# SQL à la Carte – Toward Tailor-made Data Management

Marko Rosenmüller[1], Christian Kästner[1], Norbert Siegmund[1], Sagar Sunkle[1], Sven Apel[2], Thomas Leich[3], and Gunter Saake[1]

[1]School of Computer Science, University of Magdeburg, Germany
[2]Department of Informatics and Mathematics, University of Passau, Germany
[3]METOP GmbH, Magdeburg, Germany
[1]{rosenmue,ckaestne,nsiegmun,sunkle,saake}@ovgu.de,
[2]apel@uni-passau.de, [3]leich@metop.de

**Abstract:** The size of the structured query language (SQL) continuously increases. Extensions of SQL for special domains like stream processing or sensor networks come with own extensions, more or less unrelated to the standard. In general, underlying DBMS support only a subset of SQL plus vendor specific extensions. In this paper, we analyze application domains where special SQL dialects are needed or are already in use. We show how SQL can be decomposed to create an extensible *family of SQL dialects*. Concrete dialects, e.g., a dialect for web databases, can be generated from such a family by choosing SQL features à la carte. A family of SQL dialects simplifies analysis of the standard when deriving a concrete dialect, makes it easy to understand parts of the standard, and eases extension for new application domains. It is also the starting point for developing tailor-made data management solutions that support only a subset of SQL. We outline how such customizable DBMS can be developed and what benefits, e.g., improved maintainability and performance, we can expect from this.

## 1 Introduction

The *Structured Query Language* (SQL) is the de facto standard for accessing database systems. Since its introduction in the 70s and its first standardization in 1986, SQL has grown enormously in size and complexity [CW00]. From conceptually simple queries like the standard selection-projection-join operations and aggregation, SQL contains an ever growing number of further constructs pertaining to new areas of computing, e.g., for multi media or stream processing [CW00, SBc$^+$07]. As a result, special dialects or subsets of SQL have been proposed or standardized, e.g., Structured Card Query Language (SCQL) for smart cards [Int99], TinySQL for sensor networks [MFHH05], or StreamSQL and Oracle's CQL for stream processing [ZJM$^+$08]. All major database vendors conform only to parts of the ISO/ANSI SQL standard and often provide proprietary extensions for their systems. For developers it is obvious to add new extensions or at least to chose a custom subset of SQL, because the entire SQL standard is too large and complex for a full implementation. Because there are no guidelines for developers as which subset is adequate, this is typically an ad-hoc decision.

Currently, SQL dialects are decoupled from the SQL standard or included as extension packages. Either way, the complexity of SQL increases: a user has to analyze parallel unrelated SQL specifications to derive the actually provided functionality and many unneeded language features inherited from the base language (e.g., SQL/Foundation) are provided anyway. The resulting complexity makes it difficult to learn and use SQL. Independent of the actual task, which may be as simple as querying a single value, developers have the full arsenal ranging from simple select-from-where queries over recursive queries, object-oriented and XML extensions, control-flow statements, window functions, cube expressions, and many more. It is unrealistic to expect an application developer to know all these mechanisms and their side effects (e.g., termination of recursive queries).

We strive for a simpler, but still flexible query language on the basis of SQL. Instead of a single query language that covers all possible use cases, we aim at a family of query languages. This way, we can select a SQL dialect or even configure the query language according to the requirements of a problem domain or scenario – we select SQL features à la carte to define a concrete dialect. In a data warehouse scenario, we need a different query language than for web applications and sensor networks. For example, when we query only a single table or value, e.g., in a sensor network, we do not need the complexity caused by joins, XML extensions, or recursive queries. Furthermore, if we can exclude data types of variable length, we can use a simpler underlying database implementation. If we only allow read access, e.g., as needed for navigation systems in cars, we could even provide a SQL subset without *insert*, *update*, or *delete* statements. A read-only SQL variant could also be used to provide some form of security in the query language.

For a given scenario, an application developer can decide which SQL features are necessary and then select a database management system that provides the needed functionality or use a customizable DBMS provided for special domains, e.g., embedded systems. On the other hand, a database developer can create a DBMS matching the requirements of selected SQL features.

In this paper, we put results from prior research on extensible database systems and tailor-made data management and recent developments in software engineering into a new context. In order to realize SQL à la carte, we need a customizable query language, a customizable SQL parser and optimizer, and potentially even a customizable underlying storage system. We explain how such components can be built and how an architecture for customizable DBMS looks like.

## 2 Prior Work on Tailor-made Data Management

The need for variability is apparent in database research. There is an impressive body of research on extensions or alternative solutions of almost every single part of a DBMS or even alternative implementations of the entire DBMS from scratch. Often researchers only implement a single component and not the entire system. Many of these extensions or alternative solutions also make changes to SQL by introducing new keywords. However, extensions of the query language are mostly ad-hoc. For every extension a new syntax is

invented which is often not compatible with existing systems.

### 2.1 Tailor-Made Data Management Components

There is a large number of proposals to tailor DBMS for a special use case. We present an overview of those systems according to the five layer architecture [HR83] in order to illustrate the relation between the approaches and the DBMS architecture (see Table 1).

Layer five of a DBMS represents the data access interface. The sheer size and complexity of the SQL standard and special application scenarios are the main reasons for a number of SQL derivatives that emerged in the last 10 years. These derivatives are specialized variants for application scenarios like SmartCards [Int99], sensor networks [MFHH05], stream processing [ZJM+08], or for different programming languages [Hib08]. Parts of the query optimization have also been addressed by query optimizer generation [GD87, GM93] and specialized query engines[1], e.g., XML query engines [FH08] or query engines for embedded systems [SR05].

