A Tool for Automated Analysis of Database Design

Preface

This master thesis is the result of the specialization year in the Database and Programming Technology group at the Department of Computer Science at Aalborg University, written by Morten Bested, Benjamin Krogh and Andreas Weisberg.

The topic of this thesis is ensuring quality and consistency of database designs using automated evaluations and techniques.

The thesis builds upon our experiences from our SW9 project in which we constructed a preliminary prototype. Here we unify the experiences from the two semesters. Most of the work in this thesis is original. However, the following sections are updated or rewritten from our SW9 project. The related work in Section 2 is updated, the architecture in Section 5.1 is rewritten, the report examinations of phpBB and PrestaShop in Section 6.2.1 and 6.2.2 are rewritten and the evaluation of the naming convention analysis in Section 6.3 is updated.

The CD attached to this thesis contains the tool, relevant reports, and a readme file.

DBLint: A Tool for Automated Analysis of Database Design

Morten Bested, Benjamin Krogh, Andreas Weisberg

Department of Computer Science,
Aalborg University, June, 2011

ABSTRACT

Evaluating the quality and consistency of a database schema by a manual review is time-consuming and error-prone. To accommodate this challenge, we propose *DBLint*, a fast, configurable, and extensible tool for automated analysis of database design. *DBLint* is fully implemented and includes 46 design rules derived from good database design practices. The rules discover design errors, which are collected as issues and presented in an interactive report. The issues are used to calculate a score for each table and an overall score. The scores are based on the severities of the issues, their location in the schema, and a table-importance measure. *DBLint* has been tested extensively on more than 35 real-world schemas, identifying a large number of relevant issues. Developers from four organizations have evaluated *DBLint* and found it to be useful and relevant, in particular the overall score and report. To the best of our knowledge, *DBLint* is a significant improvement over existing schema-checking tools.

1. INTRODUCTION

A good database design is a fundamental requisite for achieving good data quality [6], thus it is important to put effort into the design of a database. However, maintaining the quality of an evolving database schema is a difficult challenge. If the schema is maintained by different developers, and there are no clear agreements on design principles, the quality of the schema design may degenerate over time. Possible issues that arise throughout development are missing foreign keys, different data types between columns in foreign keys, and redundant indices. As the schema size increases, the challenge of ensuring the quality by manual reviewing becomes both time-consuming and error-prone.

This paper describes *DBLint*, an automated and DBMS independent tool for analyzing database designs. The main purpose of *DBLint* is to ensure consistency and maintainability of database designs by identifying bad design patterns. These patterns are expressed as rules that analyze a schema's metadata and the data stored in the database. *DBLint*'s main output is an interactive report containing a list of all the discovered design issues.

To illustrate many of the pitfalls facing a database designer, consider the MySQL schema in Figure 1 containing the two tables `users` and `posts`.

The *DBLint* tool detects 15 design issues in this example, several non-obvious. The issues reported are the following, grouped by severity.

```
create table users (
 user_name           varchar(32),
 id                  varchar(32) primary key,
 email1              varchar(32),
 email2              char(32) default '',
 last_visited_post   int,
 `msn#`              varchar(0));

create table posts (
 date                datetime,
 post_id             int unique,
 subject             varchar(31),
 postText            varchar(1500),
 user_id             varchar(10),
 index ix_id (post_id),
 index ix_id_subject (post_id, subject),
 foreign key (user_id)
   references users(id));

alter table users add
 foreign key (last_visited_post)
   references posts(post_id);
```

Figure 1: A database design with multiple issues.

Critical A table without a primary key, a varchar column of length zero, and different data types between a source and a target column.

High A char column with the empty string as default value, and columns not following the naming convention.

Medium Different data types in a sequence of columns (`email1` and `email2`), inconsistent maximum length of varchar columns, a redundant index, too many nullable columns in the same table, and a nullable and unique column.

Low An identifier containing a special character, a varchar column with too large maximum length, a cyclic dependency between the two tables, a column named with a reserved word from SQL, and a primary-key column not positioned first.

DBLint is envisioned to be used by developers to quickly catch common design errors, e.g., as part of a unit-test suite. Analyzing schema metadata is very fast and can be performed quickly to verify the design before deployment. A developer team can control the quality of a schema by expressing their design principles as rules in *DBLint*. Furthermore, *DBLint* can be used in a training environment to help

newcomers to understand and avoid common design errors. *DBLint*'s data analysis can be used on deployed databases to find data quality issues that arise when the system is used.

The three main principles of *DBLint* are: (1) Low configuration, which ensures a low initial cost of using *DBLint*. (2) Domain-independence, which ensures general applicability. (3) Rule extensibility, which enables *DBLint* to be adapted to specialized environments. An example of a low-configuration and domain-independent rule is the naming-convention checker that automatically discover the convention used and detects inconsistencies.

We introduce a metric score that summarizes the overall quality of a schema. The score is based on the design issues reported and a table-importance measure. The score can be used to compare different schema versions or design alternatives. *DBLint* also assigns a score to each table, such that the most problematic parts of the schema can be easily identified.

DBLint has been evaluated on 35 real-world schemas, of which 14 widely used schemas are compared. Three of these schemas have been examined thoroughly, and it is verified that *DBLint* reports a large number of relevant issues. This proves the need for a static analyzer to help the database designer develop a consistent and maintainable database design. Furthermore, *DBLint* has been tested by four developer teams with very positive results. The developers found particularly the database score to be a motivational factor that encourages team members to compete on quality.

To summarize, the main contributions of this work are the following.

- A bundled collection of 46 design rules, experimentally validated to be relevant and generally applicable.
- A pluggable and extensible rule system.
- An overall score, rating database designs between 0% and 100%.
- A comparison of 14 real-world and widely used schemas.

The paper is organized as follows. In Section 2, the related papers and tools are described. In Section 3, the design rules are presented together with a discussion about the difference between metadata rules and data rules. Furthermore, three of the design rules are presented. In Section 4, the approach for assigning the database design scores is presented. In Section 5, the architecture of the system is presented together with discussions about interesting parts of the system, such as DBMS independence, data extraction and rule execution. In Section 6, *DBLint* is tested with respect to the relevance of issues and with respect to performance. Furthermore, a comparison of 14 real-world database designs is discussed. In Sections 7 and 8, we discuss and conclude upon the findings of this work.

2. RELATED WORK

DBLint is a tool, and it is thus relevant to discuss the other tools that *DBLint* is inspired by. We have not found any academic paper that thoroughly addresses automatic diagnosis of database design. However, attempts have been made to quantify what a good data model is. These are relevant even though the approaches taken are different than ours. Following are two sections describing the related tools and related academic papers.

2.1 Tools

The main source of inspiration is a number of existing tools from other domains. The original Lint program was written to detect bugs and obscurities in C programs [12]. Lint-like tools have since been made for many other programming languages such as Python [27] and Java [15]. The idea of reporting obscurities or inconsistencies through static analysis is very similar to ours, except that *DBLint* is for a different domain. The original Lint examines source code, while *DBLint* examines database schemas. The relation between Lint and *DBLint* becomes clearer when considering that a DBMS is comparable to a compiler in that it captures errors, but does not care about inconsistencies or other bad design decisions. FindBugs [1], a Lint-like tool for Java, has provided inspiration on how to build a system that supports extensible rules, or "bug patterns" in FindBugs jargon.

SchemaSpy [7], SchemaCrawler [8], SQL Auditor [31] and Database Examiner [30] are all tools for analyzing schemas. SchemaSpy and SchemaCrawler are open-source tools that provide limited Lint-like functionality for schemas. The core features of these are extraction and presentation of database metadata, and they do not provide a thorough analysis of the database's design. SQL Auditor and Database Examiner are commercial tools with a purpose similar to that of *DBLint*, but *DBLint* differs in five main areas. (1) An extensible rule system that allows quick development of additional rules. (2) Automatic consistency checks, e.g., a naming convention rule. (3) Combined metadata and data analysis. (4) A scoring system rating database designs. (5) A table-importance measure. Furthermore, SQL Auditor supports only SQL Server, and Database Examiner supports only enterprise DBMSs. *DBLint* supports four widely used DBMSs: MySQL, Oracle, PostgreSQL, and SQL Server.

2.2 Academia

An effort has been put into the development of methods that quantifies the quality of a data model or a relational schema using a set of metrics. [25] and [5] propose and evaluate three metrics for estimating the maintainability of relational schemas. These are simple measures such as the number of attributes and foreign keys in the schema. Compared to our work, we give a broader estimation of quality as we include the result of 46 rules.

[21] has identified 25 metrics for evaluating the quality of a schema for quality factors such as understandability, correctness, and implementability. However, the metrics rely on manual evaluation as most of these cannot be measured automatically. An example of such a metric is the number of user requirements not represented in the data model. To calculate this number, a complete set of requirements must be available and evaluated against the data model manually. The strength of the manual approach is that it is possible to identify design problems that are hard to find automatically. However, our focus has been to develop a tool that evaluates a database design automatically. We regard the two approaches as complementary because they identify different design problems.

[32] presents a tool for validating schemas in SQL Server. Schemas are verified according to properties such as non-redundant integrity constraints. The number of properties verified is limited, but similar checks could be adopted in *DBLint*.

3. DATABASE DESIGN RULES

The rules in *DBLint* focus on the design of the database and are based on the following sources.

Good practices [29] and [6] describes good practices in designing a database, such as: a primary key on each table, check constraints enforcing data integrity, and a consistent naming convention. [16] discuss the importance of agreeing on design conventions and practices in programming. The concept from [16] can be applied to database designs, where a similar agreement for a database design enhances the quality of the final result.

Antipatterns [13] describes several antipatterns that should be avoided, e.g., storing lists in varchar columns and unnecessary indices.

Experience Through interviews with experienced database developers and extensive analysis of many existing database designs, several obscurities have been identified, such as self-referencing primary keys and different data types between source and target columns in foreign keys.

The severity of the design rules are different because some rules find issues that indicate problems with the data integrity, while others just indicate small deviations in the design. To accommodate these differences the design issues are categorized into four severity levels: low, medium, high and critical. The severities are used to categorize and arrange the issues in the report, and to calculate the scores. The four severity levels are defined as follows.

Critical Used on issues potentially leading to compromised data integrity.

High Used on issues indicating a design that deviates significantly from good database design practice.

Medium Used on issues that contradicts good design practice.

Low Used on issues indicating minor inconsistencies and issues not influencing the integrity of the data.

3.1 Metadata and Data Analysis

This section describes the possibilities when analyzing metadata compared to analyzing the actual data. Metadata analysis gives access to the structure of the database. This allows *DBLint* to analyze how the tables are related, what keys and indices the tables have, how data types are used, etc. This information forms a good base for consistency checks of the database design.

When including data from the database into the analysis, many new possibilities are introduced. These possibilities include simple data type checks, such as analyzing varchar columns for string encodings of other data types, and more complex analyses such as finding outlier data.

Metadata analysis can be performed at any time, hence it is useful in a development process. To perform a meaningful data analysis the database must contain real data. This means that the data analysis is not suitable in development, but can be used when the database is in production.

There is a performance aspect when considering the differences between metadata and data analysis. Metadata analysis can be performed in seconds, whereas data analysis is much more time-consuming, depending on the size of the database.

3.2 Rule Overview

DBLint has 46 rules: 27 metadata rules, listed in Table 1; and 19 data rules, listed in Table 2. The rightmost columns in the two tables show the severity for each rule, ranging from critical (C) to low (L). An argumentation for the relevance of each metadata rule can be found in Appendix A, and for each data rule in Appendix B.

#	Metadata Rule	S
1	Missing Primary Keys	C
2	Different Data Type Between Source and Target Columns in a Foreign Key	C
3	Varchar Columns of Length Zero	C
4	Inconsistent Naming Convention	H
5	Inappropriate Length of Default Value For Char Columns	H
6	Redundant Foreign Keys	H
7	Table With Too Few Columns	H
8	Too Big Indices	H
9	Too Many Nullable Columns	H
10	Too Long Column Names	M
11	Nullable and Unique Columns	M
12	Cycles Between Tables	M
13	Inconsistent Max Lengths of Varchar Columns	M
14	Self-Referencing Primary Key	M
15	Inconsistent Data Types in Column Sequence	M
16	Missing Column in a Sequence of Columns	M
17	Primary- and Unique-Key Constraints on the Same Columns	M
18	Redundant Indices	M
19	Too Short Column Names	M
20	Too Many Text Columns in a Table	M
21	Foreign-Key Without Index	L
22	Primary-Key Columns Not Positioned First	L
23	Use of Reserved Words From SQL	L
24	Different Data Types for Columns With the Same Name	L
25	Use of Special Characters in Identifiers	L
26	Table Islands	L
27	Too Large Varchar Columns	L

Table 1: Metadata Rules.

