On Design of Domain-Specific Query Language for the Metallurgical Industry

Andrey Borodin(✉), Yuri Kiselev, Sergey Mirvoda, and Sergey Porshnev

Dept. of Radio Electronics for Information Systems,
Ural Federal University, Yekaterinburg, Russia
amborodin@acm.org, ykiselev.loky@gmail.com,
sergey@mirvoda.com, s.v.porhsnev@urfu.ru
http://urfu.ru

Abstract. Many systems have to choose between user-friendly visual query editor and textual querying language with industrial strength in order to deal with big amount of complex data. Some of them provide both ways of accessing data in a warehouse.

In this paper, we present key features of domain specific querying language, which we designed as a part of informational system of a steel production plant. This language aims to give an opportunity of easy data manipulation to those who know what the data actually is and to provide an easy way to discover what the data is for others. We also provide an evaluation of the designed language. The main focus of our estimation was to measure effort required for discovering dataset and deriving simple math expressions.

Although the paper overviews a data model for one specific domain, it can be easily applied for different domains.

Keywords: Query language · Domain-specific language · Data warehouse · Metallurgy · Steel plant

1 Introduction

Most of general-purpose database management systems (DBMS) have their own programming languages. Usually they are forms of structured query language (SQL), but also they can be a variation of DML, MDX, XQuery, etc. On top of these languages, software developers build all kinds of visual query construction toolkits, data browsers, and pivot tables. Specific data management systems operate on the top of predefined data types with their own unique features and requirements. Their developers can start directly from creating specific data visualization and querying tools, using standard DML as a media from user interface (UI) to DBMS or omitting this media at all in case of specific data warehousing techniques.

Our task was to create an automated analytical and modelling system for metallurgical production [1], which had contained a data warehouse and a query constructor. An existing system already had a visual query editor as a web service. Its users could define constraints and the form of the desired data. Then this visual query was compiled to Oracle PL\SQL query and executed on data warehouse.

After investigation of the system with metallurgical technical staff, we concluded that it was not the best fit for the plant. This was mainly because of different needs of the users. Some users were data scientists and they required specific features that were hard to implement with web UI. The other ones required a possibility of sharing significant parts of queries with their colleagues. Finally, some of them simply had no mouse devices or touch manipulators due to specific environmental restrictions, so all of the users expected all querying features to be available from a keyboard.

The initial move was to add keyboard shortcuts, and users actively welcomed this. A complication of the system leaded to a necessity of the query language as a main media between a query constructor and a data warehouse. This approach could have prevented the evolving complexity of a query editor, which can be used now as the only assistant tool in a query construction. We stated a list of the following advantages of domain-specific query language (DSQL):

1. Version control systems (VCS) out of the box. Software developers have been using safe source code tracking for many decades. Since queries are the main tools for metallurgical data analysts, they should be treated the same way, but the development of all VCS features in a custom query editor is not economically viable.
2. Portability. Text queries can be written even on a whiteboard and a notepad. One of the great advantages of text queries is that they are unambiguous: there are no hidden parts in a text query.
3. Detachable. Text queries can be run in different warehouses. They should not depend on identifiers, conditions, and environmental variables.
4. Fragmentation. An analyst can extract some feature from a query and partly pass it to his colleagues. E.g. from a query computing metallurgical length of all damaged slabs of specific operational brigade we can extract a part which searches all damaged slabs of the brigade.
5. Embedding. Software developers easily can embed queries into autotests, side subsystems can embed query parts into their code or resources.
6. Specification. DSQL can be a part of a systems applied programming interface (API) for third party systems if DSQL specification is precise enough.

This paper describes features of the DSQL we developed for our system (DSQLM M stands for metallurgy), although these six advantages are common for almost any DSQL, designed similarly. Term DSQL is generic and does not reflect its orientation towards steel production data [5]. DSQLM is used to point out our specific implementation. However, the variety of auxiliary data types is stored in systems data warehouse, which is not limited only to technical, chemical or signal data. Thus, most of design decisions of this DSQLM can be applied to any other domain.