The next layer implements the record-oriented abstraction level of a DBMS. Specialized implementations address the transaction management system, e.g., to efficiently perform transactions in XML databases [HH07] or long-running business transactions [AMK+08]. In addition, special sorting algorithms may be used to provide minimal power consumption or to maximize performance [RSRK07].

Layer three represents access path management. Indexes have often been in the focus of research and many different tailored variants have been developed, e.g., specialized B-Trees for efficient data operations [Gra06, Gra04], R-Trees for multi-dimensional data access [Gut88], and other indexes that are applicable even in SmartCards [BBPV00].

Buffer management, which is part of layer two, also provides possibilities for optimizations. Specialized algorithms address for example embedded systems with restricted main memory and multi-media DBMS [RZ95].

Finally, variants of the file management subsystem, located in layer one, are used to support different storage hardware like in FlashDB [NK07]. To sum up, there are currently a large number of tailor-made data management components in use.

### 2.2 Tailor-Made Data Management Systems

The solutions presented above address variability in components of a DBMS. In order to build DBMS that provide different implementations of such components, there is much effort in creating extensible or customizable DBMS, in which special algorithms can be plugged. Customizability of DBMS is in the focus of database research for more than 20 years [DG01]. It aims at reusing functionality in different variants of a DBMS that are used for different application scenarios. Examples are kernel systems, which provide

[1] Query optimization cannot solely be assigned to layer five.

| Layer | Tailor-made solutions |
|---|---|
| L5 | SQL dialects or extensions (smartcards [Int99], sensor networks [MFHH05], hibernate [Hib08], stream processing [ZJM+08]), query optimizer generation [GM93], XML query optimization [FH08], lightweight query processing [SR05] |
| L4 | XML transactions [HH07], flexible transaction management [AMK+08], JouleSort [RSRK07] |
| L3 | R-Tree [Gut88], RingStorage [BBPV00], Specialized B-Tree [Gra06, Gra04] |
| L2 | Buffer management for media data [RZ95] |
| L1 | Flash Storage [NK07] |

Table 1: Exemplary tailor-made data management components according to the five layer architecture.

basic functionality and can be extended to fully functional DBMS (e.g., the EXODUS storage manager [CDF+86]), different kinds of extensible systems (e.g., user defined data types in Postgres [Sto87]), generators (e.g., the EXODUS optimizer generator [GD87]), or component DBMS [GD95]. Other approaches suggest to use very small components to ease development and reuse [CW00]. Extensibility is also achieved by commercial systems today and can be used to develop specialized systems to some degree [SBc+07].

Some approaches are based on the *software product line* concept that is understood as a general approach to build customizable software. The idea behind software product lines is to build a number of similar programs from reusable assets [PBvdL05]. These assets can be components but also smaller building blocks like source code fragments. Novel programming paradigms like *aspect-oriented programming (AOP)* [KLM+97] and *feature-oriented programming (FOP)* [Pre97] can be used to build customizable DBMS as product lines [NTN+04, TSH04, LAS05]. However, in most cases the query language cannot be modified and extensions are limited to the host system.

Furthermore, extending a single system for new application scenarios has been criticized as insufficient [SBc+07]. The reason is often that traditional RDBMS do not provide acceptable performance for special use cases. Examples are XML databases, stream processing systems, multimedia databases, data warehouses, and text databases. For these scenarios new DBMS have been developed from scratch instead of extending existing systems. This is similar in embedded systems that cannot be used with general purpose DBMS because of limited resources which also results in redevelopment of data management solutions. For example, *PicoDBMS* aims at smart cards [BBPV00] and *DELite* addresses special storage mechanisms and query processing in embedded devices [SR05]. Too often, when developing a system from scratch, a new query language dialect is invented that is incompatible with other systems.

### 2.3 Discussion

The analysis of previous work suggests that there is a need for tailor-made solutions for parts of a DBMS and also for entire DBMS. While variability in the implementation can be hidden from the user, variability in the query language can not. The user is directly confronted with different incompatible dialects. This makes it not only difficult to learn the query language, but also to exchange components or compare database implementations.

We want to address the issue of database variability from the user's perspective. Variability in the query language is needed and should be planned and coordinated. In the next sections, we propose a product line approach that allows a DBMS developer to create SQL variants tailored to the application scenario's requirements and avoids the complexity of DBMS extensions that come with their own language.

## 3 A Family of SQL Dialects

In order to provide a small, extensible subset of the SQL standard, we develop a family of SQL dialects. Each dialect of the family provides only the functionality needed for a particular application scenario. Starting with a minimal subset of SQL, e.g., that supports select-from-where queries, different SQL dialects can be created by adding functionality. For example, we can add grouping functionality, transactions, or queries needed only for special domains. The differences between the dialects can be described using the concept of *features* [CE00]. Features represent the functionality provided by a SQL dialect. For example, a feature TRANSACTIONS represents the queries needed for transaction processing (e.g., `BEGIN TRANSACTION`), i.e., it represents a number of keywords and language constructs of SQL. Similarly, in a stream processing system, a feature SLIDINGWINDOW represents support for sliding window queries. This way, the features describe the functionality of a SQL dialect in an abstract way. A concrete dialect is defined by a selection of features that describe the functionality needed for this dialect. For example, features READ, representing select-from-where queries, and SLIDINGWINDOW, used for querying streams, define a very simple SQL dialect that can be used for a stream processing system.