3.3 Rule Examples

To demonstrate the wide range of rules in *DBLint*, three rules are described. The first is a simple rule to get started, and the following two are more advanced rules.

A common trait of the three rules is that they focus on consistency. The first on ensuring consistency among maximum length of varchar columns, the second on ensuring that data in the database is uniform, and the third on ensuring that the naming convention for identifiers is used consistently. Additionally, an implementation of the rule "Nullable and Unique Columns" is given in Appendix F.

3.3.1 Inconsistent Max Lengths of Varchar Columns

One of the metadata rules that help keeping the design consistent, is the rule that examines all varchar columns,

#	Data Rule	S
28	Duplicate Rows in a Table	C
29	Storing Lists in Varchar Columns	C
30	Wrong Representation of Boolean Values	C
31	Defined Primary Key is not a Minimal Key	H
32	Redundant Columns	H
33	All Values Equals the Default Value	H
34	Not-Null Columns Containing Many Empty Strings	H
35	Numbers or Dates Stored in Varchar Columns	H
36	Empty Tables	M
37	Mixture of Data Types in Text Columns	L
38	Columns With Only One Value	L
39	All Values Differ From the Default Value	L
40	Inconsistent Casing of First Character in Text Columns	L
41	Unnecessary One-to-One Relational Tables	L
42	Column Values from a Small Domain	L
43	Large Unfilled Varchar Columns	L
44	Missing Not-Null Constraints	L
45	Column Containing Too Many Nulls	L
46	Outlier Data In Column	L

Table 2: Data Rules.

and compares the maximum lengths to find columns with small deviations.

Consider the example in Figure 1. The example contains four varchars columns of length 32, and one of length 31. The column of length 31 is a deviation from the majority of varchar columns and is therefore considered a design issue. This rule finds issues in many real-world schemas, e.g. Drupal, where 78 columns are of length 255, while 2 columns are of length 254. Small deviations in the column definitions such as these, may cause misunderstandings between developers.

This rule is a good example of a simple rule that is capable of analyzing large schemas to find inconsistencies. The same analysis could be done by hand, but would be more time consuming.

3.3.2 Outlier Detection

An interesting data analysis is detection of values that seems to be deviating from the general data in the database. Such values can be characterized as outliers, based on the definition from [11]. The intuition is that outlier data indicates missing check constraints or data integrity problems. Examples of this are: city names containing numbers, and dates that are much different from the majority of dates.

Many of the approaches for detecting outliers are based on knowledge about the data domain, and what possible kinds of outliers the data may contain. Because *DBLint* is domain independent, these approaches are insufficient. [14] and [3] present techniques for performing outlier detection based on distance measures in a multidimensional space. Such a solution requires the data in the database to be converted into points, which can be done for columns of any type. Using a multidimensional space makes it possible to include many different properties of the data, such as string length and word count.

In [14] an outlier is determined using a global view on the data set. An object is said to be an outlier if a certain fraction p of the other objects in the data set lies greater than distance D from the examined object. Because this approach uses a global view there are several limitations. In Figure 2, the strategy from [14] will only be able to detect object o_1 as outlier, because of the short distance from object o_2 to cluster C_2. To use this approach the variables p and D must be specified in advance, making the approach less suitable, because p and D relies heavily on the domain. [3] presents an approach that requires less configuration before the analysis, and does not result in a binary value specifying if an object is an outlier or not. Instead [3] introduce the term Local Outlier Factor (LOF) that gives each object in the data set a degree of being outlier, based on its nearest neighborhood. This means that in Figure 2 object o_2 is also determined to be an outlier because its neighborhood is a dense cluster that it is not a part of. An advantage of [3]'s approach is that the analysis is not bounded by a constant distance to determine outliers. However, the analysis needs to know the size of the neighborhood.

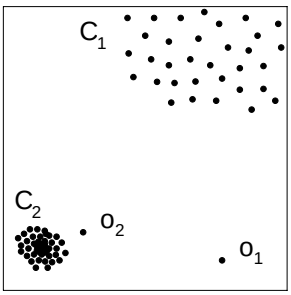


Figure 2: Data set containing two clusters, C_1 and C_2. There are two outliers, a global outlier o_1 and a local outlier o_2. The figure is based on [3].

The approach taken in *DBLint* is to represent the data as points in a multidimensional space and then apply LOF to find outliers. The capability of finding local outliers is a clear advantage in the case of *DBLint*, as it is applicable to unknown data.

Finding Outlier Data with DBLint. When analyzing the data, each column is analyzed individually. This is done to limit the scope of the analysis and because it is difficult to correlate columns without prior knowledge.

To be able to use the data it must be converted into a point representation. Many different approaches can be used to convert a value into a point. For instance, a string can be converted based on its: length, number of words, average word length, and the number of numeric values. This conversion can be seen as an abstraction of the data.

The analysis is performed table-wise, with a single scan over all rows in the table. The scan constructs an R-tree [10] for each column that is used in the LOF analysis. After the analysis, all values with a LOF larger than a configurable threshold are reported as outlier data.

3.3.3 Naming Convention Rule

A consistent naming convention is an important feature of a maintainable database design [29]. However, manually examining all identifiers to find inconsistencies is a time-consuming process, and for this reason, *DBLint* includes a rule for checking the consistency of the naming convention.

The rule is designed such that it automatically discovers the convention used and reports inconsistencies. The naming convention rule is a good example of a zero-configuration, domain independent, and non-trivial design rule.

Automatic detection of a naming convention is non-trivial because it involves aspects such as consistent use of domain specific terms, syntax, and abbreviations. To create a domain-independent, zero-configuration rule, the focus is limited to casing, word separators, and symbols.

The idea is that identifiers adhering to a naming convention, can be described by a grammar similarly to the grammars used in programming languages. These grammars usually consist of production rules that uses tokens as well as other production rules. *DBLint* does not know which grammar is used, and therefore needs a method of building the grammar from identifiers. To simplify the problem, the first step in the approach is to tokenize an identifier into a sequence of predefined token types. The token types used in *DBLint* are: `begin`, `lowerword`, `underscore`, `uppperchar`, `symbol` and `end`. The intuition behind using these tokens is that similar identifiers produce near identical sequences of tokens. For instance, tokenizing either of the identifiers `post_id` and `user_name` from the example in Figure 1, yields the following sequence of tokens: (`begin`, `lowerword`, `underscore`, `lowerword`, `end`).

The second step in the approach is to extract the naming convention from the lists of tokens. *DBLint* is able to extract and represent a naming convention in two different ways. Both methods use the tokenized identifiers, and have been tested with similar results. The two approaches are described in the following sections.

Markov-Chain Representation. The first approach uses a first-order Markov chain [28] to represent the naming convention. A state represents a token and a transition represents the probability of going from one token to another. Figure 3 shows an example of a Markov-chain representing a consistent naming convention of lowercase words separated by underscores, e.g., `user_name`.

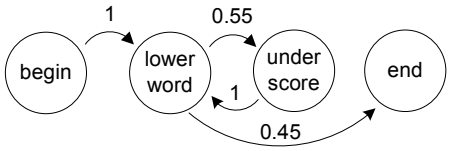


Figure 3: A consistent naming convention. Used in MediaWiki, Drupal, and Magento.

The Markov-chain representation of the naming scheme is used to locate identifiers not following the main convention. Intuitively, an identifier not adhering to the convention will at some point follow a transition with a low probability. This is illustrated in Figure 4. The inconsistency is indicated by the transition with low probability from `lowerword` to `upperchar`. Identifiers following such low-probability transitions are reported as issues in *DBLint*.

Trie Representation. The second approach to represent a naming convention is a trie, also known as a prefix tree. A trie usually contains strings, but in this case it contains tokenized identifiers. Figure 5 shows a trie for the same schema as in Figure 4. Lowercase words are represented by `w`

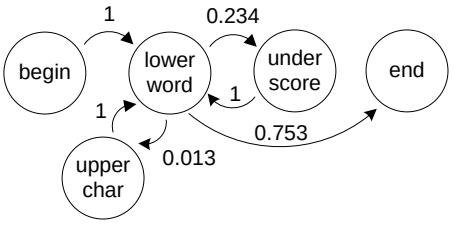


Figure 4: Markov-chain representation of an inconsistent naming convention. This example is from Joomla.

and uppercase characters are represented by `C`. The numbers indicate how many identifiers follow a specific path.

The trie is used to find the most common naming convention by following the path with highest numbers. In Figure 5, the naming convention would be `w_w`, that is a lowercase word followed by an underscore, followed by a lowercase word. Identifiers following a different path in the tree, such as `registerDate`, will be reported as issues.

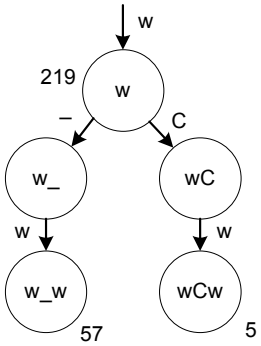


Figure 5: Trie representation of an inconsistent naming convention. This example is from Joomla.

The advantage of a Markov chain over a trie is that a Markov chain can represent naming conventions of infinite length. The advantage of a trie over a Markov chain is that a trie can model more complex naming conventions, such as every other word being uppercase. *DBLint* defaults to the Markov-chain representation, but can be configured to use the prefix tree.

3.4 Table Importance

A property of most schemas is that some tables are more important than others [33, 34]. Knowing which tables are the most important is useful in several places in *DBLint*, and thus methods of identifying these tables will be an improvement over assuming equality of all tables.

An important table have the following traits: it has a high connectivity in the schema graph, meaning that many other tables are connected to the table [34]; it has a high cardinality [34]; the entropy of the table is high [33].

DBLint uses two approaches for approximating importance: the PageRank algorithm [4] on the graph defined by foreign keys; and the table data importance metric described in [33] that in addition to foreign keys also considers the data stored in the tables. The mechanism used depends on the configuration of *DBLint*: if data is to be analyzed, *DBLint* uses table data importance, otherwise PageRank.

Both approaches are normalized such that the sum of all tables' importance is 100. The table importance metric is primarily used by *DBLint* in the following areas.

Naming Convention A naming convention is derived from a list of identifiers, as described in Section 3.3.3. The influence that each identifier has on the naming convention is determined by the importance of the table, such that important tables contribute more to the convention than peripheral tables.

Scoring The scoring system, as will be described in Section 4, weights design issues such that an issue in an important table is considered more severe. This is based on the assumption that changes or errors are likely to propagate to other tables. For instance, a primary key with an inappropriate data type is less severe for a peripheral table, compared to an important table where the primary key is referenced by many tables.

3.4.1 The PageRank Algorithm

With only metadata available, the approximation of table importance is based solely on foreign keys. Interpreting a schema as a directed graph, makes it possible to use the PageRank graph centrality algorithm [4] for computing a ranking of each vertex. Specifically PageRank calculates the relative likelihood of visiting a given vertex by randomly following edges, and sometimes jumping to a random vertex in the graph. PageRank was originally developed for estimating the relative importance of web-pages based on the link graph. Though PageRank is a centrality approximation algorithm, it is used to estimate the importance of tables in *DBLint*. The intuition is that a central table is also likely to be an important table, because other tables will depend on it.

PageRank assigns a numerical weight to each vertex in a directed graph based on the weights of its neighbors. The weight of a vertex, i.e., its PageRank, is calculated through a number of iterations. Each vertex, with m outgoing edges, gives in each iteration $\frac{1}{m}$ of its PageRank to each connected vertex. It terminates when the difference between the weights from two iterations is lower than a predefined constant.

A fundamental assumption of PageRank is that a website u linking to another website v corresponds roughly to u saying that v is important. In the context of *DBLint*, this is also assumed to be true, such that a table f with a foreign key to table p, corresponds to f saying that p is important. This is based on the fact that a foreign-key ensures an inclusion dependency relationship, yielding a strong relation between the two tables.

For PageRank to be useful on a database, it is assumed that most foreign-keys are specified in the schema. If many foreign keys are not specified this approach will degenerate, until each table has a rank of $\frac{1}{n}$, n being the number of tables. We have observed a lack of foreign keys in some schemas, and thus some cases in which PageRank does not yield useful information.

