On the Quality of Relational Database Schemas in Open-source Software
Fabien Coelho, Alexandre Aillos, Samuel Pilot, Shamil Valeev

To cite this version:


HAL Id: hal-00742605
https://minesparis-psl.hal.science/hal-00742605
Submitted on 16 Oct 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
On the Quality of Relational Database Schemas in Open-source Software

Fabien Coelho, Alexandre Aillos, Samuel Pilot, and Shamil Valeev  
CRI, Mathématiques et Systèmes, MINES ParisTech,  
35, rue Saint Honoré, 77305 Fontainebleau cedex, France. 
fabien.coelho@mines-paristech.fr, firstname.lastname@mines-paris.org

Abstract—The relational schemas of 512 open-source projects storing their data in MySQL or PostgreSQL databases are investigated by querying the standard information schema, looking for overall design issues. The set of SQL queries used in our research is released as the Salix free software. As it is fully relational and relies on standards, it may be installed in any compliant database to help improve schemas. Our research shows that the overall quality of the surveyed schemas is poor: a majority of projects have at least one table without any primary key or unique constraint to identify a tuple; data security features such as referential integrity or transactional back-ends are hardly used; projects that advertise supporting both databases often have missing tables or attributes. PostgreSQL projects appear to be of higher quality than MySQL projects, and have been updated more recently, suggesting a more active maintenance. This is even better for projects with PostgreSQL-only support. However, the quality difference between both databases management systems is mostly due to MySQL-specific issues. An overall predictor of bad database quality is that a project chooses MySQL or PHP; while good design is found with PostgreSQL and Java. The few declared constraints allow to detect latent bugs, that are worth fixing; more declarations would certainly help unveil more bugs. Our survey also suggests that some features of MySQL and PostgreSQL are particularly error-prone. This first survey on the quality of relational schemas in open-source software provides a unique insight in the data engineering practice of these projects.

Keywords-open-source software; database quality survey; automatic schema analysis; relational model; SQL.

I. INTRODUCTION

This paper is an extended version of A Field Analysis of Relational Database Schemas in Open-source Software [1] presented at DBKDA 2011. Compared to this initial version, 512 schemas are surveyed instead of 407, which enhances the accuracy of the statistical validation of our analyses; the maintenance status of the surveyed projects was collected again as of January 2012; comments have been updated and added to reflect the new data; more detailed tables are provided about the results; the bibliography is much more thorough, with over 50 new references; an appendix describes the advice available with our schema analyzer; the paper page count, excluding the appendix, is increased from 7 to 10 pages.

In the beginning of the computer age, software was freely available, and money was derived from hardware only [2]. Then in the 70s it was unbundled and sold separately in closed proprietary form. Stallman initiated the free software movement, in 1983 with the GNU Project [3], and later the Free Software Foundation [4], which is now quite large [5][6] and expanding [7] (Predicts 2010) to implement his principle of sharing software. Such free software is distributed under a variety of licenses [8], which discuss copyright and liability. The common ground is that it must be available as source code to allow its study, change and improvement as opposed to compiled or obfuscated, hence the expression open source [9][10][11]. This induces many technical, economical, legal, and philosophical issues. Open-source software (OSS) is a subject of academic studies [12] in psychology, sociology, economics, or software engineering, including quantitative surveys. Developers’ motivation [13][14][15][16][17], but also organization [18][19][20][21][22][23][24][25][26] and profiles [27][28][29] are investigated, as well as user communities [30]: Existing economic frameworks [31] are used to analyze the phenomenon, as well as the influence of public policies [32]. Research focusing on software engineering issues can also be found. The development of the Apache web server popular [33] is compared to non-OSS projects [34] and its user assistance is analyzed [35]. Quantitative studies exist about code quality in OSS [36][37][38][39][40] and its dual, static analysis to uncover bugs [41][42]. Database surveys are available about market shares [43], or server exposure security issues [44]. This study is the first survey on the quality of relational database schemas in OSS. It provides a unique insight in the data engineering practice of these projects.

Codd’s relational model [45] is an extension of the set theory to relations (tables) with attributes (columns) in which tuple elements are stored (rows). Elements are identified by keys, which can be used by tuples to reference one another between relations. The relational model is sound, as all questions (in the model) have corresponding practical answers and vice versa: the tuple relational calculus describes questions, and the mathematically equivalent relational algebra provides their answers. It is efficiently implemented by many commercial and open-source software such as Oracle, DB2 or SQLite. The Structured Query Language (SQL [46]) is available with most relational database systems, although the detailed syntax often differs in subtle and incompatible ways. The standardization effort also includes the information schema [47], which provides metadata about the schemas of databases through relations.

The underlying assumption of our study is that applications store precious transactional user data, thus should be kept consistent, non redundant, and easy to understand. We think that
database features such as key declarations, referential integrity and transaction support help achieve these goals. In order to evaluate the use of database features in open-source software, and to detect possible design or implementation errors, we have developed a tool to analyze automatically the database structure of an application by querying its information schema and generating a report, and we have applied it to 512 open-source projects. The notion of the quality of a database schema design is quite elusive, as shown in Burkett's overview [48], with a lot of focus on qualitative assessments. Key criteria such as understandability, simplicity, expressiveness, maintainability or evolvability are hard to transform into basic objective metrics. A review process has been proposed to evaluate the quality of relational schemas [49], at the price of mostly manual investigations by field experts. Some quality focus on the conceptual schema and compare alternative models [50] [51] by recognizing patterns. Following MacCabe's metric to measure automatically program complexities [52] [53] [54], several metrics address data models [55] [56] or database schemata either in the relational [57] [58] or object relational [59] models, including experimental validations [60]. These metrics rely on information not necessarily available from the database concrete schemas. Moreover, such approach help compare two schemas that model the same application domain, but are less useful when used about unrelated schemas. We have rather followed the dual and pragmatic approach [61], which is not to try to do an absolute and definite measure of the schema, but rather to uncover issues based on static analyses. Thus, the measure is relative to the analyses performed and results change when more are added. Static analysis on user application codes (not simply the schema) could also be used to help uncover hidden constraints in a schema (for instance, a join between two tables suggests a possible foreign key) and to use them to improve data quality [62], but this is beyond our simple approach.