Based on a family of SQL dialects, a DBMS which fits to a particular SQL dialect has to be chosen or created. The functionality provided by the DBMS depends on the functionality of the actually used dialect as shown in Figure 1. For example, if the SQL dialect supports transactions, we have to parse transactional queries and the DBMS itself has to provide transactions. To create such a DBMS, a developer might implement it from scratch; however, reuse of existing functionality, e.g., by using a customizable DBMS, should be preferred. Developing such customizable solutions for a particular domain is a challenging task and will be in the focus of the next section.

In the following, we present three examples for application domains where we need different SQL dialects. We integrate these scenarios in an initial family model of SQL. In Section 4, we then present approaches to build DBMS that can be used to process a number of similar SQL dialects for differing application scenarios.

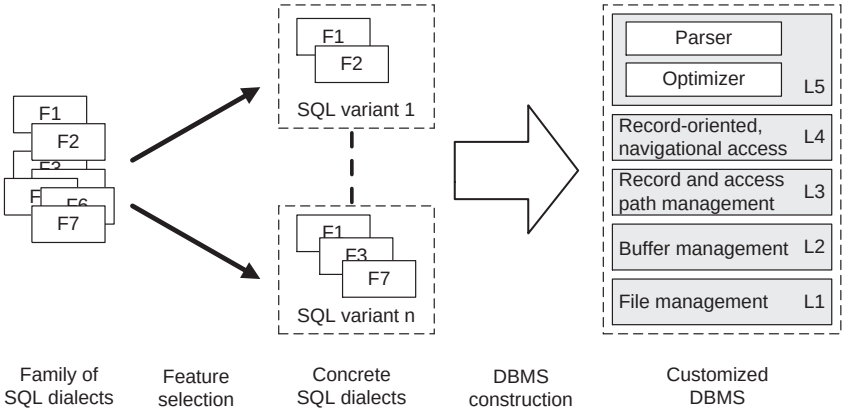


Figure 1: Defining SQL variants based on the selection of features. A DBMS has to implement the functionality to process queries of a particular variant.

### 3.1 Examples for SQL Dialects

As examples we chose web databases, sensor networks, and stream processing systems. We present an overview which functionality of SQL is needed for each scenario and illustrate how they can be integrated in a domain model.

**Web Databases.** Web databases, i.e., systems needed to store data for simple web applications, usually only require a small subset of SQL including selection-projection-join queries, aggregation, transactions, etc. Mechanisms like stored procedures, triggers, prepared statements, or views are usually not needed. An example often used for web development is MySQL. In earlier versions, MySQL was often used for web development, despite its lack of advanced features like transactions or stored procedures. During its evolution, such features were added, however, as reaction, some MySQL developers forked a new release called Drizzle[2] that slims down the MySQL code to a small lightweight data-management solution with SQL but without some of the advanced features. Developing such a slim web DBMS has a number of benefits. Examples are easier performance optimizations, reduced maintenance, reduced implementation effort, and easier development of client applications. This is especially important for web applications where developers are usually no database experts. In contrast to high-end systems like Oracle or DB2, web databases are often small systems that are free of charge and easy to install and maintain. Supporting only a subset of the DBMS functionality allows a vendor to offer cheap DBMS for this market segment and gives even end users the chance to maintain the system.

**Sensor Networks.** Similar to the web database example, sensor networks only require a small subset of SQL, e.g., as provided with TinySQL [MFHH05]. Basic functionality

[2]https://launchpad.net/drizzle

includes select-from-where queries. However, also new query types are required. For example, the TinySQL dialect explicitly supports spatial queries to access only nodes in a user defined region and temporal queries to access data for a given time interval that are not supported by general purpose DBMS.

**Stream Processing.** The last example is a stream processing DBMS that also uses special SQL extensions, e.g., StreamSQL or Oracle's CQL [ZJM+08]. In these systems, multiple streams might be supported and aggregation functions are usually executed on windows defined on the data stream. Support for multiple streams is not necessary for every stream processing application and might be omitted to simplify the query engine. Furthermore, different window definitions are possible. Since most stream processing applications use only sliding windows, there might be a simple SQL dialect that provides only this type of window. Thus, even in a single domain, different SQL dialects might be needed.

### 3.2 A Domain Model of SQL

The examples above show that each dialect provides certain queries that are not required for others. Basic functionality is needed in all dialects, e.g., select-from-where queries, but even functionality like simple data manipulation is not always needed (e.g., *insert* in a stream processing scenario).

In order to derive a single domain model that describes a whole family of SQL dialects, we will use features to describe the dialects and then combine the descriptions of all dialects in one domain model. Features are well known from domain analysis [KCH+90] and software product line engineering [CE00]. A feature is a distinguishable characteristic of software that is of interest to some stakeholder [CE00]. It describes the differences and commonalities between systems in a domain. Features and dependencies between features are depicted/organized in a hierarchical way in *feature diagrams*.

The feature concept can be used to describe differences in the domain of query languages which results in a description of a family of SQL dialects. In Figure 2, we define an initial feature diagram on the basis of differences and commonalities of the three scenarios discussed in Section 3.1. The root of the tree represents the described concept and the remaining nodes are features. For example, feature TRANSACTIONS represents functionality needed for transaction processing. A feature can also have sub-features (i.e., child nodes), e.g., feature CORE combines features READ and WRITE which represent basic functionality. Optional features are denoted with an empty dot (e.g., WRITE) meaning that the feature is not needed for every SQL dialect. In contrast, mandatory features, denoted with a filled dot, are part of every variant. Between grouped features (e.g., subfeatures of CORE) there can be OR-relations, which means that the features can be selected in combination but at least one of the features has to be selected.[3]

The SQL dialect needed for a web database can now be defined by listing the needed features, i.e., DATATYPES, STOREDPROCEDURES, TRANSACTIONS, JOIN, AGGREGATION, and all

[3]There are further relations between features (e.g., XOR) not used here.