3.4.2 Table Data Importance

Using both metadata and data makes it possible to use the approach described in [33], from now on referred to as Table Data Importance (TDI). TDI is a more suitable approach than PageRank, because PageRank has some problems with dimension tables such as `zip_codes` and `status_types` receiving too high rating, and many-to-many relation tables receiving too low because they have no ingoing edges. TDI solves these problems by (1) reading foreign-keys as undirected, and (2) assigning weights to foreign-key relationships according to the information content of the attributes in the relation. For instance, consider a schema with the two tables `users` and `zip_codes`, and a foreign-key from `users` to `zip_codes`. PageRank would not assign much importance to the `users` table due to no ingoing foreign-keys, whereas TDI would transfer importance from `zip_codes` back to `users` again. For a more in depth description of this algorithm please refer to [33].

Without foreign keys, TDI assigns a weight relative to the information content in the table, hence if there is data in the database it still yields a useful result, compared to PageRank. A negative aspect of TDI is that it requires a full scan of all data, which makes this algorithm orders of magnitude slower than PageRank.

4. SCORING

DBLint provides a metric that summarizes how well a database design follows the rules. An overall database score is given as a number between 0% and 100%. A higher score indicates a better design. A score for each table is also given, such that the most problematic parts of the database can be identified. Database designers will benefit from a score in the following ways.

- The score reveals the overall state of the schema. This information can be used to determine whether more work needs to be put into the database design in general.
- Different schema versions or design alternatives can easily be compared using the scores. For instance, the database designer can see whether a number of schema changes have improved the overall quality of the schema or degraded it. An example of this can be seen in the transition from Drupal 6.20 to 7.0, in which the score decreased from 60% to 53%.
- The team can use the score to agree on a minimum acceptable score for tables. This is an easy way to assure quality of tables when extending or modifying the database.
- A score for each table reveals the most problematic areas of the database design. This can be used to direct the focus onto the tables that contribute most negatively to the overall score.
- The team can use the score as a motivational factor or to encourage competition between developers. This was observed when evaluating *DBLint* in developer teams.

The score is not a complete and correct measure of quality. The purpose of rules is not to asses quality, but to find design issues. Therefore, a low score should primarily be seen as an indication of the schema having quality issues.

4.1 Score Calculation

DBLint uses a bottom-up approach for calculating the scores. A database can be considered a hierarchy or a tree with the database as root, and schemas, tables, and columns, as the second, third and fourth level of the tree, respectively. The score of either a database, a schema, or a table is calculated by aggregating the scores of the nodes below it. For instance, the score of a schema is based on the scores of its tables, and the score of a table is based on the scores of its columns. As a consequence, *DBLint* calculates an overall score for the database as well as a score for each schema, table, and column.

All scores are based on the issues found by the rules. A database with many critical issues will generally score less than a database with only a few low-severity issues. All issues specifies where in the schema the problem is located, e.g., column x uses a special character in the identifier or table y does not have a primary key. In *DBLint*, this is called an issue context, and issues affect only the score of the context in which they are found. The argument for this is that an issue on only a small part of the schema, e.g., a column, should not have as much impact on the overall score compared to an issue relevant for the whole schema, e.g. no foreign keys in database. The former issue is called a column-level issue and the latter is called a schema-level issue. The impact that each issue has on the score of its context is determined by its severity.

The score is chosen to be a number between 0% and 100%, to make it understandable to the user and comparable across databases. To achieve this, the scores in *DBLint* follow an exponential decay curve, illustrated in Figure 6. The figure shows the relationship between the total penalty given for issues and the score. The scoring functions are described below.

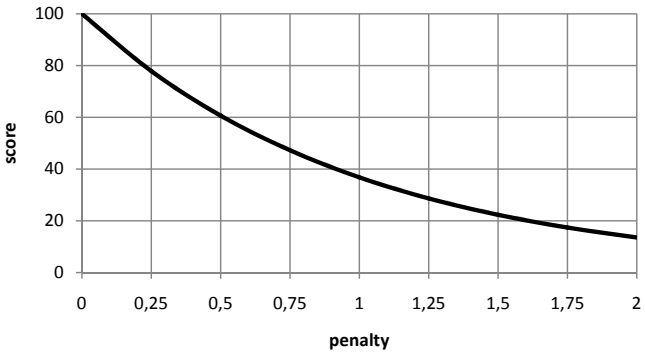


Figure 6: The eksponential decay curve used for calculating scores.

Column Score. A column starts with a 100% score, that is reduced for each column-level issue. The column scoring function for at column c is defined as follows.

$$score_c(c) = 100e^{-p_c(c)}$$

where

$$p_c(c) = \sum_{iss \in c} penalty_c(iss)$$

Each issue is assigned a penalty, determined by the function $penalty_c$. The function p_c summarizes the penalties of all issues in the given column.

Table Score. The table scoring function calculates an average of its columns, and reducing table-level issues from this average. The table scoring function for a table t is defined as follows.

$$score_t(t) = \frac{\sum_{c \in t} score_c(c)}{|t.columns|} \cdot e^{-p_t(t)}$$

where

$$p_t(t) = \sum_{iss \in t} \frac{penalty_t(iss)}{|iss.tables|}$$

$|t.columns|$ is the number of columns in table t, while $|iss.tables|$ is the number of tables sharing issue iss. The intuition behind the formula is that since a table consists of columns, the score should reflect the quality of these. The maximum score a table can get is the average of the score of its columns. This average is further reduced by table level issues, e.g., a missing primary key.

The penalty of an issue is divided if the issue is shared between multiple tables. For instance, if there is a cyclic dependency between two tables, the two participating tables shares the penalty.

Schema Score. The score of a schema is calculated by taking the weighted average of its tables, and reducing this average for each schema-level issue. The schema scoring function for a schema s is defined as follows.

$$score_s(s) = \frac{\sum_{t \in s} score_t(t) \cdot w(t)}{\sum_{t \in s} w(t)} \cdot e^{-p_s(s)}$$

where

$$p_s(s) = \sum_{iss \in s} \frac{penalty_t(iss)}{|iss.schemas|}$$

The weight of a table reflects its importance as described in Section 3.4, meaning that an important table will contribute more to the score than a peripheral table. The argument for this is that important tables are expected to be queried often, referenced by other tables, or store a lot of data. Therefore, such tables should have a higher influence on the score than other tables. The table weight function is defined as follows.

$$w(t) = max(1, \sqrt{t_{imp}})$$

The table importance t_{imp} is a number between 0 and 100, provided by the table-importance algorithm. To prevent some tables from having too much impact on the score compared to others, the square root of the table importance is used. This problem becomes evident when considering the schema for a system such as the open source ERP system Openbravo. Using PageRank on this schema, approximately 1 percent of the tables receive 75% of the importance. The most important table receives an importance value three orders of magnitude larger than a peripheral table without

incoming foreign keys. While the weight should reflect table importance, we do not believe that using the importance value directly is a fair and well-balanced approach. Therefore, we compute the square root of the importance value to prevent important tables from having too much influence. A minimum value of 1 prevents peripheral tables from having too little influence.

Database Score. The overall database score is calculated by averaging the score of its schemas, and reducing this average for each issue at the database level. The scoring function for a database d is defined as follows.

$$score_d(d) = \frac{\sum_{s \in d} score_s(s)}{|d.schemas|} \cdot e^{-p_d(d)}$$

where

$$p_d(d) = \sum_{iss \in d} penalty_s(iss)$$

Penalty Functions. The penalty functions used in the formulas above take an issue as input and returns a number, depending on the severity of the issue. A penalty function decides how rapidly a score approaches zero. The penalty function for schemas is given below.

$$penalty_s(iss) = \begin{cases} 0.29 & \text{if severity(iss)} = \text{critical} \\ 0.22 & \text{if severity(iss)} = \text{high} \\ 0.16 & \text{if severity(iss)} = \text{medium} \\ 0.10 & \text{if severity(iss)} = \text{low} \end{cases}$$

The constants defined in this function are determined through an empirical study of the 14 schemas in Figure 13, such that the average score of all databases in the data-set is 50%. This means that if the overall score of a database is larger than 50%, the database design is considered above average. Similar penalty functions for tables and columns are shown in Appendix C.

5. SYSTEM OVERVIEW

This section describes the architecture of *DBLint* and the responsibilities of the individual components. Furthermore, this section addresses the challenges of: being DBMS independent, extracting data from the database, and how to handle the scheduling of pluggable rules. Finally, the interactive report is described.

5.1 Architecture

DBLint has a layered architecture, shown in Figure 7, that divides the system's responsibilities into four low-coupled layers. The result is a flexible and maintainable architecture without ties to any concrete DBMS. The architecture is a non-strict layered architectures in that the `UI Layer` uses components from all three layers below it.

DBLint supports four common database systems, with minimal DBMS-specific code. New DBMSs can be supported by implementing an interface, effectively encapsulating all DBMS specific code. The rule system uses a plug-in architecture that decouples the rules from the rest of the system. This decoupling makes the system highly extensible and maintainable.

The architecture of *DBLint* is divided into two parts: `DBLint` and `Data Sources`. Data sources are external databases from which the `Extractor` retrieves the metadata and the data, which rules are checked against.

The boxes in `DBLint` on the figure represents separate layers. Each layer is further divided into sub-components such as `Controller` and `Model Builder`. As shown, rules are not considered a part of the *DBLint* core. Instead, the rules are loaded at run-time, represented by an arrow on the figure. The following sections describe the layers in more detail.

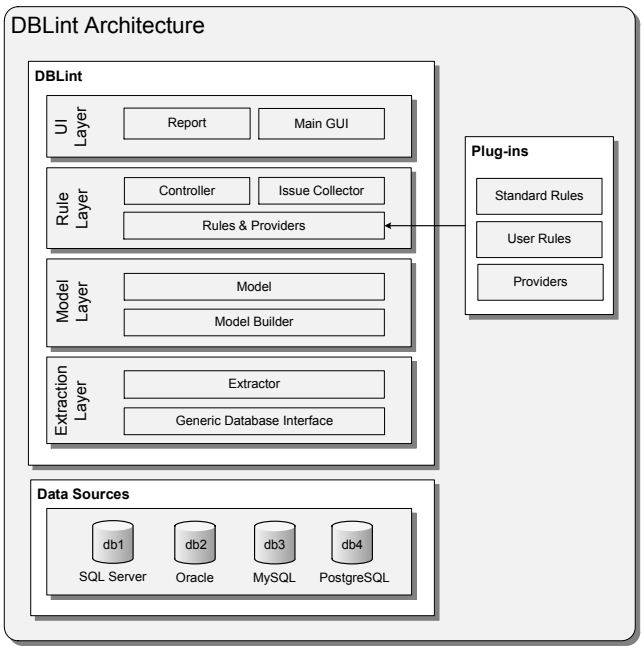


Figure 7: The layered architecture of *DBLint*. Rules are plug-ins loaded at run-time. Data sources are the external databases that *DBLint* examines.

5.1.1 Extraction Layer

The `Extraction Layer`'s main responsibility is connecting to a database, extracting metadata, and allowing data access. The `Extraction Layer` is a generic database interface that allows *DBLint* to process databases from different DBMSs in a uniform way. It handles all differences between different DBMS systems, such as loading DBMS specific drivers and accessing metadata.

5.1.2 Model Layer

The `Model Layer` consists of the two components: `Model` and `Model Builder`. The `Model` component is a collection of classes used to represent, for example, tables, columns, and indices. The `Model Builder` instantiates these classes using metadata from the `Extraction Layer`, thereby constructing an in-memory database model. This model is the main data structure used when expressing rules. The model is a fast and convenient way of accessing the database structure while being completely DBMS independent.

5.1.3 Rule Layer

Rules and providers are decoupled from *DBLint* and written as plug-ins, increasing the maintainability and flexibility

of the architecture. The definitions of rules and providers are the following.

Rule Definition A rule is an independent piece of code that reports design issues each time a violation of the rule occur. The rule examines the database design based on the in-memory database model. Furthermore, rules can access the data in the database. Rules are independent of each other, but can specify dependencies to providers.

Provider Definition A provider is similar to a rule, but instead of reporting design issues, a provider exposes additional information about the schema or the data. Providers can be used when multiple rules need the same information about a schema. For instance, if two rules need information about table importance, a provider can run the analysis and expose the information to both rules.

New rules and providers can be added easily due to the plug-in architecture, even by third-party developers. This is especially useful for companies that have their own constraints, which need to be enforced by specialized rules.