The remainder of this paper is organized as follows: Section II presents the methodology used in this study. We describe our tool, our rating strategy and the statistical validation used on the assertions derived from our analyses; Section III lists the projects by category and technology, and discusses similarities and differences depending on whether they run on MySQL or PostgreSQL; Section IV describes the results of our survey, with quite a poor overall quality of projects, as very few database schemas do not raise error-rated advices; Section V gives our conclusive thoughts.

II. METHODOLOGY

Our Salix automatic analyzer [63], is based on the information schema provided by standard databases. It is open-source, and its schema itself is included in this survey. In this Section, we discuss the queries, then describe the available advices, before presenting the statistical validation used.

A. Information schema queries

Our analyses are performed automatically by SQL queries on the databases metadata using the standard information schema. This relational schema stores information about the databases structure, including catalogs, schemas, tables, attributes, types, constraints, roles, permissions, etc. The set of SQL queries used for this study are released as the Salix free software. It is based on pg-advisor [64], a PostgreSQL-specific proof of concept prototype developed in 2004. Some checks are inspired by Currier [65], Baron [66] and Berkus [67] or similar to Boehm [68]. Note that the aim is quite different from tools which focus on advising database administrators, for instance about index creation [69]. Salix creates specific tables for each advice by querying the information schema, and then aggregates the results in summary tables in a dedicated schema. It is fully relational in its conception [70]; there is no programming other than SQL queries, but a small shell driver which creates the advices, shows or reports them in some detail to the interested user, and finally drops them out of the database. Because of performance issues when querying heavily metadata relations, the tool relies on tables which are materialized views, although using views directly would have been a preferred option if possible. The development of Salix uncovered multiple issues with both implementations of the information schema.

B. Advice classification and project grading

The 47 issues reported by our SQL queries from the standard information schema are named advices, as the user is free to ignore them. Although the performed checks are basic and syntactic, we think that they reflect the quality of the schemas. For instance, style advices help with understandability, and consistency advices help with maintainability. A detailed list of advices currently implemented in our tool is available [71]. Each advice has a category (19 design, 13 style, 6 consistency, 4 version, 5 system), a severity (7 errors, 21 warnings, 14 notes, 5 informations), and a level (1 raised per database, 10 per schema, 27 per relation, 7 per attribute, 2 per role). The severity classification is arbitrary and must be evaluated critically by the recipient: most of them should be dealt with, but in some cases they may be justifiable. For instance, having a mix of MySQL back-end engines is considered inconsistent and tagged as an error, although it may be necessary to do so because some features (e.g. full-text indexes) are only available with some back-ends. Moreover, detected errors do not imply that the application is not fully functional from a user perspective.

The 19 design advices focus on detecting design errors from the information available in the metadata. Obviously, semantic error, say an attribute is in the wrong relation, cannot be guessed without understanding the application and thus are out of reach of our automatic analysis. We rather focus on primary and foreign key declarations, or warn if they are missing. The rate of non-null attributes is also checked, with the underlying assumption from our experience that most data are mandatory in a relation. We also check the number of attributes so as to detect a possible insufficient conception effort.

The 13 style advices focus on relation and attribute names. Whether a name is significant in the context cannot be checked, so we simply look at their length. Short names are discouraged as they would rather be used as aliases in
queries, with the exception of id and pk which are accepted as attributes. We also check that the same name does not represent differently typed data, to avoid confusing the user.

The 6 consistency advices checks for type and schema consistency in a project, such as type mismatches between a foreign key and the referenced key. As databases may also implements some of these checks, it is possible that some cases cannot be triggered.

The 4 version advices focus on database-specific checks, such as capabilities and transaction support, as well as homogeneous choices of back-end engines in a project. This category could also check the actual version of a database used looking for known bugs or obsolescence. Only MySQL-specific version advices are currently implemented.

Finally, the 5 system advices, some of which PostgreSQL-specific, check for weak passwords, and key and index issues.

These advices aim at helping the schema developer to improve its relational design. We also use them in our survey to grade projects with a mark from 0 to 10, computed by removing points each time an advice is raised, taking more points if the severity is high, and flooring the result to avoid negative grades. The grading process is normalized using the number of possible occurrences, so that larger projects do not receive lower marks just because of the likelihood of having more issues for their size. Also, points are not removed twice for the same issue: for instance, if a project does not have a single foreign key, the same issue will not be raised again on every tables. Advices not relevant to our open-source database schema survey, e.g., weak password checks, were deactivated.

C. Survey statistical validation

The data collected suggest the influence of some parameters on others. These results deal with general facts about the projects (say foreign keys are more often used with PostgreSQL) or about their grading (say MySQL projects get lower marks). In order to determine significant influences, we applied Pearson’s chi-square tests [72] to compute probabilistic degrees of certainty. Beware that these statistical validations hold for our data set only. It is possible that some unwanted bias in the project selection process makes statements that are in reality false appear true, and vice versa. We followed a one project one vote principle in our analyses, so that these validations do not take into account the projects sizes or popularity. Also, our software, as all software, may include bugs with unexpected consequences. Each checked assertion is labeled with an expression indicating the degree of certainty of the influence of one parameter on an other:

- **very sure** The probability is 1% or less to get a result as or more remote from the average. Thus we conclude that there is an influence, with a very high degree of certainty.

- **rather sure** The probability of getting such a result is between 1% and 5% (the usual statistical threshold). Thus there is an influence, with a high degree of certainty.

- **marginally sure** The probability is between 5% and 25%: such a result may have been obtained even if there is no influence. The statement must be taken with a pinch of salt.

- **not sure** The probability is over 25%, or there is not enough available data to compute it. The test cannot assert that there is a significant influence. Obviously, no such assertion was included in this survey.

The rational for choosing Pearson’s chi-square test is that it does not make any assumption about the distribution of values. However, it is crude, and possibly interesting and somehow true results may not be validated. Moreover, the test requires a minimal population, which is not easily reached on our small data set especially when criteria are crossed. Finally, it needs to define distinct populations: for grades or sizes, these populations are cut at the median value in order to perform the test on balanced partitions.