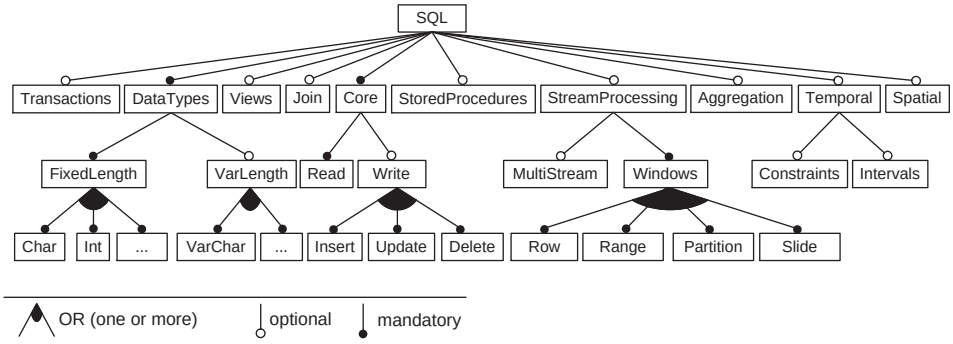


Figure 2: Feature diagram for a subset of SQL that supports core functionality and several extensions.

the CORE sub-features. Similarly, we can describe SQL dialects for a sensor network using the sub-features of CORE as well as spatial and temporal queries (provided by features SPATIAL and TEMPORAL). For sensor networks that do not provide position information, also dialects without spatial queries are supported by omitting the optional feature SPATIAL. Finally, a SQL dialect for stream processing should provide core functionality without write support. Stream extensions are provided by feature STREAMPROCESSING with optional feature MULTISTREAM (representing operations on multiple streams) and feature WINDOWS which summarizes different window functions. A small dialect might provide only sliding windows and omit support for other window types. The OR-relation between window types means that all combinations of windows might be selected to derive a valid SQL dialect, but at least one has to be selected. This is an important requirement since it avoids invalid SQL dialects without any window operations in a stream processing setting.

OR relations are not the only dependencies between features. There can be additional dependencies like *requires* or *excludes* relationships between features that avoid invalid feature combinations. Such dependencies can be inevitable (*domain dependencies*) or caused by the implementation, e.g., to simplify the development of a DBMS. For example, TEMPORAL queries might require a DATETIME data type. In Figure 2, such dependencies are not shown because most of them depend on the concrete implementation of a DBMS.

## 4 Processing the Query

With a family of SQL dialects we can tailor the query language to a given scenario. However, a tailor-made query language itself has little value, but it has to be backed up by an underlying DBMS that can process the queries. In this section, we discuss the implications of tailoring query languages for the underlying DBMS and propose three approaches how to cope with the variability of the query language.

### 4.1 Requirements on the DBMS

Depending on a given SQL dialect, the underlying DBMS has to provide the corresponding functionality to process the queries for that dialect. We propose that a given DBMS does not support all SQL dialects (but still potentially many dialects), but is tailored to a specific dialect. For example, if a dialect supports aggregation, the query engine has to understand, optimize, and execute aggregation queries. Such changes not only affect the query processor but potentially several other layers of the DBMS. In Table 2, we give an exemplary overview of potential changes required to implement Aggregation, Views, Write-Access, and Joins (cf. Fig. 2). Its purpose is to illustrate that certain features of the query language can have consequences for the implementation of the entire DBMS. The table is neither complete regarding necessary extensions to a DBMS, nor does it cover all features of SQL.

From Table 2 it can be seen that changes required by different dialects cut across the entire DBMS architecture. First, the queries have to be *parsed* and *translated* into a query execution plan. Hence, a parser is needed that can parse the queries defined by a specific dialect. For example, if a dialect does not support aggregation we do not have to parse `GROUP BY` statements nor do we have to consider aggregations in query execution plans. Next, many optimizations are only needed for special dialects. For example, if joins are not present in a dialect, query optimization can be strongly simplified and must consider only lists instead of trees. Similarly, if aggregation is supported, the optimizer must consider adequate algebraic rules and cost optimizations.

Similar changes can be found in other parts of a DBMS. For example, algorithms for aggregation and joins have to be provided if these features are selected on the language level and also specialized access paths for joins have to be considered depending on the SQL dialect. Even the lowest layers can be affected, as in those cases where write operations or variable length data types are not needed in which all layers can be simplified. The changes needed for a SQL language feature often affect the entire DBMS implementation.