The main task of the `Controller` is to schedule rules and providers for execution. Rules are scheduled to be executed in parallel if possible. The `Controller` resolves dependencies between rules and providers, and ensures the right order of execution. The `Issue Collector` collects and manages issues identified by the rules.

5.1.4 UI Layer

The `UI Layer` contains the user interface of *DBLint*. This user interface is used to: configure the database connection, select schemas and tables to be analyzed, select rules to be executed, show status of analysis during an execution, and finally generation of the report.

5.2 DBMS Independence and Database Model

To make *DBLint* DBMS independent, the extraction layer specifies abstractions for creating database connection and extracting metadata and data. The extraction layer is designed as an instance of the abstract factory design pattern [9], using several abstractions from ADO.NET [18]. The functionality required to support each DBMS is specified in an interface. This interface makes it easy to support a new DBMS, as the only requirement is to implement this interface and extending the factory to use the new implementation. The two prerequisites for supporting a DBMS are that (1) it has an ADO.NET data provider implementation and (2) the DBMS provides a mechanism for extracting metadata.

A generic model of the database is used to make *DBLint* DBMS independent, such that rules can analyze the database design regardless of which DBMS the metadata comes from.

5.2.1 In-Memory Database Model

The in-memory database model is an object representation of the database design that provides a uniform representation of database objects such as tables, columns, and referential constraints. The model support most common database concepts, but due to time constraints it does not represent User Defined Types (UDT), check constraints or views.

The database model is a lightweight representation of a database; each table uses on average 20 kB of memory, including information about database objects such as columns, indices, and keys.

5.2.2 Metadata Extraction

The SQL standard defines the information schema, which is a number of system tables containing relevant metadata about the database's definitions [17]. Given the purpose of these standardized system tables, it should be easy to extract most of the data needed. However, most DBMSs have insufficient implementations of the standard, and extracting the full information requires querying DBMS specific system tables. In addition, the standard does not specify all the information needed to construct the model, e.g., index information.

Besides the differences in the structure of the system tables, there are also many differences in how metadata is represented. Boolean values, data-type definitions, referential constraints, update and delete rules, and so on, are often expressed differently. *DBLint* handles these differences in the extraction layer, where there is an implementation of the extraction interface for each DBMS. This makes the layers above the extraction layer DBMS independent.

5.3 Data Extraction

Analyzing the data in a database gives a better insight into the usage of the database, than only analyzing metadata. However, data analysis poses the following challenges.

1. Handling large data sets.
2. Keeping a simple data interface.
3. Limiting the number of data scans.
4. Maintaining DBMS independence.

Challenge 1 is to consider the performance aspect when implementing data rules. This is in contrast to metadata rules that have a very small overhead. Furthermore, if a rule stores a local copy of each row from a database, then *DBLint* is effectively limited by main memory, therefore this should be avoided.

Challenge 2 is to maintain the plug-in type of rule definitions. Every complication of this interface, e.g., due to resource cleanup or thread synchronization, is a deviation from the principle that *DBLint* should be easy to extend.

Challenge 3 is to reduce the number of data scans, such that each row is only extracted once from the database. This is necessary to reduce the load on the database server as well as the network traffic. Furthermore, extracting each row only once may prove to be an optimization, if network bandwidth is a bottleneck.

Challenge 4 is to maintain DBMS independence, such that rules use the same interface regardless of DBMS. This is difficult due to different handling of data types depending on DBMS. Therefore, *DBLint* must translate rows from different DBMSs into a uniform format.

When a metadata rule is executed, it analyzes the complete database before returning. A data rule cannot examine the complete database in the same manner due to the possibly large data sets and multiple scans. Instead data rules only examine one table at a time. A data rule is therefore executed multiple times, once for each table in the database.

This way, the `Controller` in the `Rule Layer` fully controls which tables are to be examined when.

We hypothesize that it is faster to limit examinations to one table at a time, than it is to analyze multiple tables at a time. The intuition behind this hypothesis, is that it allows the DBMS to fill its cache memory with data from that table alone [22], thus reducing reads from persistent storage.

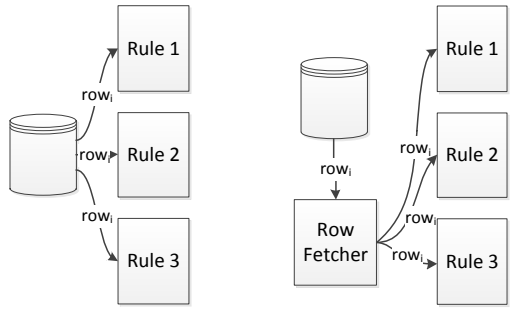
Rules access data, using the construction shown in Figure 8. Besides being relatively simple, this construction has the following advantages: it disposes the resources required to extract data, e.g. closing connections; it uses the well-known language constructs `using` and `foreach`; and finally it allows testing of different hypothesis about data extraction.

```
using (var rows = table.GetTableRows())
  foreach (var row in rows)
  {
    if (row["user_name"] != ... //Snip
  }
```

Figure 8: Code showing how to access rows from a table.

5.3.1 Data Extraction Strategies

The simplest implementation of the `GetTableRows` method in Figure 8, would be to return a data reader of a `SELECT * FROM` statement. This approach is illustrated by Figure 9a. It has the advantage of simplicity, but the disadvantage of all rules fetching all data from all tables.



(a) Simple extraction (b) Advanced extraction

Figure 9: Two strategies for data extraction.

In *DBLint*, rules are synchronized behind the scenes, such that only one data reader is created for each table, instead of having each rule creating a new data reader. The rows from this reader are distributed to all the rules enumerating the rows in the table. This is illustrated in Figure 9b. When a rule has processed enough rows to determine if there is an issue, it can cancel the enumeration by breaking out of the `foreach` loop. The `using` declarative then notifies the `Row Fetcher`, which then removes the rule from the set of rules enumerating the table.

5.4 Rule Scheduling

Each rule can specify dependencies to providers, and each provider can specify dependencies to other providers. A provider cannot expose the additional information about the schema to rules, before it has been executed. Therefore the order of execution is important and simply executing all rules and providers in parallel is inadequate. *DBLint* handles this problem by scheduling rules and providers such that all dependencies are fulfilled automatically.

Dependencies among rules and providers are represented as a dependency graph, in which rules and providers are represented as vertices, and a dependency from rule A to provider B is represented by an arrow from A to B. This is illustrated by the example in Figure 10. There are two rules in this example, the missing primary-key rule without dependencies and the inconsistent naming-convention rule with a dependency to the table-importance provider.

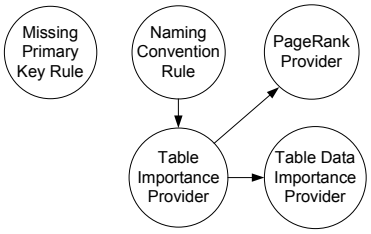


Figure 10: Rule and provider dependency graph.

To get a valid execution order, the dependency graph is sorted topologically, using a post-order depth-first traversal, starting at the nodes without incoming edges. A valid execution order is an order that satisfies the dependencies specified by rules and providers. Figure 11 shows one of the possible solutions when applying the algorithm to the example in Figure 10.

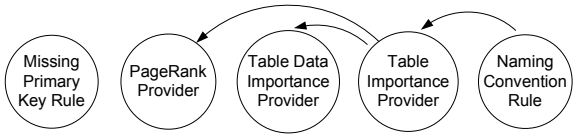


Figure 11: Topological-sorted dependency graph. The nodes in the graph are executed from left to right.

The execution order is parallelized such that rules and providers without interdependence are executed concurrently. For instance, in Figure 11 the first three nodes are executed in parallel, followed by node four and five.

5.5 Rule Configuration

Severities and thresholds in the default configuration of the rules are determined through an empirical evaluation of many schemas. However, all settings in *DBLint* are configurable as a mean to separate mechanisms from policies. If these thresholds and severities were hardcoded into the *DBLint* source code, it would be an argument against the usefulness of *DBLint*. Appendix F.1 demonstrates how severities and thresholds are implemented as configurable fields in *DBLint*.

5.6 DBLint Report

After analyzing a database, *DBLint* reports the results to the user. This report enables the user to quickly and intuitively understand the issue at hand, locate where in

the database the issue has been found, and provide enough information to understand why the issue has been raised.

Instead of providing a summary view in the user interface, the issues identified by *DBLint* are presented in an interactive HTML report that enables the user to explore issues, scores, and tables. The main reason for generating an HTML report is that it is a portable format, that can be saved and distributed. Furthermore, one of the developer teams evaluating *DBLint* requested that it should be possible to copy and distribute the report among team members.

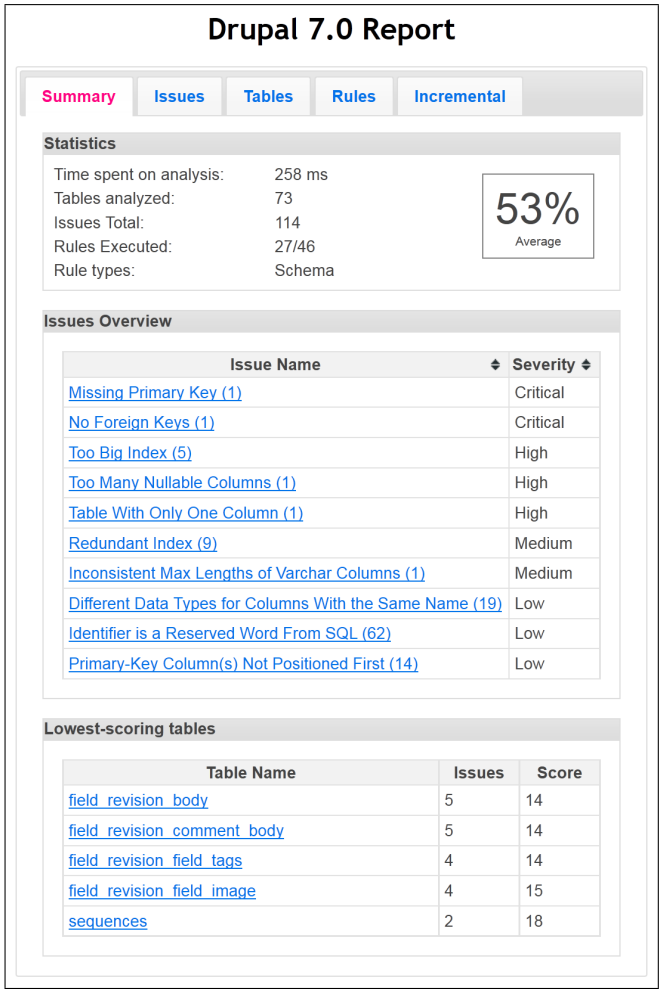


Figure 12: Summary page for the Drupal 7.0 CMS. Note that only schema rules are enabled because the database does not contain data.

5.6.1 *Report Views*

The report consist of five views: *Summary*, *Issues*, *Tables*, *Rules*, and *Incremental*. The *Summary* view is shown by the example in Figure 12. The summary shows the overall score, the types of issues found, and the most problematic tables. The summary can be used to gain an overview of the report, before exploring the issues further.

The issues found are displayed in the *Issues* view. Issues are grouped by type, e.g., missing primary key, and sorted by severity. Each group of issues can be expanded to get a detailed description of all issues. To further facilitate the understanding of an issue, it is possible to navigate to a detailed presentation of the related tables.

The *Tables* view contains a list of all tables, including their score, number of issues found, and importance. The list of tables can be sorted according to each of these properties, hence the *Tables* view has a number of uses. For instance, the most problematic tables in the database can be identified by sorting the list of tables according to the table score.

The *Rules* view shows the rules executed, and the number of issues found by each rule. This view allows the user to get a quick overview of the executed rules and identify the ones that reported most issues.

The *Incremental* view presents changes to the database between different *DBLint* runs. If a table has been modified between two runs, the table is listed in the *Incremental* view together with its previous score and the new score. This is useful for monitoring the quality of the database as the database evolves. The incremental view also lists new issues, such that the user can see exactly which issues caused the score to change.

5.7 Implementation

The implementation is written in C# 4.0 for the .NET platform, in 16,000 lines of code excluding blank lines and comments. One of the main reasons for using .NET is Language Integrated Query (LINQ), which provides a powerful querying mechanism, such that rules are able to easily query the metadata object model.

The report is generated using NVelocity, a template engine for .NET, outputting a collection of static HTML files. Javascript/jQuery is used to make the HTML files more interactive. The report has been successfully tested in major browsers such as Chrome, Firefox, Internet Explorer, and Opera.