2 Language Goals

Potential DSQLM users are steel plant employees; most of them are technological data analysts who study properties of plant products in correlation with production process parameters. Some of them are support staff in finance, ITSM and management.

Most of the users have higher education; some of them hold PhD in technology with works related to steel production. Most of them have neither deep knowledge of informational technologies nor experience in using SQL-like languages. Most of the users know the programming data types while lack of the understanding of data processing algorithms.

In collaboration with the staff, we designed this set of goals:

1. DSQLM should be compatible with Google-style queries and document models (bag of words - BoW [7]).
2. DSQLM should be declarative, i.e. to define not how to get the result but to declare the requirements for retrieving the result.
3. DSQLM should be as deterministic as possible. This means that two executions of the same query should yield the same results. This goal is not strict, because overwhelming determinism could render impossible environmental variables usage, for example, use of the current time in a query would not meet such requirement.
4. DSQLM should aim to reduce the complexity of genealogy. In steel production, units are arranged in a genealogical graph; i.e. metal sheets are produced from different slabs and slabs are produced from different bars. An analyst should be able to examine properties of all bars while studying a certain sheet. This requires a language to have an easy way to define data aggregation over domain data model.
5. The result of DSQLM query is a data table, which can be exported to such tools like Excel, Statistica, and MATLAB.
6. The queries should be fragmental to extract features and combinable to develop query libraries. This point contradicts the 2^{nd} advantage of text queries (portability), yet it can be handled with assistant tools extracting all-self-sufficient queries from query referencing libraries.
7. DSQLM is a tool to extract relevant data from a warehouse.

During language development, these were rather goals then requirements. Although users describe all of them as crucial, they cannot be achieved simultaneously. Note that goals 4 and 5 are domain specific, yet all other goals are generic and can be applied to advance any DSQL

3 Domain Data Model

Although relational database management systems (RDBMS) suffer from the effect of object-relational impedance mismatch (ORIM) [2], such systems are most widely used for data warehousing. The ORIM is a set of conceptual and technical difficulties that are often encountered when a RDBMS is used by a program written in an object-oriented programming language or style.

While DSQL is not necessary object-oriented, ORIM is closely related to reducing of genealogy complexities. Using SQL as a query media, most genealogy manipulations are translated into JOIN constructions and non-obvious aggregation conditions that are hard to understand. We observed this in a visual query editor and made the following conclusion. Using SQL-like syntax and RDBMS data model (tuples and relations between them) would lead to unclear DSQL syntax and semantics.

An alternative for RDBMS would be object oriented and document [9] databases. In particular the system contains MongoDB warehouse as one of important storages. However, if DSQL were based on MongoDB query language it would require the understanding of JavaScript concepts that are hard to master for users. Moreover building DSQL that is strongly connected to data model would be restrictive in the future. The warehouse is designed to be easily adaptable to data of certain plant, whereas readjustment in language syntax, libraries and documentation could cause significant delays in the release of a system.

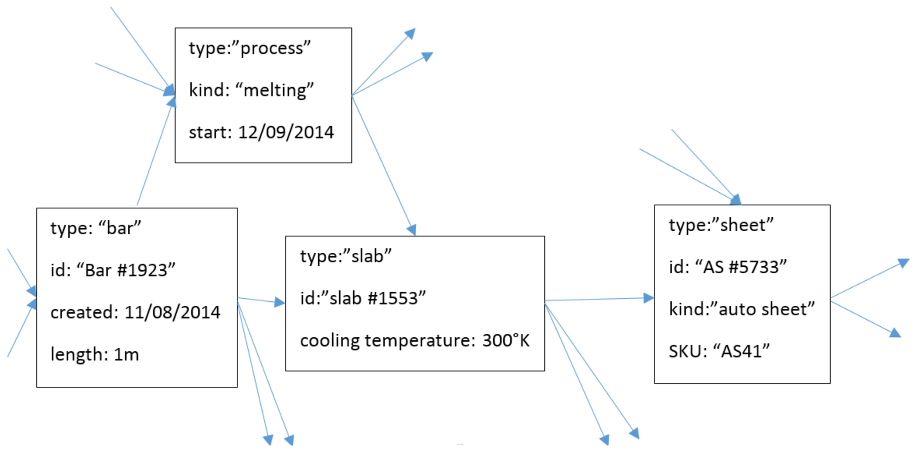

Fig. 1. Partial example of a dataset

Hence, we designed the following data model (Fig. 1 presents four objects of a possible dataset):

