1999.
- [12] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
- [13] M. L. Kersten, G. Weikum, M. J. Franklin, D. A. Keim, A. P. Buchmann, and S. Chaudhuri. A Database Striptease or How to Manage Your Personal Databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 1043–1044. Morgan Kaufmann, 2003.
- [14] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30(1):122–173, March 2005.
- [15] J. Melton. Working Draft : SQL Foundation . ISO/IEC 9075-2:2003 (E) 5WD-02-Foundation-2003-09, ISO/ANSI, 2003.
- [16] J. Melton. Working Draft : SQL Framework . ISO/IEC 9075-1:2003 (E) 5WD-01-Framework-2003-09, ISO/ANSI, 2003.
- [17] D. Nyström, A. Tešanović, M. Nolin, C. Norström, and J. Hansson. COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. IEEE Computer Society, 2004.
- [18] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering : Foundations, Principles, and Techniques* . Springer, 1998.
- [19] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.
- [20] S. Sunkle. Feature-oriented Decomposition of SQL:2003. Master’s thesis, Department of Computer Science, University of Magdeburg, Germany, 2007.
- [21] R. F. van der Lans. *Introduction to SQL: Mastering the Relational Database Language, Fourth Edition/20th Anniversary Edition*. Addison-Wesley Professional, 2006.

⁵<http://fame-dbms.org>