6. TESTS

DBLint is tested in order to validate three hypotheses. The first hypothesis is that the rules in *DBLint* are applicable in general, i.e., *DBLint* finds issues in most schemas. The second hypothesis is that the issues found by *DBLint* are relevant. To the best of our knowledge, it is impossible to give a formal verification of the rules. Instead *DBLint* is applied to three widely used systems and the resulting reports thoroughly examined. This examination is intended to indicate whether issues are relevant. The third hypothesis, is that *DBLint* is fast enough to be used repeatedly throughout development, e.g., as part of a test suite.

Finally, the results of the evaluations of *DBLint* by four developer teams are described.

6.1 Database Design Comparison

DBLint has been tested on more than 35 schemas, ranging from small in-house schemas to large ERP systems, using all supported DBMSs. *DBLint* detected issues in all examined schemas. Out of the 35 schemas, we select 14 well-known and widely-used systems and compare their scores, shown in Figure 13. The horizontal axis is logarithmic with respect to the number of tables, and the vertical axis is the score. Note that data rules were excluded from this test, because real-world data sets were not available for all systems.

The result of applying each rule to these 14 schemas can be used to show the relevance of each rule. Table 7 in Appendix E shows how many issues each rule detected in

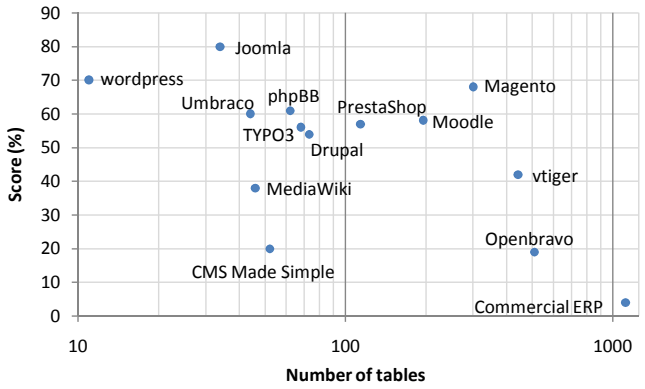


Figure 13: Overall scores of 14 well-known schemas.

the schemas, and the number of schemas that violated each rule. Rules "Different Data Type Between Source and Target Columns in a Foreign Key", "Redundant Foreign Keys", and "Varchar Columns With Length Zero" did not detect any issues. However, the first two of these rules did find issues when *DBLint* was evaluated by developer teams. Several general issues were identified, e.g., on average 15% of all tables are missing a primary key and on average 65% of all tables are not participating in any relationship with other tables. This is surprising as primary/foreign keys are fundamental concepts of good relational database design [29]. A similar list for data rules is shown in Table 8 in Appendix E. The databases used for this test is the same as the databases used for performance testing in Section 6.4.2.

6.2 Report Examination

To validate that *DBLint* finds relevant issues, reports from analyzing the three systems PrestaShop, phpBB, and Moodle have been examined manually. The three schemas have been chosen because they are non-trivial and widely used. PrestaShop has a reasonable amount of foreign keys, while phpBB does not have any. PrestaShop and phpBB have only been analyzed by metadata rules, whereas Moodle has been analyzed by data rules.

The issues in the reports are examined and categorized into one of the following three categories.

True positive There is evidence supporting that the issue is a real problem.

False positive There is evidence challenging the issue.

Undecidable Insufficient domain knowledge makes the issue undecidable.

6.2.1 *PrestaShop 1.3.3*

The PrestaShop issues are summarized in Table 3. The most important information is the bottom line which states that out of a total of 139 issues detected, 122 are validated to be actual design issues in the schema, 11 are undecidable and 6 are false positives.

It should be noted that the 21 critical issues are tables without primary keys, 66 of the medium issues are due to redundant indices, 22 of the low-severity issues are due to reserved words and 3 low-severity issues are due to primary-key columns not positioned as the first columns in the table. The rules detecting these issues cannot raise false positives, following the intuition that either you have a primary key or you do not. As such they will not be discussed further.

The issues raised by "Too many nullable columns" and "Table with too few columns" could not be determined to be true positives or false positives without better domain and application knowledge. Hence they are undecidable.

Rule "Inconsistent length of varchar columns" reported 20 columns with length 255 and 3 columns with length 256. Inspecting the columns (all of them) did not yield any evidence supporting that this difference is justified.

Rule "Table island" raised six issues. We believe that three of the issues are true positive. The names of the tables and columns strongly indicate that there should have been a foreign key, e.g., the column `order_message_lang.id_lang` should have referenced `lang.id_lang`. One issue is undecidable and the last two issues are false positives; both tables are independent tables unrelated to other concepts in the database, such as the `alias` table used to translate misspelled user search terms.

Rule "Different data types for columns with same name" reported 12 issues of which six are true positives, such as the column `date_upd` that appears 12 times as `DateTime` and one time as `Date`. Four issues are considered false positives. These instances involve columns with names such as "title" or "description". Such names can refer to different concepts depending on context, hence they are false positives. The last two issues are undecidable. They are reported on the two tables `range_price` and `range_weight`, which have very similar definitions. The two tables have the same columns, indices and foreign keys, but the two columns, `delimiter1` and `delimiter2` differ in their data types.

	True positive	Undecidable	False positive	Total
Critical	21	0	0	21
High	0	8	0	8
Medium	67	0	0	67
Low	34	3	6	43
Total	122	11	6	139

Table 3: The results of examining the issues for PrestaShop.

As can be seen in Table 3, false positives are only found in low severity issues.

6.2.2 *phpBB 3.0.7*

The issues for phpBB are summarized in Table 4. Note that the schema for phpBB is smaller than PrestaShop. 41 issues are reported totally, of which 31 have been found to be true positives, one false positive and nine undecidable.

In the phpBB schema there are no foreign keys, 11 tables without primary keys, five tables with redundant indices and one primary key where the columns are not in the same order as the primary key index. Because of the nature of these issues they will not be discussed further.

Rule "Table with too few columns" reported one issue: a table with a single column that is primary-key and appears to be a foreign key to the `users` table (inferred from naming convention). This corresponds to a boolean, i.e., something that can be modeled using an extra attribute in the `users` table. Considering that the `users` table has 76 columns

already, we think that the extra column in that table is justified. The extra byte could come from the 40 character varchar column used to store IP addresses.

Rule "Inappropriate length of default value for char columns" reports five issues. These issues are categorized as undecidable because all five occurrences are char columns of size 32 with the empty string as default value. This means that these columns may occupy more space than needed. However, all these columns are part of primary-keys, which is seen as a sign of the default value being rarely used.

Rule "Inconsistent max lengths of varchar columns" reported five columns of length 30 and four columns of length 32, but inspection shows that the columns refer to different concepts, and as such could be justified. On the other hand, we have not found any evidence indicating that the lengths are not arbitrary constants, hence the issues are undecidable.

Rule "Too large varchar column" reported 15 issues of which 12 are true positives and 3 are undecidable. We have tried to estimate the usage of the columns and see whether it is better, from a performance perspective, to use a CLOB instead. Instances where the entity is likely to be used without the large varchar field has then been rated true positives. It should be noted that of the issues reported 13 columns have a maximum of 4000 characters, and 2 columns have a maximum of 8000 characters.

Rule "Different data types for columns with the same name" reports one issue. The issue is a column named `code` and is used in two tables with size 8 and 50. The issue has been rated false positive because `code` can refer to different concepts depending on context.

	True positive	Undecidable	False positive	Total
Critical	12	0	0	12
High	1	5	0	6
Medium	5	1	0	6
Low	13	3	1	17
Total	31	9	1	41

Table 4: The results of examining the issues for phpBB.

The conclusion of examining the reports for PrestaShop and phpBB, is that the metadata rules detect relevant issues, with only few false positives. Furthermore, false positives occur only on issues with low severity.

6.2.3 Moodle 1.9.12

The issues for Moodle are summarized in Table 5. The Moodle schema has only been analyzed by data rules, and there are 273 issues in total. 151 issues have been found to be true positive, 96 undecidable and 26 false positive.

In the analyzed instance of Moodle 126 tables are not in use, which means that a medium severity issue is reported for each. These issues are all categorized as true positives and will not be discussed further.

Four issues are reported by the "Storing lists in character column" rule and each issue was categorized as true positive. The lists are storing references to rows in other tables. This means that the columns are used for one-to-many relations. These relations should be stored in additional tables, such that it is possible to use foreign keys to ensure data integrity.

The rule "The empty string used to represent null" reported 34 issues. Five of these are true positives because the columns are nullable and already contain null values. 29 of the issues are undecidable.

Rule "Number or dates stored in varchar column" reported three issues, of which one is categorized as true positive. The column `tolerance` with the default value '0.0' contains only numbers, indicating that the data type should have been a number. The last two issues are undecidable. However, the columns only contain numbers.

Rule "All values equals the default value" accounts for 17 issues of which 14 are undecidable. Three issues are false positives and the columns contain system settings that are all set to a standard value.

The rule "Redundant column" reports one issue, which is undecidable. The issue is found on a column containing 947 values that is redundant to a primary key column with values generated from a sequence.

Rule "Duplicated rows" finds two issues. Both of these issues are found by excluding the primary key column from the analysis and both primary keys are sequential integers. One issue is a true positive where a logging mechanism enters the same row twice into a log table. The second issue is a false positive, because it occurred on a table used for extending the system. In the current system the table has only one column besides the primary key column, and this column contains a default setting value.

Rule "All values are different from the default value" accounts for 27 issues. Four issues are true positive because they belong to mandatory columns such as `username` and `password` in the user table. The last 23 issues are categorized as undecidable.

Rule "Large unfilled varchar columns" reported 18 issues of which three are true positives. These columns contain data much smaller than the maximum allowed. In two of the columns the data is a concatenation of the same URL and a variable hash value. The result is that all values have the same prefix (the URL) followed by a fixed sized hash value. In this case the prefix value should be stored in a new table with a referential constraint and the column with hash values could be made smaller. The last 15 issues are undecidable.

The rule "Column values from a small domain" reported five issues. Three issues are true positives, because the values stored in the columns are closely correlated, e.g., `write` and `read`. The last two issues are false positives, because there is no immediate correlation between values.

Rule "Inconsistent casing of first character in text column" reported four issues. One issue is a true positive, because it contains e-mail addresses and one of these begins with a capitalized letter. The last three issues are all categorized as false positive. An example of a false positive is an Entity Attributes and Value (EAV) table where the value column contains a mixture of data types and casing.

Rule "Missing not-null constraint" reported three issues, of which two are true positives. These issues are found on columns that appear to be mandatory. The last issue is undecidable, because it contains last-modified dates. It is not possible to determine if this also includes the creation date (such a column is not present in the table), which means that the column should be mandatory as well.

The rule "Column containing too many nulls" reports nine

issues, of which one is a true positive. This is the column `createdby`, which appears to be mandatory but contains few values. Seven issues are undecidable because we cannot determine if these columns are unnecessary or just not in use in the current system. The last issue is a false positive. It belongs to a `modifiedby` column in a table that has not been modified yet.

Rule "Column with only one value" reports four issues, which are all categorized as undecidable.

	True positive	Undecidable	False positive	Total
Critical	4	0	0	4
High	6	46	3	55
Medium	127	0	1	128
Low	14	50	22	86
Total	151	96	26	273

Table 5: The results of examining the data issues for Moodle.

The result from examining data stored in the Moodle schema reveals that data rules are more uncertain and that it is difficult to determine if the issues are true positives. Data rules reveal interesting information about the data and the schema design. Much of this information could be used to present statistical information instead of including it as issues. However, this would require adding new features to *DBLint*.

Analyzing the Moodle schema showed that data analysis faces some challenges when the schema is designed for general purposes, which is the case in a Content Management Systems (CMS). Many of the false positives and undecidable issues are due to a range of unused functionality in the CMS. This is also indicated by the large number of empty tables.

Outlier Data in Moodle. The data analysis performed on Moodle does not include the rule that detects outlier data. The rule has been left out because tests have shown that this kind of rule conflicts with the principles of *DBLint*. The current implementation of outlier detection, with low-configuration finds data of interest in Moodle. However, it is difficult to characterize these occurrences as design issues. An example of outlier data found in Moodle is a column containing 748 rows with decimal values, of which one value is 100 and all other values are between zero and 25. Another example is a column with first names containing 257 rows, of which the majority is a single word, and one instance where the name consists of three words.