We also computed a correlation matrix to look for possible inter-parameter influence. The result suggested that the parameters are pretty independent beyond the obvious links (say the use of a non-transactional back-end is correlated with isolated tables), and did no help uncover significant new facts.

III. Projects

We discuss the projects considered in this study, grouped by categories, technologies, sizes and release dates. We first present how projects were selected, and then an overview.

A. Project selection

We have downloaded 512 open-source projects starting in the first semester of 2008, adding to our comparison about every project that uses either MySQL [73] or PostgreSQL [74] that we could find and install with reasonable time and effort. The database schemas included in this study are derived from a dump of the database after installation, or from the creation statements when found in the sources. These projects were discovered from various sources: lists and comparisons of software on Wikipedia (Software lists about: photo galleries, content management systems, Internet forums, reference management, issue tracking systems, wikis, social networking, church management, student information systems, accounting, weblog, Internet relay chat, health-care, genealogy, etc.) and other sites; package dependencies from Linux distributions such as Debian [75] or Ubuntu [76] requiring databases; security advisories mentioning SQL [77]; searches on SourceForge [78] which use SQL databases.

Some projects were fixed manually because of various issues, such as: the handling of double-dash comments by MySQL, attribute names (e.g., out) rejected by MySQL, bad foreign key declarations or other incompatibilities detected when the projects were forced to use the InnoDB back-end instead of MyISAM, or even some PostgreSQL table definitions including a MySQL specific syntax that were clearly never tested. A particular pitfall of PostgreSQL is that by default syntax errors in statements from an SQL script are ignored and the interpreter simply jumps to the next statement. When installing a project, the flow of warnings often hides these errors. Turning off this feature requires modifying the script, as no command option disables it. More than a dozen PostgreSQL projects contained this kind of issues, which resulted in missing tables or ignored constraint declarations.
### B. Overview of projects

We have studied the relational schemas of 512 (see appendix for the full list) open-source projects based on databases: 482 of these run with MySQL, 126 with PostgreSQL, including 96 on both. A project supporting PostgreSQL is very likely to support also MySQL (76%), although the reverse is not true (only 19%) (very sure), outlining the relative popularity of these tools. Only 30 projects are PostgreSQL specific. Although there is no deliberate bias in the selection process described in the previous section, where we aimed at completeness, some implicit bias remain nevertheless: for instance, as we can speak mostly English and French, we found mostly international projects advertised in these tongues; Table I shows main project categories, from the personal mundane (game, homepage) to the professional serious (health-care, accounting, system). Table II shows the same for project technologies. Projects in rare categories or using rare technologies do not appear in these cut-off tables. The result is heavily slanted towards PHP web applications (77%), which seems to reflect the current trend of open-source programming as far as the number of projects is concerned, without indication of popularity or quality. The ratio of PHP projects increases from PostgreSQL only support (26%) to both database support (58%) (very sure) to MySQL only support (86%) (very sure): PHP users tend to choose specifically MySQL, possibly because of traditional LAMP (Linux, Apache, MySQL, PHP) setups advertised with PHP programming. For instance, a search on the Amazon website in January 2012 returns 18 times more results with PHP MySQL compared to PHP PostgreSQL.