1. The main unit of data is an object ("slab", "sheet", "process", "firm", "user", "aggregate", etc.)
2. An object has different parameter values ("color", "weight", "length", "creation time", etc.)
3. An object has forward and backward references to objects. A set of forward objects is assumed as all objects dependent or created by referencing object. There is a corresponding backward reference for each forward reference.

4. Every object has a string parameter “type”; every type is an object itself with type “type”. Type objects do not contain references.
5. Every parameter value contains a corresponding object with a type “parameter” and information related to its semantics, unit of measure and data type.

The data model is targeting the goal 4. Its also aims to reflect semantics of data. This means that a dataset neccesarily includes data about its structure as a first-class data.

Semantics of every reference depends on a target of this reference. However, the main purpose of the references is to reflect oriented genealogical graph.

This data model can be applied generically to any DSQL experiencing bottleneck in objects genealogical relations.

4 Language Features

Since the result of a DSQLM is a table, the core of textual query should be oriented towards table specification. A query includes parts, which can be essentially one of three kinds:

1. Column definition (Computable expression)
2. Row filter (Boolean expression)
3. Search filter (BoW)

If a table requires a Boolean column, this must be specified in column options discussed further. Query parts are delimited with commas and consist of an identifier, options, and a body. The body is surrounded with curly brackets; options are surrounded with square brackets. We chose an excessive amount of formatting characters to support an easy parsing and, if necessary, to develop clear formatting guidelines. Many languages like XML subsets tend to overwhelm their users with many strict control characters(i.e. the requirement of matching opening and closing tags), while languages like SQL tend to derive necessary information from the context (i.e. distinguish whether the inspected list of values are required for regular or merge insert clause). We have tried to achieve the balance between these two approaches by explicitly defining top-level delimiting information in many different characters.

Example 1. queryId1 *[options1]* {*body1*}, queryId2 {body2}, queryId3, queryId4 [*options4*]

Example 1 shows four different structures of a query part. Elements written in italic are not really compilable token streams. They are just placeholders, though body placeholders can be considered as BoW query part. The first query part contains identifier section, options section and body section. The second part contains no options. The third one contains only an identifier: such query part must be cross-compiled from query libraries. The last part is cross-compiled from a library with overridden options. A query part can also omit both identifier and options.

The options contain the information about column caption (if it is not equal to identifier), Boolean filter override, sorting order, sorting priority and sorting boost (applies only to BoW).

The result of a query is a table. Every row in it is based on some object from the data model. A row is created if it satisfies conditions of all row filters and search criteria. A query part is treated as a BoW if it cannot be parsed neither as a row filter, nor as a computable expression. An object meets search criteria if every word in BoW is contained in parameter name or parameter string value. We dont use stemming and lemmatization for now.

If a sorting option is applied to a search filter, objects ranking R is computed as follows:

$$\begin{aligned} R(query, object) &= \sum_{pv} \frac{H(pv)}{N(pv)} \, , \\ H(pv) &= \sum_{i=1..N(query)} N_{gram}(query, pv, i) \, , \end{aligned} \tag{1}$$

Where pv is selected from all values of object parameters, function H computes a corresponding rank for each parameter value. This rank is computed as follows. Every possible consecutive sequence of query words, founded in pv, increase rank by 1 (function *Ngram* computes the number of occurrences of every n-gram from the query into pv). The calculated value are normalized by *N(pv)* – the number of unique terms in the query. Then the values for every pv are summed up in the rank R of the object.

All words in a query are considered equally important, so weight coefficients like TF-IDF [11] are not applied. This helps better prediction a behavior of the ranking model. Some weight tuning can be achieved through manual manipulation with a boost coefficient in the options. Term exclusion is also implemented by assigning a negative boost coefficient to unwanted terms.