### 4.2 Toward Tailor-made DBMS

If SQL features can cut across the entire DBMS, how can we map a family of SQL dialects to a DBMS implementation? There are several possible scenarios, which all provide benefits over the current status quo of a huge SQL standard and uncontrolled, incompatible, specialized SQL dialects:

- In the first scenario, an existing DBMS is mapped to one of the SQL dialects. For example, if we have a DBMS for sensor networks, the DBMS developer could ex-post describe the supplied language features as a feature selection of the SQL language model. Our sensor DBMS could support features CORE, SPATIAL, and TEMPORAL (cf. Fig. 2). This information can be used for communication with stakeholders and customers. It allows for comparing a specific DBMS with other implementations and a customer can easily check if her functional requirements are met by a given implementation.

| Layer | | Aggregation | Views | Write-Support | Joins |
|---|---|---|---|---|---|
| Data Access | Parsing | 'group by', 'min', 'avg', ... | 'create view', ... | 'insert', 'update', 'delete', ... | 'join', multiple columns, ... |
| | Translation | introduce grouping operator $\gamma$ in relational algebra | resolve views | | introduce $\bowtie$ operator in relational algebra, ... |
| | Optimization | algebraic rules & heuristics for $\gamma$, cost estimations for $\gamma$ | view materialization | | several algebraic rules & heuristics, cost estimations, select join strategy & algorithm, dynamic programming, ... |
| Operations & Access Path | | operations for sorting, min, avg, ... | | write operations (data and all access paths) | join algorithms, join indexes, ... |
| Page management | | temporary pages | | TID concept, segment space management | |
| Buffer management | | temporary buffer pools | | dirty page management | |
| OS | | | | write support | |

Table 2: Impact of SQL language features on the DBMS architecture.

The benefit of this scenario is its power in communicating differences and commonalities of DBMS between stakeholders.

- When implementing a new DBMS, developers chose to support a subset of SQL language features, i.e., one or more SQL dialects as specified by the feature model. For example, if we want to implement a sensor network DBMS, a DBMS domain engineer could ex-ante define the SQL language features which she wants to support. This way, we would not have to define an own SQL subset or even proprietary SQL extensions, but could reuse this family of SQL dialects. Developers are not directly confronted with the full SQL standard, but have clearly defined subsets that appear feasible to implement.

  The main benefit of this scenario is that developers can reuse an existing decomposition of the SQL standard. This reduces the likelihood that they invent their own language or implement just an arbitrary, potentially even undocumented SQL subset when they are faced with the overwhelming full SQL standard. Together with the first scenario, this reduces the overall complexity compared to uncontrolled, proprietary SQL dialects.

- Finally, it is also possible not to map a single system to one or more SQL dialects, but to introduce variability in the DBMS implementation as well. If we have a flexible, extensible family of DBMS systems, e.g., developed from core systems [DG01], component systems [Sel08], or software product lines [NTN+04, TSH04, LAS05, KAB07, RSS+08], we can directly map variability of the query language to variability of the query processor's implementation. Such approaches allow a developer to reuse functionality when implementing a number of similar DBMS and bear the potential of decreasing development costs and time-to-market and increasing software quality. This, however, does not necessarily mean that we want to develop a product line of DBMS that can match all possible SQL dialects, ranging from embedded systems to data warehouses. Such "one size fits all" approach is technically extremely challenging, if possible at all. Nevertheless, it is possible to tailor a family of database systems for certain domains, e.g., different embedded scenarios. Such a variable DBMS would not support all dialects, but a larger subset of dialects.

In the remainder, we aim at the third scenario, because it raises the most challenges, but also provides the most opportunities. We discuss how to implement customizable DBMS to match the requirements of multiple SQL dialects and not only a single dialect. We present an overview following the five layer DBMS architecture top-down.

### 4.2.1 Parsing SQL Dialects

Parts of a DBMS depend on the functionality of the supported SQL dialect, they are only needed if a particular feature of the family of SQL dialects is used. In contrast, a SQL parser is needed for every DBMS and functionality of the parser does not significantly differ for different SQL dialects, except that different parts of the SQL grammar have to be parsed. Hence, reuse of parser code for different SQL dialects is desirable and easy to

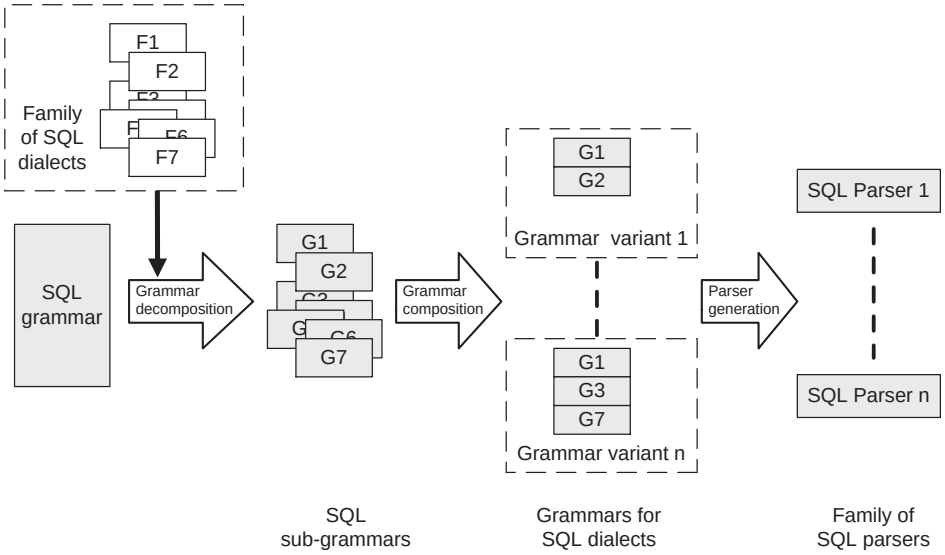


Figure 3: Generating a family of SQL parsers by decomposing the SQL grammar.

achieve. SQL parsers can be created using parser generators that receive a SQL grammar as input and generate a program that can parse SQL queries according to this grammar. To create a parser for a particular dialect, we need the grammar for that dialect and thus can automatically generate the according parser.