The survey covers 18993 tables (MySQL 13494, Post-
Table III

<table>
<thead>
<tr>
<th>Advice</th>
<th>Lvl</th>
<th>Cat</th>
<th>Sev</th>
<th>MySQL Proj</th>
<th>MySQL Adv</th>
<th>PostgreSQL Proj</th>
<th>PostgreSQL Adv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema without any FK</td>
<td>sch</td>
<td>design</td>
<td>error</td>
<td>425</td>
<td>88</td>
<td>425</td>
<td>88</td>
</tr>
<tr>
<td>Tables without PK nor Unique</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>262</td>
<td>54</td>
<td>1521</td>
<td>11</td>
</tr>
<tr>
<td>FK type mismatch</td>
<td>table</td>
<td>consist.</td>
<td>error</td>
<td>2</td>
<td>0</td>
<td>170</td>
<td>0</td>
</tr>
<tr>
<td>Backend engine inconsistency</td>
<td>table</td>
<td>version</td>
<td>error</td>
<td>30</td>
<td>6</td>
<td>306</td>
<td>0</td>
</tr>
<tr>
<td>FK length mismatch</td>
<td>table</td>
<td>consist.</td>
<td>error</td>
<td>4</td>
<td>0</td>
<td>260</td>
<td>0</td>
</tr>
<tr>
<td>Integer PK but no other key</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>437</td>
<td>90</td>
<td>7470</td>
<td>55</td>
</tr>
<tr>
<td>Homonymous heterogeneous attributes</td>
<td>att.</td>
<td>style</td>
<td>warn</td>
<td>7</td>
<td>5</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>Unsafe backend engine used in schema</td>
<td>sch.</td>
<td>version</td>
<td>error</td>
<td>433</td>
<td>89</td>
<td>433</td>
<td>89</td>
</tr>
<tr>
<td>Attribute count per table over 40</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>98</td>
<td>20</td>
<td>220</td>
<td>1</td>
</tr>
<tr>
<td>Isolated Tables</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>30</td>
<td>6</td>
<td>979</td>
<td>7</td>
</tr>
<tr>
<td>Tables without PK but with Unique</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>117</td>
<td>24</td>
<td>405</td>
<td>3</td>
</tr>
<tr>
<td>Unique nullable attributes</td>
<td>att.</td>
<td>design</td>
<td>warn</td>
<td>73</td>
<td>15</td>
<td>261</td>
<td>0</td>
</tr>
<tr>
<td>Nullable attribute rate over 80%</td>
<td>sch.</td>
<td>design</td>
<td>warn</td>
<td>34</td>
<td>7</td>
<td>34</td>
<td>7</td>
</tr>
<tr>
<td>Redundant indexes</td>
<td>table</td>
<td>system</td>
<td>warn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Attribute name length too short</td>
<td>att.</td>
<td>style</td>
<td>warn</td>
<td>27</td>
<td>5</td>
<td>91</td>
<td>0</td>
</tr>
<tr>
<td>Large PK referenced by a FK</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>10</td>
<td>2</td>
<td>118</td>
<td>0</td>
</tr>
<tr>
<td>Table name length too short</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>16</td>
<td>3</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>Composite Foreign Key</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>5</td>
<td>1</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>FK not referencing a PK</td>
<td>table</td>
<td>design</td>
<td>warn</td>
<td>2</td>
<td>0</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Redundant FK</td>
<td>table</td>
<td>system</td>
<td>warn</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Non-integer Primary Key</td>
<td>table</td>
<td>design</td>
<td>note</td>
<td>268</td>
<td>55</td>
<td>2261</td>
<td>16</td>
</tr>
<tr>
<td>MySQL is used</td>
<td>base</td>
<td>version</td>
<td>note</td>
<td>482</td>
<td>100</td>
<td>482</td>
<td>100</td>
</tr>
<tr>
<td>Attribute count per table over 20</td>
<td>table</td>
<td>design</td>
<td>note</td>
<td>230</td>
<td>47</td>
<td>684</td>
<td>5</td>
</tr>
<tr>
<td>Tables with Composite PK</td>
<td>table</td>
<td>design</td>
<td>note</td>
<td>196</td>
<td>40</td>
<td>1781</td>
<td>13</td>
</tr>
<tr>
<td>Attribute name length quite short</td>
<td>att.</td>
<td>style</td>
<td>note</td>
<td>201</td>
<td>41</td>
<td>748</td>
<td>0</td>
</tr>
<tr>
<td>Attribute named after its table</td>
<td>att.</td>
<td>style</td>
<td>note</td>
<td>139</td>
<td>28</td>
<td>3114</td>
<td>2</td>
</tr>
<tr>
<td>Table without index</td>
<td>table</td>
<td>system</td>
<td>note</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Nullable attribute rate in 50-80%</td>
<td>sch.</td>
<td>design</td>
<td>note</td>
<td>76</td>
<td>15</td>
<td>76</td>
<td>15</td>
</tr>
<tr>
<td>Table name length quite short</td>
<td>table</td>
<td>design</td>
<td>note</td>
<td>70</td>
<td>14</td>
<td>102</td>
<td>0</td>
</tr>
<tr>
<td>Table with a single attribute</td>
<td>table</td>
<td>design</td>
<td>note</td>
<td>74</td>
<td>15</td>
<td>419</td>
<td>3</td>
</tr>
<tr>
<td>Mixed attribute name styles</td>
<td>table</td>
<td>style</td>
<td>note</td>
<td>115</td>
<td>23</td>
<td>1007</td>
<td>7</td>
</tr>
<tr>
<td>Mixed table name styles</td>
<td>sch.</td>
<td>style</td>
<td>note</td>
<td>51</td>
<td>10</td>
<td>261</td>
<td>54</td>
</tr>
<tr>
<td>Attribute name length short</td>
<td>att.</td>
<td>style</td>
<td>info</td>
<td>326</td>
<td>67</td>
<td>2911</td>
<td>2</td>
</tr>
<tr>
<td>Unsafe backend engine used on table</td>
<td>table</td>
<td>version</td>
<td>info</td>
<td>433</td>
<td>89</td>
<td>10423</td>
<td>77</td>
</tr>
<tr>
<td>Nullable attribute rate in 20-50%</td>
<td>sch.</td>
<td>design</td>
<td>info</td>
<td>137</td>
<td>28</td>
<td>137</td>
<td>28</td>
</tr>
<tr>
<td>Table name length short</td>
<td>table</td>
<td>style</td>
<td>info</td>
<td>136</td>
<td>28</td>
<td>258</td>
<td>1</td>
</tr>
</tbody>
</table>

The per-category tables and attributes-per-table counts show that accounting, health-care and market projects seem more ambitious than other categories (marginally sure). The per-technology analysis counts suggests that Perl, Python and Java projects are larger than those based on other technologies (marginally sure).

These projects are mostly recent, at least according to their status at an arbitrary common reference date chosen as March 31, 2009: 310 (60%) were updated in the last year, including 179 (34%) in the last six months, and the others are either obsolete or stable. The rate of recently updated projects raises from MySQL-only projects (55%) to projects with both support (73%) (very sure) or with PostgreSQL support at (76%) (very sure), but there is no significant difference on the recent maintenance figures between projects that are PostgreSQL-only and projects with both databases support.

New data about the status of projects were collected on January 9, 2012. We could not find 69 projects in this new survey (61 MySQL-only, 1 PostgreSQL-only and 7 with both support). Moreover, 153 projects are stale, that is not
updated between the 2009 and 2012 data (128 MySQL-only, 6 PostgreSQL-only and 19 with both support). Nearly half of the MySQL projects are stale or lost, while it is only one quarter of the PostgreSQL projects. MySQL-only projects are more often lost or stale than others in 2012 (very sure), and it is still true for MySQL projects compared to PostgreSQL-only projects (rather sure). More generally, on these new data, MySQL-only projects are less maintained than others (very sure), and it is still true compared to projects with both support (very sure) and compared to projects with PostgreSQL-only support (rather sure). There are about six months (180 days) between the median update date of MySQL-only projects and PostgreSQL-only projects. Even if we ignore lost and stale projects to focus on projects that were indeed updated in the 2012 data, PostgreSQL-only projects were more recently updated than others (rather sure). Yet again, there is no significant update status difference between projects with PostgreSQL support and projects that support both databases on the 2012 data. To conclude, the maintenance of PostgreSQL projects seems more intense: projects that include PostgreSQL support were updated more recently both in 2009 and in 2012.

IV. Survey Results

We now analyze the open-source projects of our survey by commenting actual results on MySQL and PostgreSQL, before comparing them. Table III summarizes the advices raised for MySQL and PostgreSQL applications. The first four columns give the advice title, level, category and severity. Then four columns for each database list the results. The first two columns hold the number of projects (i.e. schema) tagged and the overall rate. The last two columns give the actual number of advices and rate, which varies depending on the level. A per-project aggregate is also available online [71].