6.3 Naming Convention

To verify the approach used to find deviations in the naming convention two schemas are manually examined with respect to their naming convention. The results of this are compared with *DBLint*'s results. The two systems are Typo3, in which *DBLint* found 51 naming convention issues, and Drupal in which there are no issues.

When manually examining the two schemas, we first analyzed all identifiers to determine the naming convention. Afterwards, all the identifiers were examined again to find those that deviated from the convention.

Comparing the result from the manual examination and the results from *DBLint* using the Markov-chain representation, we can see that they agree on all inconsistencies. From this we conclude that the approach taken in discovering naming convention satisfies our intentions.

6.4 Performance

The purpose of the performance tests is to show that *DBLint* is a fast tool that is applicable in a development process. Furthermore, we want to show that the method used for extracting data and executing rules is efficient with respect to both time and network traffic.

Test Setup. The setup used for performance testing consists of two machines, a client machine running *DBLint* and a database server. The database server is a virtualized Windows 2008 Server with a 2.3 GHz quad core processor and 6 GB RAM. The client machine has a 2.26 GHz dual core processor with 4 GB RAM, running Windows 7. The client and server are connected by a 100 Mbps LAN network, and *DBLint* is configured to use 20 concurrent connections to the database server.

6.4.1 Metadata and Data

There is a significant difference in the time it takes to run metadata rules compared to data rules. This is expected as the amount of data is possibly much larger than the metadata. On average, the metadata analysis takes 1.3 seconds for every 100 tables. The results are based on the 14 schemas in Figure 13, and shows that *DBLint* is suitable for use in a development process regarding speed. Data analysis takes on average 2.5 minutes for every 100,000 rows, corresponding to 7 GB/hour.

6.4.2 Data Access and Execution

The purpose of this test is to show that the strategy used for extracting data and executing rules in *DBLint* is efficient. Brief descriptions of each strategy for accessing data and executing rules are given in the following.

Sequential Rules are executed sequentially one by one. This method is simple to implement, but is expected to be slow, and expensive in network traffic because all rules extract data independently.

Concurrent Rules are executed concurrently and each data rule runs on multiple tables at the same time. This method is expected to be faster than the sequential method because of better CPU utilization.

Table synchronized Rules are executed concurrently on one table at a time. This is expected to be faster than the previous method because the database server should be able to hold the entire table in its cache, thus minimizing disk I/O on the database server.

Row synchronized Data reads are synchronized such that each row is fetched once and distributed among rules, minimizing network traffic.

Data Set. The data set consists of nine databases containing a total of 9,090,618 rows. Four of these databases contain test data, totaling 1.25 million rows. A complete list of the databases in the tests is shown in Table 6.

Database	Test data	Rows
Moodle		93,420
Commercial ERP		6,164,421
vtiger	x	7,942
Drupal		1,446,707
Xcart	x	64,490
Small e-commerce		128,284
Wordpress		12,530
SASSDM	x	1,102,472
Openbravo	x	70,352
		9,090,618

Table 6: Data sets used for performance evaluations.

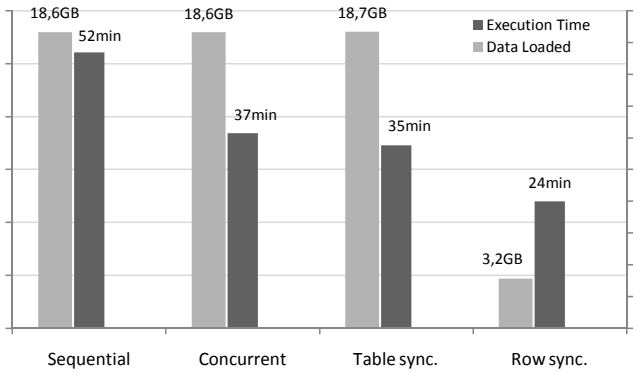


Figure 14: Comparison of the four strategies used for extracting data and executing rules.

Results. The results of the test are shown in Figure 14. The amount of data transferred from the database is shown for each method, as well as the total execution time. The first three methods transfer the same amount of data, which is expected because there is no synchronization of data between the rules. Executing rules concurrently yield a faster execution, as shown by the sequential and concurrent execution-time bars in Figure 14. Synchronizing data rules such that they run on one table at a time did not yield any significant improvements. The intuition for this to be faster, is that the database server should be able to hold the entire table in its cache and therefore serves data faster. There are three explanations why it did not improve the execution time. (1) The database server is not the bottleneck, i.e., the network and the client machine executing the rules is. (2) There will be a wait at the end of each table as the fastest rules are waiting for the slowest rule to complete. (3) Buffer eviction strategies are optimized for regular OLTP work-loads rather than full table scans [23, 26].

The row-synchronized method currently implemented in *DBLint*, is the most efficient method with respect to both network traffic and execution time. Significantly less data is extracted from the database, because rules are able to share the extracted data. The total execution time is also improved, possibly due to the reduced network traffic, reducing the network bottleneck.

To conclude, the method used for executing rules and extracting data in *DBLint* is efficient in terms of both network traffic and execution time, compared to executing the rules sequentially or concurrently without synchronization.

6.5 User Feedback

DBLint has been successfully used to examine eight schemas, developed and used by four organizations. The output from *DBLint* was reviewed by senior developers and they expressed their opinions about the design rules, the discovered design issues, and the tool itself.

The idea of having a tool assisting the development process and giving feedback about the quality of a database design is good. The developers found it less intimidating to receive feedback from a tool than from a superior/colleague.

The schema rules implemented in *DBLint* identified relevant design issues that are considered design errors by the developers. Examples of such issues are inconsistent lengths on varchar columns, redundant indices, and different data types between source and target columns in foreign keys. The data rules also identified issues, however, many of these issues could not be characterized as design issues, but more as additional information about the design. For instance the use of different data types in a varchar column could often be justified; however, the ability to give the information that a column contained a mixture of data types was relevant.

Discussing the main principles of *DBLint*, our intentions of making *DBLint* a low-configuration tool, and the ability of using it in a matter of seconds was appreciated. The extensibility of *DBLint* was also mentioned as an important feature, making the tool much more adaptable.

Data analysis was included into *DBLint* to make a wider analysis and possibly reveal new issues. However, during the evaluation of *DBLint*, it was suggested that data analysis could also be used to monitor how the data evolves. For instance, a change in the percentage of null values may require an adaptation of the overlying applications.

Overall, *DBLint* received positive comments and has proven to be relevant for developers in assisting the development of better database designs.

7. DISCUSSION

Throughout the tests we made a number of observations related to the scoring system and outlier detection. These observations are due to a number of trade-offs in the design of *DBLint*, and mark the boundaries of what is possible within an automated, low-configuration tool, such as *DBLint*.

7.1 Scoring

Achieving a fair score that correctly reflects the quality of a database design, based on issues reported by the rules, is challenging. A score will necessarily be a trade-off between many criteria, such as simplicity and completeness. The score in *DBLint* is designed to solve many problems, thereby trading simplicity for completeness. Some of the key aspects of the scoring system in *DBLint* are the following.

- The score is independent of the size of the schema, because the scoring function calculates an average instead of a sum.
- Issues in *DBLint* have different context, i.e., some issues concern specific columns while other issues concern the schema as a whole. The scoring system takes this into account by only reducing the score of the relevant context.

- Issues are required to specify a severity such that issues which are less severe do not reduce the score as much as critical issues.
- Table importance is included when calculating the score of a schema, such that important tables contribute more to the score than peripheral tables.
- All scores are given as a number between 0% and 100%, which is easy to read by the user. This is implemented by making the scores follow an exponential-decay curve, starting at 100% for databases without issues.

If a more simplistic scoring system is chosen, some of the above problems cannot be addressed. An example of a simpler scoring system is one that has a single function incrementing a number for each issue in the database. A lower number would then indicate fewer issues and a better design. The problem with this approach is that the scope of each issue will be ignored, e.g., a schema-level issue and a column-level issue will contribute equally to the score. Another problem is that this score is not independent of schema size.

The scoring system in *DBLint* does not take the number of rules into account. This means that the score will drop as more rules are added. One could argue that the score should stay the same because the schema is unchanged. On the other hand, as more rules are added, new problems are unveiled and the number of issues to correct becomes larger. Furthermore, it is difficult to take rules into account because they are very different. For instance, some rules are general and generate many issues, while others are triggered less often.

7.2 Experiences with Outlier Detection

DBLint, being a low-configuration tool with domain-independent rules, conflicts with outlier detection. The problem is that when converting data into points, all data is processed equally independently of what is actually stored. This means that data such as e-mails, titles, and descriptions, are all treated equally. Consider the example where varchars are converted to points in three dimensions: the number of words, the average length of words, and the total length of the varchar. Applying this to a column containing descriptions makes sense, but not on a column containing e-mails. On such a column the first dimension will have the same value for all rows, while the other two dimensions will be equal for all rows, hence they will likely all be in the same cluster. This example shows that there is a need for different conversions depending on the type of data stored in the columns.

Using the same approach for all columns of the same type causes too many false positives and may omit other outliers. It would be possible to make these more complex configurations available in *DBLint*. It would, however, conflict with the low-configuration principle of the tool.

8. CONCLUSION

DBLint is a fast, configurable, and extensible tool for analyzing database designs. *DBLint* addresses the problem of time-consuming manual reviews, being an automated tool containing 46 design rules. The rules are configurable, DBMS independent as well as domain independent, and examine both metadata and data. *DBLint* outputs an interactive report that enables the user to browse all detected issues.

DBLint calculates an overall score as well as on individual tables, based on the found design issues. A score has a number of benefits, e.g., developers can be pointed towards the most problematic parts of the database, or get an overall idea of how well the given schema is designed.

DBLint has been extensively tested on more than 14 real-world schemas, identifying a large number of relevant design issues. Furthermore, *DBLint* has been tested in four developer teams with positive results. Several issues were verified to be relevant, of which some have been corrected afterwards. The evaluations substantiated our intuition of a need for a database design verification tool, to assist developers in keeping a consistent and high quality database design.

9. FUTURE WORK

Currently, the in-memory database model in *DBLint* does not include check constraints, views, and UDTs. Including these will extend the possibilities of metadata analysis and give a more complete view of the database design.

Currently, the focus in *DBLint* is on diagnostics and not auto-correcting issues. This is because *DBLint* is unaware of the impact auto-correction will have to the overlying applications, which makes it dangerous. However, a possible extension is to produce SQL code that correct some of the issues, and display it together with the issue.

The current architecture of *DBLint* allows for advanced data analyses such as functional-dependency detection and inclusion-dependency detection. By implementing these analyses, rules will be able to verify that a database is properly normalized. Furthermore, these analyses will allow *DBLint* to detect missing foreign keys, which was requested during the user evaluation of *DBLint*.

The data access methods in *DBLint* can be extended to include data sampling, allowing for analysis of much larger data sets.

10. ACKNOWLEDGMENTS

We would like to thank the database design teams at Aveva Denmark, House of BI and Atira for evaluating the *DBLint* tool and giving feedback. We would also like to thank Michael M. Hansen from Aalborg University for evaluating and discussing the tool.

11. REFERENCES

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 1–8. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-595-3.

[2] R. Bouman. Finding redundant indexes using the mysql information schema. `http://www.oreillynet.com/databases/blog/2006/09/_finding_redundant_indexes_usi.html`.

[3] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. *SIGMOD Rec.*, 29:93–104, May 2000. ISSN 0163-5808.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Seventh International World-Wide Web Conference (WWW 1998)*, 1998.

[5] C. Calero, M. Piattini, and M. Genero. A case study with relational database metrics. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*. IEEE Computer Society, Washington, DC, USA, 2001. ISBN 0-7695-1165-1.

[6] C. Coronel, S. Morris, and P. Rob. *Database systems: design, implementation, and management*. Course Technology Cengage Learning, 2009. ISBN 9780538469685.

[7] J. Currier. SchemaSpy. `http://schemaspy.sourceforge.net`.

[8] S. Fatehi. SchemaCrawler. `http://schemacrawler.sourceforge.net`.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0201633612.

[10] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, June 1984. ISSN 0163-5808.

[11] D. M. Hawkins. *Identification of outliers*. Chapman and Hall, 1980. ISBN 041221900.

[12] S. C. Johnson. Lint, a C Program Checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[13] B. Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010. ISBN 9781934356555.

[14] E. M. Knorr and R. T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 392–403. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-566-5.

[15] Lint4j. `http://www.jutils.com`.

[16] S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619670.