In order to create a grammar according to a SQL dialect, an approach for grammar composition can be used as shown in Figure 3. The idea is (1) to decompose a full SQL grammar into sub-grammars according to the features derived from the domain analysis of the SQL standard and (2) to compose sub-grammars on the basis of the features a user selects for a specific SQL dialect. Composed grammars are then used to generate SQL parsers using parser generators.

In [SKS+08, Sun07], we have presented a decomposition of SQL:2003 and described an approach to generate variants of SQL parsers based on the language features. We used a more fine-grained decomposition of SQL than depicted in Figure 2 which was driven by the SQL syntax resulting in hundreds of features. Based on a family of SQL dialects (similarly as presented in Fig. 2), we have also decomposed the SQL:2003 grammar according to the features of the family retrieved from the standard.[4] That is, for each feature we have extracted the according part of the SQL grammar into a sub-grammar for that feature. For example, a feature AGGREGATION (cf. Fig. 2) that represents the `GROUP BY` statement and some other syntactical extensions can be extracted into a separate sub-grammar. A SQL dialect that only supports core operations and aggregation is then created by composing the sub-grammars for READ and WRITE operations (i.e., `SELECT`, `UPDATE`, etc.) and

[4]The extracted features of SQL:2003 are based on the BNF grammar specification of SQL:2003 and other information given in SQL/Foundation. We have not further decomposed extension packages like XML.

AGGREGATION and generating the parser from that composed grammar.

When processing a SQL query, a parser internally creates a syntax tree which represents the query and is the basis for generating a query execution plan. Hence, a generated parser has to be extended with functionality that translates the syntax tree into a query tree which is the representation needed for query optimization.

### 4.2.2 Query Optimization

Based on a generated parser that can be tailored to a SQL dialect, a complete SQL query processing engine can be created that processes only the queries of a chosen SQL dialect. Query optimization and execution have been one of the main challenges when creating customizable and extensible DBMS for more than 20 years. For example, the Volcano project addressed this issue and has shown that customizability is possible but complex [GM93]. These solutions use query optimizer generators to introduce variability needed for customization. When processing queries of a SQL dialect, the optimization process depends on the features selected for this dialect. As a result, transformation rules that specify the needed transformations for optimizer generators, have to be defined depending on the features of a given dialect, e.g., for grouping operations (cf. Table 2). Furthermore, whole parts of an optimizer are only needed if a particular SQL feature is included in a concrete dialect, e.g., considering materialized views during optimization is only needed if views are available in the dialect.

Novel software engineering techniques, not available 20 years ago (e.g, software product lines and feature-oriented programming), can help in developing customizable query engines and reduce the complexity of building variable solutions. This was already shown for other parts of DBMS and we currently apply this to query processors. Using a software product line approach, also allows for providing an abstract view on query processors with respect to variability and enables systematic reuse between different variants.

As a first step, customizable solutions do not have to provide full customizability according to a whole SQL family, but only parts of a query processor might be addressed. Further variability can then be introduced in a step wise manner. Customizable query optimizers might also not address the whole space of SQL in general but only smaller sub-parts, i.e., a number of similar dialects.

Because of the strong interdependence of query optimization and execution, e.g., indexing, buffer management, pages, data types, and information in the data dictionary, a customizable query processor cannot be developed without taking the underlying layers of a DBMS into account. As already shown in Table 2, there are components in all parts of a DBMS that are affected. However, customizability of the query engine does not enforce a customizable DBMS. For example, a query engine can be used for SQL dialects with or without support for Joins as long as the underlying DBMS supports the needed join algorithms. That is, for a number of SQL features we can have a fixed implementation of the lower layers of a DBMS that includes all of the features. This inhibits some of the benefits expected from a small SQL dialect (e.g, reduced functionality in all layers of a DBMS); however, it still reduces the functionality and complexity of the query processor. The other solution, a fully

customizable DBMS, is most challenging but also most promising with respect to expected benefits, i.e., a complete DBMS that perfectly fits to a chosen SQL dialect. We will analyze such customizable DBMS in the last part of this section.

### 4.2.3 Query Execution

We have shown how a customizable query processor can be developed and we will now outline how this might be continued in the lower layers of a DBMS. There are implementation techniques, e.g, using components, that achieve customizability at least for parts of a DBMS. In the following, we will not show how a concrete solution could look like, but analyze how customizability should be used in the context of creating a DBMS that can be tailored to a particular SQL dialect. There are a number of techniques for implementing this customizability and we will now review some of them.

**Components.** The idea of customizable or variable DBMS is to reuse functionality in different variants of a DBMS [DG01]. Variability provided by such systems is usually limited to exchanging parts of a DBMS that can be modularized very well, e.g, with components [Sel08]. Functionality that is scattered over the entire DBMS is hard to modularize and cannot be easily exchanged with a different implementation. This, however, is not a barrier for developing customizable DBMS in general. As already stated for query optimization, also a customizable DBMS does not have to provide all the variability provided by the SQL dialects that it implements. Benefits can already be achieved by exchanging selected parts of a DBMS. For example, to support a DBMS with and without join operations, temporary pages and temporary buffer pools can be developed as optional parts of a DBMS (cf. Table 2). Hence, choosing a SQL dialect without joins results in a DBMS without these optional components. Thus already existing component DBMS can be used to realize the underlying variable DBMS. Using components to implement the DBMS, however, this is only possible for functionality that can be modularized and is not cutting across the whole system.