A. Primary keys

A majority of MySQL projects (262 – 54%) have at least one table without neither a primary key nor a unique constraint, and this is even worse with PostgreSQL projects (76 – 60%). The certainty of the observation (rather sure) on MySQL-only vs PostgreSQL-only is low because of the small number of projects using the later. As 11% of all MySQL tables and 18% of all PostgreSQL tables do not have any key, the view of relations as sets is hindered as tuples are not identified, and data may be replicated without noticing.

A further analysis gives some more insight. For MySQL, 41% of tables without key do have some KEY option for indexes, but without the UNIQUE or PRIMARY keyword that makes it a key. Having KEY not always declaring a key was clearly a bad design choice. A little 5% of tables without key have an auto increment attribute, which suggest uniqueness in practice, but is not enforced. Also, the missing key declaration often seems to be composite. Some tables without key declarations are intended as one tuple only, say to check for the version of the schema or configuration of the application. Similarly, 28% of PostgreSQL tables without key have an index declared. Moreover, 22% have a SERIAL (auto incremented) attribute: Many designers seem to assume wrongly that SERIAL implies a key. A comment found in the SQL Grey project source suggests that some keys are not declared because of MySQL key size limits.

A simple integer primary key is provided on 61% of tables, with a significantly decreasing rate from MySQL-only (65%) to both database support (62%) (rather sure) down to PostgreSQL-only support (39%) (very sure). If these primary keys were non-semantic numbers to identify tuples, one would expect at least one other key declared on each table to identify the underlying semantic key. However it is not the case: most (85%) of these tables do not have any other key. When a non simple primary key is available, it is either based on another type or a composite key. The composite keys are hardly referenced, but as the foreign keys are rarely declared one cannot be sure, as shown in the next section.

B. Referential integrity

Foreign keys are important for ensuring data consistency in relational databases. They are supported by PostgreSQL, and by MySQL but with some back-end engines only. In particular, the default MySQL back-end does not support foreign keys, and this feature was deemed noxious in previous documentations: Version 3.23 includes a Reasons NOT to Use Foreign Keys constraints Section arguing that they are only useful to display diagrams, hard to implement and terrible for performance. Foreign key constraints are introduced with the InnoDB engine starting with MySQL 3.23.44 in January 2001. Although the constraints are ignored by the default MyISAM engine, the syntax is parsed, and triggers the creation of indexes. Version 3.1 documentation has a Foreign Keys Section praising the feature, as it offers benefits, although it slows down the application. Caveats describe the inconsistencies that may result from not using transactions and referential integrity. From a pedagogical perspective, this is a progress.

Foreign key constraints have long been a missing or avoided feature in MySQL and this seems to have retained momentum in many projects, as it is not supported by the default engine: few MySQL projects (57 – 11% of all projects, but 72% of those with InnoDB) use foreign key constraints. The foreign key usage rate is slightly higher (20%) when considering projects supporting both databases (marginally sure).

Among MySQL projects, 403 (83%) use only the default MyISAM back-end engine, thus do not have any foreign key checks enabled. In the remainder, 49 (10%) use only InnoDB, and 30 (6%) use a combination of both. More projects (21 – 21%) rely on InnoDB among those supporting both MySQL and PostgreSQL (marginally sure). A third of InnoDB projects (30 – 37%) are not consistent in their engine choice: 34% of tables use MyISAM among the 79 InnoDB projects. A legitimate reason for using MyISAM tables in an InnoDB project is that full-text indexes are only available with the former engine. However, this only applies to 11 tables in 6 projects, all other 1441 MyISAM tables in InnoDB projects are not justified by this. A project may decide to store transient data in an unsafe engine (e.g., memory) for performance reason. However, this case is rare, as it represents only 15 tables in 8 projects. About 26% of tables use MyISAM as a default implicit choice in
InnoDB projects, similar to 28% when considering all MySQL projects. Some engine inconsistencies seem due to forgotten declarations falling back to the default MyISAM engine.

We have forced the InnoDB back-end engine for all MySQL projects: 22 additional projects declare 92 new foreign key constraints previously ignored. These new foreign keys are very partial, targeting only some tables. They allow to uncover about two dozen issues, either because the foreign key declaration was failing (say from type errors detected by MySQL) or thanks to analyses from our tool. Additional checks based on foreign keys cannot be raised on schemas that do not declare any of them. Thus isolated tables warnings must be compared to the number of projects that do use referential constraints: 30 – 52% of these seem to have forgotten at least some foreign keys, and it is actually the case by checking some of these projects manually.

The foreign key usage is better with PostgreSQL projects, although it is still a minority (56 projects – 44%). This rate is close to the foreign key usage of MySQL projects when considering InnoDB projects only. It gives a better opportunity for additional advices to be checked. The foreign key usage rate raises significantly to 74% when considering PostgreSQL-only projects vs dual support projects (very sure).

On the very few projects with partial foreign key declarations, several of these declaration reveal latent bugs, including type mismatch, typically CHAR targeting a VARCHAR or vice versa, or different integers, and type length mismatch, usually non matching VARCHAR sizes. We found 23 such bugs out of the small 1979 declared MySQL attribute constraints, and 163 among the 4424 PostgreSQL constraints. The rate is greater for PostgreSQL, possibly helped by the use of SERIAL which may be considered as a primary key by developers without being declared as such. There are also 153 important warnings related to foreign keys raised for MySQL, and 265 for PostgreSQL. If this error ratio is extrapolated to the number of tables, hundreds of additional latent bugs could be detected using the missing referential constraints.

C. Miscellaneous issues

More issues were found about style, attribute constraints and by comparing projects with dual database support.

There is 13669 noticeable style issues raised from our analyses (7640 for MySQL, 6029 for PostgreSQL), relating to table or attribute names, including a number of one-letter attribute names or two-letters table names. The id attribute name is used in the slash project with up to 6 different types, mixing various integers and fixed or variable length text types. In PHP/PPetition, a date attribute has types DATE, DATETIME or VARCHAR. 81% of MySQL projects and 78% of PostgreSQL have such style issues.

Many projects do not bother with NOT NULL attribute declarations: 110 MySQL projects (22%) and 58 PostgreSQL projects (46%) have over half of their attributes null-able. This does not reflect the overall use of constraints: for MySQL, the average number of key-related constraints per table is 1.07 (from BOARDPLUS 0.00 to jwhoisserver 3.57), while for PostgreSQL it is 1.24 (from ANDROMEDA 0.00 to adenpiere 4.25). Project ANDROMEDA is astonishing: there is not a single constraint declared (no primary key, no foreign key, no unique, no not null) on the 180 tables, although there are a number of non-unique indexes and of sequences.