However, the actual sorting is based not on the computed rank R, but on the ordering value O computing as follows:

$$sign\,(boost) \cdot [log_2\,(|boost| \cdot R)] \ . \tag{2}$$

Where [ ] is a function of rounding to ceiling integer value no less than zero, sign(*boost)* is a function that returns -1 for negative arguments and 1 otherwise, *boost* is a boosting value, specified in column options. Boosts are especially convenient when we combine different queries (with different boosts).

Taking the logarithm of a rank is introduced to provide a meaningful BoW parts combination. The real values of R for two different objects are rarely equal. Which means that R can be used for sorting without any other sorting search clauses.

Boolean filters and computable expressions can access the parameter values by surrounding parameter names in double quotes.

Example 2. {“type” = ’slab’}, [desc]{“creation time”}, {“Cr”}, {“V”}, {“Ni”}, {“Length” >“Width” / 2}, defective, today

Example 2 shows a query, which selects some chemical information for the newest slabs. Also slabs should be defective, created the same day when the query is executed and their length should be at least twice larger than width. These two restrictions of the query are extracted from libraries. The restriction on the creation date is context-dependent (addresses the current time). Cross-compilation can also create a context, i.e. a user can define his brigade number ("myBrigadeNumber{42}"), while common libraries use this number ("myBrigadeProduction{"operation brigade" = myBrigadeNumber }").

Parameters values can be extracted not only from the current row object, but also from referenced objects (with or without aggregation).

Example 3. Processes, melting, today, {"started"},
{"Id".Next("type"='slab').SumStr[,]}

Example 3 shows a query that extracts an information with identifiers of all slabs grouped by processes started today and related to melting. Forward and backward references can be used in a chain and embedded into referencing conditions.

Example 4. Processes, melting, today, {"started"},
{"Id".Next("type"='slab').Prev(type='bar' and "".Next.Next(
"state" = 'defective').Any).Count}

Example 4 shows the same set of processes as in Example 3, but calculates the total amount of different bars used in production of slabs of current process, which (bars) usage resulted in a defective sheet. This genealogical querying would result in an enormous amount of comboboxes and checkboxes in a visual query editor. If one would provide similar functionality.

5 Analysis

Designed DSQLM focuses mostly on data discovering and solving problems of metallurgical genealogy in data selection and preparation. Analytical capabilities (like SSA computation [6], regression and forecast analysis) are widely covered by a modeling system that is one of data consumers of this DSQLM. Although getting MDX-like pivot table using two sets of filter columns and an aggregation function could be implemented, this functionality was intentionally left undone. DSQLM was designed only to extract the relevant data.

DSQLM can aggregate data, but only by grouping on real objects in data warehouse. The language does not allow to set a limit on the rows in the result or to get a subset of retrieved objects for a query. These restrictions of DSQLM deters partial queries like "Top 5 selling mangers" in favor of queries like "All managers, ranged by sales" (financial information also is included in the system). However the system provides a viewer with the usual result paging functionality. Of course, a user can still achieve limit\offset constraints by tricking with context, but system does not encourage this programming style.

Creating a deterministic querying model with rich filtering possibilities is very challenging for indexing structures given that the estimated size of plant‘s raw operational data is measured in terabytes. Indexing questions will be covered in further papers.

The important thing is that ranking query part does not exclude objects from the result. To exclude objects not satisfying a query and order those by rank, the query should be included twice – with and without a sorting option. For example, a query “[desc]{ defective slab }”, will be based only on the objects mentioning “defective” or “slab”. While the result of a query “[desc]{ defective slab },[desc, boost 50]{ excess chromium }” will be based on the objects mentioning “defective” or “slab” or “excess” or “chromium”. To exclude objects without mentioning certain terms one have to add an extra filter of Boolean type.

Ranking features of the system tends to favor consecutive occurrences of query words in the data. Lets look on the following example. A rank H for query “{steel plant data}” and parameter value “meaningful steel plant data” is 6, while for parameter value “steel plant collecting data” its only 4.