**New Programming Paradigms.** Some features of a DBMS affect large parts of the source code and thus also cut across other components. In order to achieve customizability of such *crosscutting* features, these can be implemented in a modular way using new programming paradigms. For example, Nyström et al. used aspect-oriented programming (AOP) in combination with components to modularize crosscutting features that effect multiple other components [NTN+04]. Tešanović et al. used AOP to refactor the C version of Oracle's Berkeley DB to modularize crosscutting features that had been implemented with C preprocessor statements before [TSH04]. Feature-oriented programming (FOP) is an alternative programming paradigm that can be used for modularization of crosscutting features. In a number of case studies, we have used AOP and FOP to refactor and implement DBMS product lines [LAS05, KAB07, RSS+08] and have shown that FOP can be used without negative impact on performance as sometimes found with components.

**Combining a Family of SQL Dialects and Customizable DBMS.** Independent of the concrete implementation technique (e.g., using components or FOP), the product line concept can be used to generate the underlying DBMS according to the features of a SQL dialect. A DBMS product line is also described using a feature model as it is done for our family of SQL dialects. However, the features of a DBMS product line, e.g., as in the FAME-DBMS product line [RSS+08], differ from the features found in the SQL family. For example, a feature RECOVERY, representing recovery functionality of a DBMS, does not have a representation in the SQL family. Both product lines, the DBMS product line and the family of SQL dialects, can be integrated by combining their feature models, i.e., features of the SQL family have to be mapped to features of the underlying DBMS product line. For example, if a feature JOIN of our SQL family is selected, the DBMS has to provide a join algorithm which might be implemented in a feature JOIN of the DBMS. This is similar for feature TRANSACTIONS which requires a transaction management system in the DBMS. Such dependencies between the SQL feature model and a DBMS feature model can then be represented as constraints between features of both models. When defining a concrete SQL dialect by selecting the needed SQL features, we can then automatically decide which features of the DBMS are needed. Features, representing functionality not defined by the SQL family (e.g., RECOVERY), additionally have to be selected when needed to complete the DBMS configuration.

## 5 Discussion and Perspective

In the following, we discuss some issues that are important when developing tailor-made DBMS and present some perspectives for further developments.

**Granularity of a SQL Decomposition.** In Section 3.2, we have provided a coarse-grained decomposition of SQL but we could also show that a much more fine-grained decomposition is possible [SKS+08]. Finding the appropriate granularity for a SQL decomposition and answering the question what *good* SQL dialects are, are some of the challenges to be tackled in future research. One of the involved problems is the resulting complexity of a DBMS implementation which increases for a fine-grained decomposition. On the other hand, a fine-grained decomposition results in SQL dialects that better fit to application scenarios. Furthermore, we only considered SQL/Foundation as a basis for decomposition and ignored extension packages like XML. These are already separated from SQL/Foundation but may also be further decomposed.

**Extensibility of the SQL Family.** Using a family of SQL dialects and a domain model that describes the family, it is easier to understand the standard due to a structured representation and clear separation of functionality described by the model. A SQL domain model can be used to communicate DBMS functionality between developers and also customers. However, it does not avoid arbitrary uncontrolled additions to the SQL syntax for new application scenarios. Nevertheless, it is easier to align extensions with the existing model and to

achieve compatibility of dialects. Furthermore, it is easy for developers to derive a concrete dialect by deciding which SQL features have to be included. Especially when creating small dialects, e.g., for embedded systems, web databases, and also for research prototypes and teaching SQL, a listing of needed features is a simple task and ensures that the dialect is compatible with the standard. A documentation that only describes functionality of the used dialect further simplifies the use of SQL. Generating such documentation variants is possible using a product line approach.

**One Size Fits all?** Stonebraker et al. suggested that a single DBMS is not appropriate for all application scenarios [SC05]. This also applies when using a customizable DBMS, e.g., implemented with a software product line, because with increasing functionality also the complexity of the implementation increases and the reuse between different DBMS decreases due to highly differing requirements. Hence, independently of the implementation technique, a customizable DBMS will usually not cover all possible SQL dialects but only a number of similar dialects that are promising with respect to reuse of functionality. For example, there might be a number of SQL dialects for data management in embedded systems (e.g., for sensor networks and smart cards) that are significantly different, but can be addressed with one customizable and extensible underlying storage system for embedded devices. On the other hand, there might be functionality that cannot be reused between systems because of completely different implementations. For example, it is hard to achieve any substantial reuse between DBMS for embedded systems and DBMS for server systems. For that reason, there will probably be only a high degree of reuse between similar systems of a domain (e.g., embedded systems) and developing one customizable DBMS for a number of different domains is not reasonable.

Developing a customizable DBMS, i.e., a product line of DBMS, always introduces additional implementation effort to achieve the needed reuse. Because of this reuse, however, the additional effort pays off if different similar products are needed and these products do not have to be developed in parallel. Reasons are reduced maintenance, lower development costs and time, and improved software quality. Hence, when developing customizable DBMS it is questionable which dialects such a DBMS should support. Finding the correct dialects to support and thus a good balance between reuse and complexity is a key for success when developing customizable DBMS.

## 6 Conclusion

Approaches that aim at providing tailor-made data management components can be found in all layers of DBMS including different SQL dialects for special domains which are often uncoupled from the SQL standard. Work on these tailor-made solutions, however, is usually uncoordinated and each solution comes with an own SQL dialect that sometimes does not reflect the underlying functionality.

Our goal is to provide a uniform foundation for tailor-made data management starting with the query language. With a family of SQL dialects we have shown how some of the

complexity issues of SQL can be addressed by structuring the SQL standard and providing extensibility to add new language features for special domains like sensor networks or stream processing systems. We have shown how we can create concrete SQL dialects by choosing SQL features à la carte from the family of SQL dialects. Concrete dialects tailor SQL as a query language for a particular domain or even a specific application scenario.