It is interesting to compare the schemas of the 96 projects available with both databases. This dual support must not be taken at face value: PostgreSQL support is often an afterthought and is not necessarily functional, including project such as ELGG, TAGADASH, QUICKTEAM OF TIKIWIKI where some PostgreSQL table declarations use an incompatible MySQL syntax; 38 (39%) projects have missing tables or attributes between the MySQL and PostgreSQL versions: 398 tables and 191 individual attributes are missing or misspelled one side or another. Among the missing tables, 73 look like some kind of sequence, and thus might be possibly legitimate, although why the auto increment feature was not satisfactory is unclear. At the minimum, the functionalities are not the same between the MySQL and PostgreSQL versions of these projects.

D. Overall quality

We have computed a synthetic project quality evaluation ranging from 10 (good) to 0 (bad) by removing points based on advice severity (error, warning, notice), level (schema, table, attribute) and project size. The MySQL projects quality average is 4.4 ± 1.4 (from 9.5 JWHOISSERVER to 0.0 MANTIS), significantly lower than PostgreSQL 5.4 ± 1.8 (from 9.4 COMICS to 0.0 NURPAWIKI) (very sure). This does not come as a surprise: most MySQL projects choose the default data-safe MyISAM engine, hence incur a penalty. Also, the multiplicity of MySQL back-ends allows the user to mix them unintentional, what is not possible with PostgreSQL. When all MySQL-specific advices are removed, the quality measure is about the same for both databases. However, as PostgreSQL schemas provide more information about referential integrity constraints, they are also penalized as more advices can be raised based on the provided additional information. For projects which support both databases, the grade’s correlation is significant and positive (0.55), which is logical as the same style warnings are triggered on both sides.

Table [V] shows the projects per quality decile. The PostgreSQL-only project quality is more spread than MySQL projects (very sure). Table [V] compares the quality of projects according to size, with small up to 9, medium up to 29, and large otherwise. The quality is quite evenly distributed among sizes, which suggests that our effort to devise a size-neutral grading succeeded. Table [VI] compares quality based on the project categories. The number of projects in each category is too small to draw deep conclusions. Table [VII] addresses the technology used in the project: Java and Python lead while C, PHP and Ruby are near bottom. PHP projects take less care of their relational design (rather sure), but this may be explained by the fact that MySQL is used more often in these projects, and that an unsafe engine is selected more often (very sure). Yet again, the very small count of projects with some of the technologies do not allow to draw deep conclusion about them. Finally, Table [VIII] and Table [IX] show that quality evaluation does not change much depending whether projects are updated more often.
### TABLE IV
**QUALITY PER DECILE**

<table>
<thead>
<tr>
<th>Size</th>
<th>nb</th>
<th>%</th>
<th>avg</th>
<th>σ</th>
<th>min</th>
<th>med</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>181</td>
<td>38</td>
<td>4.7</td>
<td>1.4</td>
<td>0.0</td>
<td>4.5</td>
<td>9.5</td>
</tr>
<tr>
<td>medium</td>
<td>164</td>
<td>34</td>
<td>4.2</td>
<td>1.3</td>
<td>0.0</td>
<td>4.3</td>
<td>8.7</td>
</tr>
<tr>
<td>large</td>
<td>137</td>
<td>28</td>
<td>4.3</td>
<td>1.4</td>
<td>0.0</td>
<td>4.4</td>
<td>8.2</td>
</tr>
</tbody>
</table>

### TABLE V
**QUALITY PER SIZE**

<table>
<thead>
<tr>
<th>Category</th>
<th>nb</th>
<th>%</th>
<th>avg</th>
<th>σ</th>
<th>min</th>
<th>med</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>irc</td>
<td>12</td>
<td>2</td>
<td>5.1</td>
<td>1.3</td>
<td>2.0</td>
<td>5.4</td>
<td>7.0</td>
</tr>
<tr>
<td>mail</td>
<td>15</td>
<td>3</td>
<td>4.4</td>
<td>1.7</td>
<td>1.7</td>
<td>4.7</td>
<td>8.4</td>
</tr>
<tr>
<td>project</td>
<td>23</td>
<td>5</td>
<td>4.3</td>
<td>1.4</td>
<td>0.0</td>
<td>4.6</td>
<td>6.2</td>
</tr>
<tr>
<td>system</td>
<td>47</td>
<td>10</td>
<td>4.5</td>
<td>1.4</td>
<td>0.0</td>
<td>4.5</td>
<td>9.5</td>
</tr>
<tr>
<td>game</td>
<td>16</td>
<td>3</td>
<td>4.4</td>
<td>2.0</td>
<td>0.9</td>
<td>4.5</td>
<td>9.1</td>
</tr>
<tr>
<td>blog</td>
<td>27</td>
<td>6</td>
<td>4.4</td>
<td>0.9</td>
<td>2.5</td>
<td>4.5</td>
<td>7.2</td>
</tr>
<tr>
<td>forum</td>
<td>19</td>
<td>4</td>
<td>4.3</td>
<td>0.9</td>
<td>2.4</td>
<td>4.4</td>
<td>5.7</td>
</tr>
<tr>
<td>cms</td>
<td>82</td>
<td>17</td>
<td>4.2</td>
<td>1.1</td>
<td>0.0</td>
<td>4.3</td>
<td>8.3</td>
</tr>
<tr>
<td>homepage</td>
<td>12</td>
<td>2</td>
<td>4.1</td>
<td>0.9</td>
<td>3.0</td>
<td>4.1</td>
<td>5.9</td>
</tr>
<tr>
<td>market</td>
<td>22</td>
<td>5</td>
<td>4.0</td>
<td>1.4</td>
<td>1.8</td>
<td>4.0</td>
<td>8.2</td>
</tr>
<tr>
<td>accounting</td>
<td>12</td>
<td>2</td>
<td>4.4</td>
<td>1.9</td>
<td>1.9</td>
<td>5.6</td>
<td>7.5</td>
</tr>
</tbody>
</table>