Note that the ranking model also takes into account the density of query words in parameter values, so same occurrences of query words would give lower rank for a large text, than for a small one. The default *boost = 100* means that the minimum triggering term frequency is 1 term in 50 words of text. Ordering values O have large discreteness to avoid abuse of ranging features. Its not supposed to be used for retrieving just a few results satisfying queries. Instead of this the model should assist to retrieve the most relevant objects without limiting inspect window.

Sorting instructions can combine ranking criteria and computable functions. For example, there can be a query retrieving incidents sorted firstly by employees' last names, secondly by the rank of a query having a textual description of the incident, and finally by the amount of damage in monetary valuation.

6 Evaluation

A full-scale practical evaluation of DSQLM is yet to come along with the introduction of the next step of overall system in production. The language will be used to access systems data warehouse by plant chemical analysts. Their feedback will help us in further improvements of the DSQLM.

Nevertheless, we conducted two internal experiments. In our Department, we start to teach SQL and databases to sophomores. So we picked subjects from freshmen in order to simulate the plant employees who lack database knowledge but mastered a data types. We selected 10 such students.

The goals of the experiments were:

1. To evaluate an effort of dataset discovery with DSQLM
2. To evaluate an effort of deriving simple math expression with DSQLM

All subjects had been working separately and consequently. We explicitly asked them not to share any information about the experiment for the sake of evaluation clearance.

In the first experiment, the subjects were presented five rules defining data model (without Fig. 1 which contains significant details) and Examples 1-4 with notes. We asked them to discover dataset and state everything they understand about the data. The participants had 10 minutes to verbally describe their results. We expected them to mention all of the following concepts:

1. Steel, metal, melting or rolling
2. Production, process or output
3. Slab
4. Aggregate
5. Sheet

If a subject mentioned at least one word in each point we considered him mastered DSQLM. Nine of ten participants mastered DSQLM in allotted time and advanced next.

In the second experiment these nine subjects were given a formula and a text description of mean square deviation (MSD). We asked them to compute it for parameter value of a slab in each process. The subjects were forbidden to ask any additional information about DSQLM.

Five of nine subjects managed to derive required query.

The results of the first experiment showed that dataset discovery is effortless enough for production system. The results of the second experiment showed that expressiveness of DSQLM is acceptable for production system.

Subjects who mastered MSD query also noted that a sequence containing a parameter name, a reference path and aggregation function *("length".Next.Avg(. . .))* is not a very straightforward sequence of aggregation definition.

Our big concern regarding the upcoming real-world evaluations is an intentional high discreteness of the ranking. It was made to prevent overuse and provide deterministic, according-to-expectation ranging with use of multiple search clauses. However, this can spurn from usage those who are not familiar with information retrieval.

7 Related Work

One of the interesting works in the area of specialized query languages was published by Madaan and Bhalla [10]: they discuss difficulties of domain-specific querying of medical repositories. The key idea of replacing a visual query editor and a multistage text query with assisted input in our system is based on their work.

Tian et al. [15] presented NeuroQL a language for neuroscience data, which they compile to SQL and XQuery. This work contains a concept of query language compilation to SQL with segregation of DSQL data model from SQL data model. Our initial effort was to produce SQL tables reflecting domain model, but this work clearly stated that the scheme of database should not depend on changes in our domain specific data model.

Baratis et al. [3] presented Temporal Ontology Querying Language (TOQL) that aims to reduce the complexity of temporal ontology databases at least to the level of SQL. This work provided a fundamental background to address the main problem of metallurgical data. This data is aligned to oriented production graph, like TOQL is aligned to time. This particular feature of metallurgical data resembles a temporal database and is very familiar to every potential user of our DSQL.

BimQL [12] project is also of certain interest. They designed a language in order to help civil engineers with retrieving an information from construction models. The idea of querying objects of different types in the same table output was taken from BimQL. Also BimQL has an amazing concept of the result visualization that is natural for its domain. This particular feature is yet to be implemented in our DSQLM.