A family of SQL dialects is only a first step for achieving a uniform basis for data management that includes all application domains. Integrating existing research that addresses variability in all parts of a DBMS is needed and a complete family of SQL dialects has to be created. Query processing and the underlying DBMS layers may then be turned into customizable systems in a step-wise manner. In the long run, even the standardization process of SQL could be based on language features and a family of dialects. There are many challenges for every part of a DBMS and all application domains; however, it allows us to overcome the complexity problems of SQL and to build tailor-made data management solutions.

## Acknowledgments

We thank Ingolf Geist and Eike Schallehn for discussions about this work. The work of Marko Rosenmüller and Norbert Siegmund is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C, and German Research Foundation (DFG), project number SA 465/32-1. Sven Apel's work is funded partly by German Research Foundation (DFG), project AP 206/2-1. The presented work is part of the projects ViERforES[5] and FAME-DBMS[6].

## References

[AMK+08] A.-B. Arntsen, M. Mortensen, R. Karlsen, A. Andersen, and A. Munch-Ellingsen. Flexible Transaction Processing in the Argos Middleware. In *EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 12–17. University of Magdeburg, 2008.

[BBPV00] 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 (VLDB)*, pages 11–20. Morgan Kaufmann, 2000.

[CDF+86] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. E. Richardson, and E. J. Shekita. The Architecture of the EXODUS Extensible DBMS. In *Workshop on Object-Oriented Database Systems*, pages 52–65. IEEE Computer Society Press, 1986.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[5] http://vierfores.de/
[6] http://fame-dbms.org/

[CW00] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1–10. Morgan Kaufmann, 2000.

[DG01] K. R. Dittrich and A. Geppert. Component Database Systems: Introduction, Foundations, and Overview. In *Component Database Systems*, pages 1–28. dpunkt.Verlag, 2001.

[FH08] J. A. M. Filho and T. Härder. Tailor-Made XML Synopses. In *International Baltic Conference on Databases and Information Systems (DB&IS)*, pages 25–36, June 2008.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. *SIGMOD Record*, 16(3):160–172, 1987.

[GD95] A. Geppert and K. R. Dittrich. Strategies and Techniques: Reusable Artifacts for the Construction of Database Management Systems. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 932 of *Lecture Notes in Computer Science*, pages 297–310. Springer Verlag, 1995.

[GM93] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 209–218, 1993.

[Gra04] G. Graefe. Write-Optimized B-Trees. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 672–683, 2004.

[Gra06] G. Graefe. B-tree Indexes for High Update Rates. *SIGMOD Record*, 35:39–44, 2006.

[Gut88] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Readings in Database Systems*, pages 599–609. Morgan Kaufmann, 1988.

[HH07] M. P. Haustein and T. Härder. An Efficient Infrastructure for Native Transactional XML Processing. *Data and Knowledge Engineering (DKE)*, 61(3):500–523, 2007.

[Hib08] Hibernate. Relational Persistence for Java and .NET, 2008. http://www.hibernate.org.

[HR83] T. Härder and A. Reuter. Concepts for Implementing a Centralized Database Management System. In *Proceedings of the International Computing Symposium*, pages 28–60, 1983.

[Int99] 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.

[KAB07] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 223–232, 2007.

[KCH+90] 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.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[LAS05] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of the East-European Conference on Advances in Databases and Information Systems (ADBIS)*, volume 3631 of *Lecture Notes in Computer Science*, pages 324–337. Springer Verlag, 2005.

[MFHH05] 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, 2005.

[NK07] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 410–419. ACM Press, 2007.

[NTN+04] 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.

[PBvdL05] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Verlag, 2005.

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Verlag, 1997.

[RSRK07] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: a Balanced Energy-efficiency Benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 365–376. ACM Press, 2007.

[RSS+08] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *EDBT'08 Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 1–6, March 2008.

[RZ95] D. Rotem and J. L. Zhao. Buffer Management for Video Database Systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 439–448. IEEE Computer Society Press, 1995.

[SBc+07] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. One Size Fits All? Part 2: Benchmarking Studies. In *Third Biennial Conference on Innovative Data Systems Research*, pages 173–184, 2007.

[SC05] M. Stonebraker and U. Cetintemel. One Size Fits All: An Idea Whose Time Has Come and Gone. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 2–11, 2005.

[Sel08] M. Seltzer. Beyond Relational Databases. *Communications of the ACM (CACM)*, 51(7):52–58, 2008.

[SKS+08] S. Sunkle, M. Kuhlemann, N. Siegmund, M. Rosenmüller, and G. Saake. Generating Highly Customizable SQL Parsers. In *Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, pages 29–33, 2008.

[SR05] R. Sen and K. Ramamritham. Efficient Data Management on Lightweight Computing Devices. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 419–420. IEEE Computer Society Press, 2005.

[Sto87] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 289–300. Morgan Kaufmann, 1987.

[Sun07] S. Sunkle. Feature-oriented Decomposition of SQL:2003. Master's thesis, Department of Computer Science, University of Magdeburg, Germany, 2007.

[TSH04] A. Tešanović, K. Sheng, and J. Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *Proceedings of International Database Engineering and Applications Symposium*, pages 291–301. IEEE Computer Society Press, 2004.

[ZJM+08] S. Zdonik, N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, M. Cherniack, U. Cetintemel, and R. Tibbetts. Towards a Streaming SQL Standard. *Proceedings of the VLDB Endowment*, 1(2):1379–1390, 2008.