[17] J. Melton and A. Simon. *SQL:1999: understanding relational language components*. Morgan Kaufmann series in data management systems. Morgan Kaufmann, 2002. ISBN 9781558604568. LCCN 2001090723.

[18] Microsoft. Ado.net. `http://msdn.microsoft.com/en-us/library/aa286484.aspx`.

[19] Microsoft. Maximum size of index keys. `http://msdn.microsoft.com/en-us/library/ms191241.aspx`.

[20] Microsoft. Reserved keywords. `http://msdn.microsoft.com/en-us/library/aa238507.aspx`.

[21] D. L. Moody. Metrics for evaluating the quality of entity relationship models. In *Proceedings of the 17th International Conference on Conceptual Modeling*, ER '98, pages 211–225. Springer-Verlag, London, UK, 1998. ISBN 3-540-65189-6.

[22] MySQL. The innodb buffer pool. `http://dev.mysql.com/doc/refman/5.5/en/innodb-buffer-pool.html`.

[23] Oracle. Memory architecture. `http://download.oracle.com/docs/cd/B28359_01/server.111/b28318/memory.htm`.

[24] Oracle. Schema object names and qualifiers. `http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/sql_elements008.htm`.

[25] M. Piattini, C. Calero, and M. Genero. Table oriented metrics for relational databases. *Software Quality Control*, 9:79–97, June 2001. ISSN 0963-9314.

[26] PostgreSQL. Notes about shared buffer access rules. Internal documentation for V9.0.4 located in: /src/backend/storage/buffer/README.

[27] Pylint. `http://www.logilab.org/857`.

[28] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[29] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006. ISBN 0072958863, 9780072958867.

[30] D. Software. Database examiner. `http://www.dbesoftware.com`.

[31] SSW. Sql auditor. `http://www.ssw.com.au/ssw/SQLAuditor`.

[32] E. Teniente, C. Farré, T. Urpí, C. Beltrán, and D. Gañán. SVT: schema validation tool for microsoft SQL-server. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 1349–1352. VLDB Endowment, 2004. ISBN 0-12-088469-0.

[33] X. Yang, C. Procopiuc, and D. Srivastava. Summarizing relational databases. *Proceedings of the VLDB Endowment*, 2(1):634–645, 2009.

[34] C. Yu and H. V. Jagadish. Schema summarization. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 319–330. VLDB Endowment, 2006.

All websites have been accessed may 23, 2011

APPENDIX

A. SCHEMA-RULES ARGUMENTATION

Missing Primary Key

A primary key uniquely identifies rows in tables. Missing a primary/unique key on a table allows duplication of rows, which should be avoided. Furthermore, individual rows cannot be referenced using foreign keys when the table lacks a primary/unique key. If a table does not contain columns suitable for a primary key, it is always possible to create a surrogate key.

Different Data Type Between Source and Target Columns in a Foreign Key

A foreign key is a relationship between two tables, a source table and a target table. Values from the source column is stored in the target column, hence the data type of the two columns should be the same. However, it is possible to create a working foreign-key relationship between two columns of different data types. For instance, a source column `number(8)` and a target column `number(4)`. This may lead to an application crash when inserting data because the domain of the target column is smaller than the source column.

Varchar Columns of Length Zero

A column designed to contain no data is simply a bad design practice. A varchar of length 0 could be used to represent boolean values, such that the empty string equals true and a null value equal false. However, there are better and less obscure ways to model boolean values.

Inconsistent Naming Convention

Using consistent naming of attributes and entities makes life much easier for the database designer and application programmers [29]. An inconsistent naming convention complicates writing queries and understanding the schema.

Inappropriate Length of Default Value For Char Columns

A char column always occupies the specified length, even when the empty string is used. Therefore, char columns should only be used if the length is small or the size of the data is known in advance. Otherwise, varchar columns should be used because they occupy only the space corresponding to the actual data.

Redundant Foreign Keys

Duplicate foreign keys could have contradicting referential actions, such as `CASCADE` and `SET NULL`. Having contradicting referential actions may lead to unforeseen events when, e.g., deleting rows. Furthermore, if the foreign-keys have indices the DBMS will have to maintain more indices. A duplicate foreign key can be deleted with little effort.

Table With Too Few Columns

Tables with zero or one column are suspicious. A table with zero columns cannot contain any data. A table with one column can be accepted under special circumstances, but should generally be avoided.

Too Big Indices

Large indices reduce performance because they are expensive to maintain, and should be avoided when smaller keys are sufficient. Some DBMSs have a maximum key size on indices, e.g., SQL Server is limited to 900 bytes per key [19]. In some cases, a large natural primary key can be replaced with a surrogate key.

Too Many Nullable Columns

In *DBLint* there are two cases where a table is said to contain too many nullable columns:

1. All columns are nullable except the primary key columns.
2. A large percentage of the columns are nullable.

The first case is especially bad if the primary key is a single surrogate key, because a row can contain no useful data. In the second case, it is likely that the developer forgot to add the appropriate not-null constraints.

Too Long Column Names

The maintainability of a schema might decrease with long names, because it makes identifiers harder to remember and queries more difficult to write. Furthermore, Oracle does not allow column names to exceed 30 characters [24].

Nullable and Unique Columns

Null in a database typically refers to "value does not exist" or "value unknown", and as such should not be allowed in columns, which have a unique constraint defined. Null values in unique indices are handled differently depending on DBMS: Some DBMSs allows zero or one null value in a unique index, while, e.g., Oracle allows multiple null values. This difference may be a portability issue, and cause misunderstandings among developers of the different DBMSs.

Cycles Between Tables

A cycle can be necessary to model specific data structures, e.g., a hierarchical structure. However, the developer should be aware that the cycle exists, because circular dependencies may cause several problems if deferrability and delete rules are not considered. These problems are the following.

- If there is a cascade delete on all references, it is possible to delete all data in the tables.
- If no references are deferred and the columns are mandatory, data cannot be inserted.

Inconsistent Max Lengths of Varchar Columns

Inconsistent maximum length of varchar columns is a rule purely about consistency. Consider an example with 200 columns of maximum length 256 and three columns of length 255. These three columns are deviating from the majority, and could be 256 without conflicting with the data in the columns.

Self-Referencing Primary Key

Having a foreign key relation on a primary key column referencing itself strongly indicates an error. The foreign key must reference its own row and does not contain any useful information. Such a foreign key can be deleted without any loss of functionality or conflicts in the database.

Inconsistent Data Types in Column Sequence

A sequence of related columns can be inferred from the naming, e.g. `address_1`, `address_2`, ..., `address_n`. Another example is columns used for extensibility, e.g., 10 columns (`cust_col_1`, ..., `cust_col_10`), used to store unforeseen information after the database is deployed.

All columns in the sequence should have the same data type to avoid confusion and potential errors. Imagine that there are 10 columns in a sequence and the third column's data type is integer and the others are varchars. This may result in problems because a developer might mistake the third column for being a varchar, like the others. Furthermore, varying data types in a column sequence violates consistency.

Missing Column in a Sequence of Columns

If there exist a sequence of columns, e.g., `col_1`, `col_2`, ..., `col_n`, the postfix number should be ordered sequential from 1 to n. If a column is missing from a sequence it has probably been forgotten or deleted without proper refactoring.

Primary- and Unique-Key Constraints on the Same Columns

Having a primary- and unique-key constraint on the same columns makes the unique constraint redundant. The unique key can be deleted without affecting data integrity.

Redundant Indices

Redundant indices are usually not necessary. A redundant index is an index where the sequence of columns is a prefix of another index, e.g., the index `inx_a(col_1)` is redundant to `inx_b(col_1, col_2)`. Having redundant indices is a performance issue because the DBMS needs to maintain more data structures than necessary. There are exceptions where a redundant index is reasonable, but most likely it can be deleted without any problems. [2]

Too Short Column Names

Columns should be named with meaningful and distinct names [29]. This makes it easy to read and understand the data model and queries. Very short column names have a tendency to consist of abbreviations or letters that have certain meaning in the development team. However, these columns are not very maintainable and make queries less understandable.

Too Many Text Columns in a Table

LOB columns containing text are used to store large string values. Normally they will only take up the space they need, however the data are stored outside the table, and hence it requires an additional I/O for each value. If a table contains a large number of these columns it could indicate that the developer were unaware of the different data types.

Foreign-Key Without Index

When deleting/updating a row from the referenced table, the DBMS checks that the specific row is not referenced, and takes corresponding action depending on the delete/update rule. This check must look-up values in the referencing table, which requires a full table scan if an index does not exist. Having an index on the foreign-key columns will make this look-up faster.

Primary-Key Columns Not Positioned First

It is convention to position the primary-key columns first [29]. The order of columns in a table is important for readability purposes. A related case is when a table contains a sequence of columns, such as (`address_1`, `address_2`, ...), and it is natural to place the columns ascending based on the postfix number. Similarly placing the primary-key columns first makes it possible to quickly see how rows are uniquely identified.

Use of Reserved Words From SQL

Reserved SQL keywords such as `date` and `from` should be avoided when choosing identifiers [20]. Avoiding reserved SQL keywords in identifiers makes the queries more readable and names will not need to be escaped in queries.

Different Data Types for Columns With the Same Name

A column's name often refers to a concept, hence when the same name is used with different data types the representation of that concept is inconsistent. Possible errors that could arise include implicit casts.

SQL clauses such as natural join and using, matches columns based on names. Without care two columns could easily be matched, which will make implicit casts.

Generic names such as `value` and `content`, do not necessarily refer to the same concepts.

Use of Special Characters in Identifiers

Special characters in identifier names should be avoided, except the character '_' for the following two reasons.

- Identifiers must be escaped in queries.
- Identifiers cannot be mapped directly into programming languages.

In practice there are almost no good reasons for using special characters instead of an understandable/describing name. For instance, a product table containing a column with products numbers, could be named '`#`' but a better solution would simply be `product_no`.

Table Islands

Having a connected schema graph means that the data is related. If the schema is not connected it is possible that one is trying to model two separate concepts or businesses. In that case it is better to extract the table islands into separate schemas.

Too Large Varchar Columns

Large varchar columns are a problem because they may cause the row to overflow resulting in chaining. Chained rows are slower to extract from the database as they require additional I/Os.

B. DATA-RULES ARGUMENTATION

Duplicate Rows in a Table

Duplicate rows in a table are not desirable [29], because they require additional space and may lead to an inconsistent state. In *DBLint* the check for duplicate rows is divided into two categories:

Pure duplication A row is duplicated if the entire row is represented in the table more than once.

Semi duplication A row is a semi-duplicate if two rows contains the same data, when ignoring the auto-increment primary-key column. Semi-duplication shows that the same data is present but are identified in different ways.

Duplicate rows indicate problems in the way data is validated when modified. Pure-duplicate rows can be deleted from the database, saving storage space.

Storing Lists in Varchar Columns

Storing lists in varchar columns is recognized as an anti-pattern in [13], and it is a violation of the first normal form [29]. It indicates that the overlying application has logic that handles such a list. However, in a database context such a list should be modeled using a second table, with a one-to-many relation. Furthermore, if the list is used to reference rows in another table, it is not possible for the DBMS to enforce referential constraints on the relation. This means that the list can reference a row that no longer exists, leading to problems in the overlying application.

Wrong Representation of Boolean Values

A boolean value is either true or false, i.e., it is possible to represent the value with only one bit. Not all DBMSs have a data type for boolean values, e.g., Oracle, resulting in many alternative ways of storing booleans. We have observed booleans stored in char and varchar columns using any of the following values: (true, false), (yes, no), (t, f), (y, n), (j, n), (1, 0), (2, 1).

A good boolean representation is both unambiguous and space efficient. Words are space inefficient, hence ruled out. The convention from the C programming language, i.e., (1, 0) for true and false respectively is a possibility. This representation is, however, problematic as it relies on the programmer being an experienced C programmer. We see the (2, 1) representation as testament to this problem. Instead, single char columns with values such as (t, f) or (y, n) could be used. This requires only one byte and is unambiguous.

Another aspect of representing booleans is that is should be consistent across the schema, i.e., it should not be a mixture of chars, words and numbers.

Defined Primary Key is not a Minimal Key

A primary key is a minimal superkey [29]. If the defined primary key is not a minimal superkey, it means that it is possible to identify a row with fewer attributes. Using a superkey instead of a primary key is even less attractive when other tables need to reference it. Each of the referencing tables will need to hold more information than actual needed, resulting in using more space and less efficient indices.