Development of DSQL would be much harder without LL-parsing by Parr [13] (also known as ANTLR). An alternative solution could be Fords packrat parsing [4] (also known as PEG). The initial DSQLM implementations used adaptive grammars, but this feature did not add clearance to DSQLM syntax. DSQLM grammar was developed with ANTRL 4, context free grammars shaped the way DSQLM is expressed.

8 Conclusion

In recent decades, querying languages and conventions have become widespread: search engines provide a wide range of search operators [11], IDEs have a lot of features available through built-in languages [8], genetic databases are even capable to help researchers to express which genome they are looking for [14]. There are numerous domains where specific querying languages can help to deal with specific problems. SQL, once created to rule them all, addresses generic problems, and it has been doing this well.

Maybe new kinds of structured user input superior to keyboard will end all text programming languages efforts at once. Currently it is certain: there is no better way to get a query than ask a user to type it. Neither gesture, nor speech recognition shows significant results in, for example, creating programs. Even if we would read mind somehow, there is no guarantee it will change a coding style.

In this article, we described the key ideas of our domain specific language that is not strongly connected with steel production. Described ranking model shows promising results for our domain, however it can be also applied for other domains as well.

Acknowledgment. Research is conducted under the terms of contract No. 02.G25.31.0055 (project 2012-218-03-167) with financial support by Ministry of Education and Science of the Russian Federation.

References

1. Aksyonov, K., Bykov, E., Aksyonova, O., Antonova, A.: Development of real-time simulation models: integration with enterprise information systems. In: ICCGI 2014, The Ninth International Multi-Conference on Computing in the Global Information Technology, pp. 45–50 (2014)
2. Ambler, S.W.: The object-relational impedance mismatch (update of February 15, 2006)
3. Baratis, E., Petrakis, E.G.M., Batsakis, S., Maris, N., Papadakis, N.: TOQL: Temporal ontology querying language. In: Mamoulis, N., Seidl, T., Pedersen, T.B., Torp, K., Assent, I. (eds.) SSTD 2009. LNCS, vol. 5644, pp. 338–354. Springer, Heidelberg (2009)
4. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl. ACM SIGPLAN Notices 37(9), 36–47 (2002)
5. Fowler, M.: Language workbenches: The killer-app for domain specific languages (2005)
6. Golyandina, N., Osipov, E.: The "caterpillar" – ssa method for analysis of time series with missing values. Journal of Statistical Planning and Inference 137(8), 2642–2653 (2007)
7. Harris, Z.S.: Distributional structure. Word (1954)
8. JetBrains: Searching through the source code, http://www.jetbrains.com/idea/webhelp/searching-through-the-sourcecode.html
9. Kim, W.: Introduction to object-oriented databases, vol. 90. MIT Press, Cambridge (1990)
10. Madaan, A., Bhalla, S.: Domain specific multistage query language for medical document repositories. Proceedings of the VLDB Endowment 6(12), 1410–1415 (2013)
11. Manning, C.D., Raghavan, P., Schütze, H.: Scoring, term weighting and the vector space model. Introduction to Information Retrieval, pp. 109–133 (2008)
12. Mazairac, W., Beetz, J.: Towards a framework for a domain specific open query language for building information models. In: Proceedings of the 2012 eg-ice Workshop (2012)
13. Parr, T., Fisher, K.: Ll (*): the foundation of the antlr parser generator. ACM SIGPLAN Notices 46(6), 425–436 (2011)
14. Stypka, Ł., Kozielski, M.: Methods of gene ontology term similarity analysis in graph database environment. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B. z. (eds.) BDAS 2014. CCIS, vol. 424, pp. 345–354. Springer, Heidelberg (2014)
15. Tian, H., Sunderraman, R., Calin-Jageman, R.J., Yang, H., Zhu, Y., Katz, P.S.: NeuroQL: A domain-specific query language for neuroscience data. In: Grust, T., Höpfner, H., Illarramendi, A., Jablonski, S., Fischer, F., Müller, S., Patranjan, P.-L., Sattler, K.-U., Spiliopoulou, M., Wijsen, J. (eds.) EDBT 2006. LNCS, vol. 4254, pp. 613–624. Springer, Heidelberg (2006)