### TABLE VI
**QUALITY PER PROJECT MAIN CATEGORIES**

<table>
<thead>
<tr>
<th>Techno.</th>
<th>nb</th>
<th>%</th>
<th>avg</th>
<th>σ</th>
<th>min</th>
<th>med</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>python</td>
<td>5</td>
<td>1</td>
<td>5.9</td>
<td>2.0</td>
<td>3.7</td>
<td>6.2</td>
<td>8.2</td>
</tr>
<tr>
<td>sql</td>
<td>7</td>
<td>1</td>
<td>4.0</td>
<td>2.5</td>
<td>0.0</td>
<td>5.3</td>
<td>5.9</td>
</tr>
<tr>
<td>java</td>
<td>16</td>
<td>3</td>
<td>4.8</td>
<td>2.8</td>
<td>0.0</td>
<td>5.2</td>
<td>9.5</td>
</tr>
<tr>
<td>c++</td>
<td>6</td>
<td>1</td>
<td>4.8</td>
<td>1.2</td>
<td>3.3</td>
<td>4.5</td>
<td>7.0</td>
</tr>
<tr>
<td>c</td>
<td>33</td>
<td>7</td>
<td>4.6</td>
<td>1.4</td>
<td>2.0</td>
<td>4.4</td>
<td>8.4</td>
</tr>
<tr>
<td>php</td>
<td>391</td>
<td>81</td>
<td>4.4</td>
<td>1.2</td>
<td>0.0</td>
<td>4.4</td>
<td>9.1</td>
</tr>
<tr>
<td>perl</td>
<td>16</td>
<td>3</td>
<td>3.9</td>
<td>2.1</td>
<td>0.0</td>
<td>4.3</td>
<td>8.7</td>
</tr>
<tr>
<td>ruby</td>
<td>5</td>
<td>1</td>
<td>4.5</td>
<td>0.9</td>
<td>3.7</td>
<td>4.2</td>
<td>5.6</td>
</tr>
</tbody>
</table>

### TABLE VII
**QUALITY PER PROJECT MAIN TECHNOLOGIES**

<table>
<thead>
<tr>
<th>Date</th>
<th>nb</th>
<th>%</th>
<th>avg</th>
<th>σ</th>
<th>min</th>
<th>med</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>recent</td>
<td>162</td>
<td>34</td>
<td>4.3</td>
<td>1.3</td>
<td>0.0</td>
<td>4.4</td>
<td>8.6</td>
</tr>
<tr>
<td>older</td>
<td>320</td>
<td>66</td>
<td>4.4</td>
<td>1.4</td>
<td>0.0</td>
<td>4.4</td>
<td>9.5</td>
</tr>
</tbody>
</table>

### TABLE VIII
**QUALITY PER PROJECT UPDATE IN MARCH 2009**

<table>
<thead>
<tr>
<th>Date</th>
<th>nb</th>
<th>%</th>
<th>avg</th>
<th>σ</th>
<th>min</th>
<th>med</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>recent</td>
<td>112</td>
<td>23</td>
<td>4.3</td>
<td>1.3</td>
<td>0.0</td>
<td>4.5</td>
<td>7.8</td>
</tr>
<tr>
<td>older</td>
<td>155</td>
<td>32</td>
<td>4.4</td>
<td>1.5</td>
<td>0.0</td>
<td>4.5</td>
<td>9.5</td>
</tr>
<tr>
<td>stale</td>
<td>147</td>
<td>30</td>
<td>4.5</td>
<td>1.4</td>
<td>0.9</td>
<td>4.4</td>
<td>9.1</td>
</tr>
<tr>
<td>lost</td>
<td>68</td>
<td>14</td>
<td>4.2</td>
<td>1.0</td>
<td>0.0</td>
<td>4.2</td>
<td>6.3</td>
</tr>
</tbody>
</table>

### TABLE IX
**QUALITY PER PROJECT UPDATE IN JANUARY 2012**

<table>
<thead>
<tr>
<th>Date</th>
<th>nb</th>
<th>%</th>
<th>avg</th>
<th>σ</th>
<th>min</th>
<th>med</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>recent</td>
<td>41</td>
<td>33</td>
<td>5.7</td>
<td>1.4</td>
<td>0.0</td>
<td>5.6</td>
<td>7.7</td>
</tr>
<tr>
<td>older</td>
<td>52</td>
<td>41</td>
<td>5.2</td>
<td>1.9</td>
<td>0.0</td>
<td>5.3</td>
<td>9.3</td>
</tr>
<tr>
<td>stale</td>
<td>25</td>
<td>20</td>
<td>5.1</td>
<td>2.2</td>
<td>0.7</td>
<td>5.3</td>
<td>9.4</td>
</tr>
<tr>
<td>lost</td>
<td>8</td>
<td>6</td>
<td>5.5</td>
<td>2.3</td>
<td>2.0</td>
<td>5.5</td>
<td>9.0</td>
</tr>
</tbody>
</table>
V. Conclusion

This is the first survey on the quality of relational schemas in open-source software. The overall quality results are worse than envisioned at the beginning of the study. Although we did not expect a lot of perfect projects, having so few key declarations and referential integrity constraints came as a surprise. We must acknowledge that our assumption that data are precious, and that the database should help preserve its consistency by enforcing integrity constraints and implementing transactions, is not shared by most open-source projects, especially when based on MySQL and PHP. This is illustrated by bug report 15441 [79] about missing keys on tables in mediawiki, the software behind Wikipedia: it had no effect on the software after more than three years, although it triggered some discussions at the beginning of 2012.

We can only speculate about the actual reasons that explain the poor quality of the surveyed schemas in open-source projects. One way to investigate further these issues would be to collect data about and from the people who designed the relational schemas of these projects. For instance, if MySQL or PHP users are found less savvy about software development, that could account for a lower quality and maintenance of the corresponding projects. Some interesting questions could be investigated: What are their educational and professional background? Did they receive any formal education about computer programming in general? About relational database design in particular? Do they consider database design as an important issue? How are they perceiving the actual quality of their schemas, and the quality of their software? When did they started database design? For MySQL, what database engines do they use? Did the initial policy of discouraging foreign key usage influence them? We attempted to conduct such a survey by contacting some people by e-mail and encouraging them to fill a web form online. The return ratio of this survey attempt was null. This establishes the fact that schema designers in open-source software do not wish to answer such questions, with a very high degree of accuracy.