Redundant Columns

A table with two or more columns containing the exact same values for all rows indicates that one or more columns are unnecessary. If one of the columns is in a unique key or primary key, it indicates a third normal-form violation [29].

All Values Equals the Default Value

If all values in a column equal the default value, then the entropy of the column is equivalent to the column containing only empty strings. This is seen as indicative of the overlying application ignoring this field. If the column is unused it should be removed to prevent cluttering of the design and to save space.

Not-Null Columns Containing Many Empty Strings

If a varchar column has a not-null constraint, it is mandatory. If the column contains many empty strings, it indicates that the overlying application circumvents this restriction. This could be the result of misunderstandings between application and database developers. Modeling unknown or nonexistent values with the empty string should be avoided.

Numbers or Dates Stored in Varchar Columns

If a varchar column contains only numbers or dates, it indicates that an incorrect data type is chosen. Choosing a more strict data type ensures better data quality.

There are design patterns, such as the EAV, that uses the varchar data type to store many different data types. However, if the column contains numbers or dates exclusively, it indicates that the data type of the column could be changed.

Empty Tables

A table without data clutters the design unnecessarily. Note that this only applies to regular tables, and not to temporary tables.

Mixture of Data Types in Text Columns

Having a varchar column that contains a mixture of data types can be necessary in some design situations, such as when using EAV where multiple data types are stored in the same column. However, in general this is seen as the overlying application modeling different concepts using one column.

Columns With Only One Value

If a column contains only one value it indicates a possible redundancy. However, there are exceptions to this rule, such as columns with boolean values, or columns containing only values from a small domain. An example of this could be all users having the same time zone.

All Values Differ From the Default Value

If the default value differs from all values in a column, the default value is not used. The default value could be a legacy from an earlier design. Removing the default value from the column definition should not affect the overlying application. Values from a small domain such as booleans are an exception to this, because of cases where, e.g., a table `users` have an `activated` column. This column will have the default value 'false', but all users will be activated and hence have the value 'true'.

Inconsistent Casing of First Character in Text Columns

If the casing of the first character differs in a text column, is a sign of data quality issues. For instance, it could indicate that the overlying application does not validate user input, such as e-mails correctly.

Unnecessary One-to-One Relational Tables

Modeling a one-to-one relation with a relational table connecting two entities is often unnecessary. If the relational table covers most values in one of the source tables, the relation could be modeled using an additional column.

Column Values from a Small Domain

If a varchar column contains values from a small domain, the data could come from an enum structure. Some DBMSs supports the enum data type that should be used instead. If the enum type is unavailable on the used DBMS, the column should have a check constraint ensuring that the column only contains allowed values.

Large Unfilled Varchar Columns

The maximum length of a varchar should be selected such that it matches the data that are stored in the column. If the data in the column only uses less than half of the maximum length, the column width could be decreased.

Missing Not-Null Constraints

If a column is defined to be nullable without containing any null values, the column should be declared with the not null constraint. This is possibly due to the designer forgetting to add a not-null constraint.

Column Containing Too Many Nulls

A column with very few values could indicate functionalities rarely used or legacy columns.

Outlier Data In Column

Outlier data may indicate missing check constraints or dirty data. When a column contains data that deviates from the majority, it may be generated by another mechanism. To avoid that a process stores dirty data, the definition of the column could be made more strict by adding check constraints.

C. SCORING PENALTY FUNCTIONS

The penalty function for tables p_t is defined as follows.

$$penalty_t(iss) = \begin{cases} 1.40 & \text{if severity(iss)} = \text{critical} \\ 1.00 & \text{if severity(iss)} = \text{high} \\ 0.80 & \text{if severity(iss)} = \text{medium} \\ 0.60 & \text{if severity(iss)} = \text{low} \end{cases}$$

The penalty function for columns p_c is defined as follows.

$$penalty_c(iss) = \begin{cases} 1.60 & \text{if severity(iss)} = \text{critical} \\ 1.40 & \text{if severity(iss)} = \text{high} \\ 1.20 & \text{if severity(iss)} = \text{medium} \\ 1.00 & \text{if severity(iss)} = \text{low} \end{cases}$$

D. SUMMARY

Real world database schemas are often very complex, and therefore difficult to create and maintain without making errors. However, some errors can be detected automatically, thus lowering the burden on the developer. In this paper, we propose *DBLint*, an automated tool designed to assist developers when developing schemas, such that they avoid many common pitfalls. The targeted audiences include both new and experienced database developers.

DBLint comes with 46 database design rules. Of these, 27 analyze schema metadata and 19 analyze the data in the database. The set of analyses range from straight-forward checking of specific properties such as "does this table have a primary key?", to non-trivial analysis such as "detect the naming convention and find deviations" and "detect outliers in the given data set".

DBLint has a flexible, extensible and layered architecture. The architecture ensures DBMS independence, while providing access to metadata and data. A major aspect of the architecture is that it handles as much as possible for rules, such that they are kept simple. Examples of this include that each rule may specify a number of configurable options that *DBLint* identifies, saves, and restores across runs. Rules are plug-ins, loaded and executed at run-time.

When all rules have been executed, *DBLint* generates a report with the detected issues. This report is an interactive HTML document, and an intuitive and effective way of reading issues. The report is effective because it, in addition to describing issues, also describes the analyzed database. For instance, when examining a redundant index, the full metadata information of the related table is available.

The report contains an overall score that summarizes all detected issues in the schema. The score is between 0% and 100%. This score is aggregated over all issues detected, whether they are on the schema, table or column level. The score is calibrated such that a score of 50% corresponds to the average schema. The score is normalized with respect to the size of schema, hence two schemas of different size can be compared directly without corrective measures. In addition to the total score, a score for each table is given, such that the most problematic tables can be identified.

We have compared the issues found in 14 schemas from widely-used systems. From this comparison we have made the interesting observation that many schemas do not use foreign-keys, and many tables do not have a primary-key declared. This is surprising, as these concepts are fundamental database concepts. This observation clearly demonstrates the need for *DBLint*. In addition to this quantitative study, we perform a qualitative study. Three reports from analyzing the metadata and data of three open source systems have been examined thoroughly and we conclude that *DBLint* finds many relevant issues and that the noise-to-signal rate of *DBLint* is low with only few false positives.

DBLint has been successfully evaluated by four organizations, with positive feedback regarding: the rules, the low-configuration principle, the score and the report.

Finally we have evaluated the performance of *DBLint*, showing that roughly 100 tables/second can be analyzed when only considering metadata. When considering data approximately $4 \cdot 10^4$ rows/minute can be analyzed.

E. RULE SUMMARY

Table 7 shows the number of issues found by each rule when running *DBLint* on the 14 schemas from well-known systems. The column 'C' shows the total issue count and the column 'S' shows the number of schemas violating each rule. Table 8 contains a similar list of data rules when running *DBLint* on nine databases containing both real-world and test data.

#	Metadata Rule	C	S
1	Missing Primary Keys	256	11/14
2	Different Data Type Between Source and Target Columns in a Foreign Key	0	0/14
3	Varchar Columns of Length Zero	0	0/14
4	Inconsistent Naming Convention	64	5/14
5	Inappropriate Length of Default Value For Char Columns	21	3/14
6	Redundant Foreign Keys	0	0/14
7	Table With Too Few Columns	152	7/14
8	Too Big Indices	229	6/14
9	Too Many Nullable Columns	197	9/14
10	Too Long Column Names	13	2/14
11	Nullable and Unique Columns	13	4/14
12	Cycles Between Tables	11	1/14
13	Inconsistent Max Lengths of Varchar Columns	21	9/14
14	Self-Referencing Primary Key	2	1/14
15	Inconsistent Data Types in Column Sequence	1	1/14
16	Missing Column in a Sequence of Columns	2	2/14
17	Primary- and Unique-key Constraints on the Same Columns	6	1/14
18	Redundant Indices	249	13/14
19	Too Short Column Names	6	1/14
20	Too Many Text Columns in a Table	4	3/14
21	Foreign-Key Without Index	2408	2/14
22	Primary-Key Columns Not Positioned First	1179	11/14
23	Use of Reserved Words From SQL	1726	13/14
24	Different Data Types for Columns With the Same Name	576	12/14
25	Use of Special Characters in Identifiers	19814	1/14
26	Table Islands	421	14/14
27	Too Large Varchar Columns	264	5/14

Table 7: Metadata Rules.

#	Data Rule	C	S
28	Duplicate Rows in a Table	17	6/9
29	Storing Lists in Varchar Columns	5	2/9
30	Wrong Representation of Boolean Values	668	4/9
31	Defined Primary Key is not a Minimal Key	0	0/9
32	Redundant Columns	156	7/9
33	All Values Equals the Default Value	126	4/9
34	Not-Null Columns Containing Many Empty Strings	2265	8/9
35	Numbers or Dates Stored in Varchar Columns	767	7/9
36	Empty Tables	1372	8/9
37	Mixture of Data Types in Text Columns	812	8/9
38	Columns With Only One Value	36	9/9
39	All Values Differ From the Default Value	391	8/9
40	Inconsistent Casing of First Character in Text Columns	425	8/9
41	Unnecessary One-to-One Relational Tables	0	0/9
42	Column Values from a Small Domain	113	9/9
43	Large Unfilled Varchar Columns	388	9/9
44	Missing Not-Null Constraints	312	8/9
45	Column Containing Too Many Nulls	299	7/9

Table 8: Data Rules.

F. SAMPLE RULE CODE

This section shows the code for the rule "Nullable and Unique Columns". Additionally it shows how to implement configurable fields in a rule.

Figure 15 shows the code for the rule. It extends the abstract class `BaseSchemaRule` that requires two properties and one method to be overridden: `Name`, `Severity`, and `Execute`. The `Name` property specifies the name of the rule that is visible in the *DBLint* rule selection window, as well as in the *Rules* view in the report. The `Severity` property specifies the default severity of the rule that can be configured in the user interface. Finally, the `Execute` method is the actual rule implementation.

Line 1-17 is boilerplate code and will not be discussed further. Line 18-28 iterates over all columns that are both unique and nullable; and Line 20-27 reports an issue for each of these columns. Note how the description of the issues is handled in line 22-25. The `Description` object takes a string similar to a format string, but its parameters may be objects such as tables and columns.

F.1 Configurability

Figure 16 shows how the configurable options from the rule "Too big index" are implemented. A special class, `Property<T>` is used that takes a type parameter `T`. `T` can be any of the following: `bool`, `int`, `float`, `string`, or `Severity`.

The constructor for `Property<T>` requires three arguments: The title of the property, the default value (of type `T`), the description and optionally a function used to validate values.

These properties are extracted using Reflection and configurable via the user interface. Additionally, *DBLint* persists the configured fields.

```
1  public class NullableAndUnique : BaseSchemaRule
2  {
3    public override string Name
4    {
5      get { return "Nullable and Unique Columns"; }
6    }
7
8    // The default severity of this rule. Can be configured in the GUI.
9    protected override Severity Severity
10   {
11     get { return Severity.Medium; }
12   }
13
14   public override void Execute(Database database,
15     IIssueCollector issueCollector,
16     IProviderCollection providers)
17   {
18     foreach (var column in database.Columns.Where(c => c.IsNullable && c.Unique))
19     {
20       var issue = new Issue(this, this.DefaultSeverity.Value);
21       issue.Name = "Nullable and Unique Column";
22       issue.Description = new Description(
23         "Column '{0}' in table {1} is both nullable and unique",
24         column,
25         column.Table);
26       issue.Context = new ColumnContext(column);
27       issueCollector.ReportIssue(issue);
28     }
29   }
30 }
```

Figure 15: The actual code used in *DBLint* for the rule "Nullable and Unique Columns".

```
Property<int> MaxSize =
  new Property<int>("Maximum Key Size",
                    200,
                    "The maximum number of bytes allowed [...]",
                    v => v > 0);

Property<int> MaxColumns =
  new Property<int>("Maximum Columns in Index",
                    7,
                    "The maximum number of columns allowed [...]",
                    v => v > 0);

Property<int> MaxColumnsUnique =
  new Property<int>("Maximum Columns in Unique Index",
                    5,
                    "The maximum number of columns allowed [...]",
                    v => v > 0);

Property<int> VarcharSizeReductionFactor =
  new Property<int>("Estimated Varchar Fill Rate (%)",
                    20,
                    "The average fill rate of varchars [...]",
                    v => v >= 0 && v <= 100);
```

Figure 16: The configurable fields from the rule "Too big index". [...] indicates that a string is truncated for formatting purposes.