Another relevant question is whether our results would be different if we studied closed-source projects developed by payed professionals, possibly using non open-source database technologies from Oracle or Microsoft. However, accessing such data at a level compatible with statistical validation seems very difficult. If we were to believe some of our experience, the results could end up being quite similar, especially when considering PHP/MySQL projects.

It is interesting to note that the first author contributed both to the best PostgreSQL project (COMICS), and to one of the worst MySQL project (SLXBBL), which is Salix executed on its own schema. This deserves an explanation: COMICS is a small database used for teaching SQL. The normalized schema emphasizes clarity and cleanliness with a pedagogic goal in mind. Even so, the two raised warnings deserve to be fixed, although one would require an additional attribute. SLXBBL tables generate a lot of errors, because they are views materialized for performance issues. Also, they rely on MyISAM because some SQL create table statements must be compatible with both MySQL and PostgreSQL to ease the tool portability. Nevertheless, the comparison of schemas allowed to find one bug: an attribute had a different name, possibly because of a bad copy-paste.

Acknowledgement

We are indebted to Pierre Jouvelot for helping with the title and proof reading. We also thank the anonymous reviewers for their helpful remarks that we tried to address for the better of the paper.

References

APPENDIX
List of Advices

1) Schema without any FK
   Why use a relational database if data are not related at all?
   Well, that might happen...

2) No attribute in table
   There must be something in a table.

3) Tables without PK nor Unique
   All tuples must be uniquely defined to be consistent with the
   set theory. There is no unique subset of attribute which can be
   promoted as a PK.

4) Nullable attribute rate over 80%
   Warning: Most of the time, attributes should be NOT NULL.
   Too high a rate of nullable attributes may reveal that some fields
   are lacking a NOT NULL.

5) Attribute count per table over 40
   Having so many attributes in the same table may reveal the
   need for additional relations.

6) Complex Foreign Key
   As for primary keys, simple foreign keys are generally better
   design, and make updates easier.

7) FK not referencing a PK
   A Foreign Key should rather reference a Primary Key.

8) Integer PK but no other key
   A simple integer primary key suggests that some other key
   must exist in the table.

9) Isolated Tables
   In a database design, tables are usually linked together.

10) Large PK referenced by a FK
    Having large primary keys referenced by a foreign key may
    reveal data duplication, as the primary key is likely to contain
    relevant information.

11) Tables without PK but with Unique
    All tables should have a primary key to be consistent with the
    set theory. A unique constraint may be promoted as the primary
    key.

12) Attribute has a pseudo 'NULL' text default
    Possibly the NULL value was intended instead of the 'NULL'
    text.

13) Unique nullable attributes
    A unique nullable attribute may be a bad design if NULL does
    not have a particular semantic.

14) Nullable attribute rate in 50-80%
    Notice: Most of the time, attributes should be NOT NULL.
    Too high a rate of nullable attributes may reveal that some fields
    are lacking a NOT NULL.

15) Attribute count per table over 20
    Having many attributes in the same table may suggest the need
    for additional relations.

16) Non-integer Primary Key
    Having integer primary keys without specific application se-
    mantics make updates easier.

17) Table with a single attribute
    Possibly some more attributes are needed to have a semantic.

18) Tables with Composite PK
    A simple primary key, without specific semantics, is usually a
    better design, and references through foreign keys are simpler.

19) Nullable attribute rate in 20-50%
    Information: Most of the time, attributes should be NOT NULL.
    Too high a rate of nullable attributes may reveal that some fields
    are lacking a NOT NULL.

20) FK length mismatch
    A Foreign Key should have matching referencing and refer-
    enced type sizes.

21) FK type mismatch
    A Foreign Key should have matching referencing and refer-

22) Destination table and FK in different schemas
    A constraint and its destination table are usually in the same
    schema.

23) Source table and constraint in different schemas
    A constraint and its source table should be in the same schema.

24) Table and index in different schemas
    An index and its table should be in the same schema.

25) Tables linked but in different schemas
    Linked tables are usually in the same schema.

26) Backend engine inconsistency
    Different backends are used in the same database. It may be
    legitimate to do so if a particular feature of one backend is
    needed, for instance full text indexes.

27) Unsafe backend engine used in schema
    An unsafe backend (e.g. MyISAM) used at least once lacks
    referential integrity, transaction support, and is not crash safe.

28) MySQL is used
    MySQL lacks important features of the SQL standard, includ-
    ing missing set operators.

29) Unsafe backend engine used on table
    An unsafe backend (e.g. MyISAM) lacks referential integrity,
    transaction support, and is not crash safe.

30) Schema name length too short
    A schema name with less than 3 characters is really too short.

31) Table name length too short
    A table name with less than 2 characters is really too short.

32) Attribute name length too short
    An attribute name with less than 2 characters is really too short.

33) Homonymous heterogeneous attributes
    Better avoid using the same attribute name with different types
    on different tables in the same application, as it may confuse
    the developer.

34) Mixed table name styles
    Better use homogeneous table names.

35) Schema name length quite short
    A schema name with 4 characters is quite short.

36) Mixed attribute name styles
    Better use homogeneous attribute names.

37) Table name length quite short
    A table name with 3 characters is quite short.

38) Attribute name length quite short
    An attribute name of 2 characters is quite short.

39) Attribute named after its table
    An attribute contains the name of its table, which is redundant.

40) Schema name length short
    A schema name with 5 characters is short.

41) Table name length short
    A table name with 4 characters is short.

42) Attribute name length short
    An attribute name with 3 characters is short.

43) SuperUser with weak password
    User system error
    SuperUser with empty or username password.

44) Redundant FK
    Redundant Foreign Keys are costly to maintain.

45) Redundant indexes
    Redundant indexes are costly to maintain.

46) User with weak password
    User system warning
    User with empty or username password.

47) Table without index
    Not a single index on a table.