Database Design Debts through Examining Schema Evolution

Mashel Al-Barak
School of Computer Science
University of Birmingham UK, and King Saud University
KSA
mbarrak@ksu.edu.sa

Rami Bahsoon
School of Computer Science
University of Birmingham
Birmingham, UK
r.bahsoon@cs.bham.ac.uk

Abstract— **Causes of the database debt can stem from ill-conceptual, logical, and/or physical database design decisions, violations to key design databases principles, use of anti-patterns etc. In this paper, we explore the problem of relational database design debt and define the problem. We develop a taxonomy, which classifies various types of debts that can relate to conceptual, logical and physical design of a database. We define the concept of Database Design Debt, discuss their origin; causes and preventive mechanisms. We draw on MediaWiki case study and examine its database schema evolution to support our work. The contribution hopes to make database designers and application developers aware of these debts so they can minimize/avoid their consequences on a given system.**

Keywords— Technical debt; Database design debt; Database debt

I. INTRODUCTION

Information Systems(ISs) continually evolve in response to changes in users' requirements, environment and emergent needs for improved information provision and service qualities. The evolution can be, for example, driven by changes in technologies (e.g., moving to mobility), advances in communication channels (e.g., integrating new apps), business mergers leading to the unification of ISs, and the increasing interest in micro- level provisioning for services.

Databases are core for the functioning of these systems and most systems tend to adapt existing legacy databases, extending on their underlying design. As a result, evolving databases and their schemas through refactoring is a common practice; the exercise is often carried out to meet new requirements and to optimize emergent qualities.

Databases schema evolution has been studied extensively in the literature; tools have been invented to automate the evolution process and experiments were conducted to analyze the impacts of evolution [1], [2]. There are several studies on database design flaws, their causes and consequences [3]. Interestingly, studies have reveals situations, where non-optimal database design decisions and flaws can carry some benefits, which can be arguably viewed as a form of debt in situations where short-terms gains compromises long-term benefits. System developers and database designers need to follow optimal database design practices in the adaptation and refactoring process. However, the process can suffer from ill-design and immature decisions that can introduce what we term as database design debts, which we explore in this paper.

The novel contribution of this paper is: We define the concept of Database design debt and we explore its causes. We develop to a taxonomy, which classifies various types of debts that can relate to conceptual, logical and physical design of a database. We discuss their origin, causes and preventive mechanisms. We draw on case study to provide examples that relate to these debts. The contribution hopes to make database designers and application developers aware of these debts so they can minimize/avoid their consequences on the system.

II. APPROACH

The main objective of this research is to identify technical debts that relate to designing and evolving databases schemas. Despite that vast contributions of research on technical debt over the last few years, where attempts included requirements, architecture, code (e.g., [4], [5], [6], [7]),databases design debts have received little attention. Database design also captured the attention of many researchers [3], [8], [9]. However, these attempts were discussed in isolation of their economics and the consequences of databases design choices creating or minimizing debts. The contribution hopes to assist database designers to identify, track and consequently manage or avoid technical database design debts. After that, we can go further and manage these debts to improve the evolution of the schema on the long run.

Our approach for understanding the concept of database technical debt is by developing a taxonomy that defines and classifies different kinds of debts that can relate to relational databases.

Taxonomies can help researchers and practitioners understand and analyze the relationship between concepts in complex domains by classification and organizing knowledge [15]. Fig 1 represents the final taxonomy developed to classify debts related to relational databases. We developed the taxonomy following an iterative approach detailed in [16]. Due to brevity, we will only describe its application.

978-1-5090-3854-1/16 $31.00 © 2016 IEEE
DOI 10.1109/MTD.2016.9

Meta-characteristic: First, we chose *design phase* as our meta-characteristic.

Ending Conditions: We defined the following objective ending conditions: (1) At least one design debt is classified under every characteristic of every dimension, (2) All design principles and practices have been examined, (3) No new dimensions or characteristics were added in the last iteration, (4) Every debt is unique and not repeated. As for the subjective ending conditions, we adopted them from [16]: (5) concise, (6) robust, (7) comprehensive, (8) extendible, and (10) explanatory.

Approach: We followed the deductive approach (conceptual to empirical). Starting with a conceptual approach, we proposed the types of database design debts after reviewing the theoretical literature about database design processes and practices. Then, we moved to the empirical approach by examining a real world case study to see how they fit in the classification.

III. Definition for Database Design Debt

There is no general agreement on the definition of Technical debt (TD). Existing definitions have steered by various research perspectives and their pending problems for TD identification, tracking, and management etc. Examples of these perspectives includes, but not limited to, code-level debt, software architecture-level debt, maintenance, services, enterprise social ones etc. (e.g., [10], [11], [12]). The common ground of these definitions is that the debt can be attributed to poor and suboptimal engineering decisions that may carry immediate benefits, but are not well geared for long-term benefits. Our definition follows similar ethos, but with explicit focus on databases and schema evolution (i.e. the artifacts that are concerned with the debt).

Database design debt can be attributed to immature or suboptimal database design decisions that lags behind the optimal/desirable ones. These decisions can have negative impact on the schema, its maintenance and evolution. They can also ripple to negatively affect the beneficiary systems and their economic sustainability.

The causes of the database debt can stem from conceptual, logical, and/or physical database design decisions, violations to key design databases principles, use of anti-patterns etc. Analogous to the concept of "code smell" [13], some "smells" that can be found in a database [14]: tables with too many columns or rows, redundant data, "smart" columns and multi-purpose columns or tables are just among the examples. These "smells" are consequences for violating database design principles.

IV. Taxonomy for database Design debt

Database design is mainly composed of [17] :

A. Conceptual design phase

This design is normally conducted after requirements analysis phase. It is a high-level data model, which translates the requirements to a conceptual schema. It is usually done through modelling and abstractions (e.g., the use of the Entity-Relationship diagram and its variations); the modelling Conceptual design debt is not purely technical and can be attributed to social aspects in engineering the system. It can be accidently introduced by the designer due to the lack of expertise, novelty of the domain, inability to deal with abstractions, the lack of human-skills in gathering requirements etc.

B. Logical design phase

This is the first step for implementing the database. It is transforming the conceptual schema into the implementation schema or the "relational" database schema that will be used in the database system. It is basically mapping the entities and relationships to tables and columns. To complete this phase successfully, developers must follow appropriate design principles in order to reflect the real world and improve the data quality of the database. Technical debt can occur in this phase due to several reasons: lack of knowledge of suitable design principles, and ignorance of how to apply those principles to meet business objectives. The debt can be carried from the conceptual level.

C. Physical design phase

Logical design basically concerns of "What" to store, physical design, on the other hand, is concerned with "How" to store it. During this final phase of database design, developers specify files organization, indexes, storage and other physical features for the database. This phase is closely related to the performance of the database in terms of response time, space utilization and transaction throughput.

V. Logical Design

A. Normalization

Normalization process was first introduced by Codd in 1970 [18], as a process of testing a schema more than once to ensure that it satisfies a certain normal form. The main goal of normalization is to eliminate data redundancy. This can be achieved by ensuring that every non-key column in every table is directly dependent on the key, the whole key and nothing but the key, which is accomplished by decomposing a table into several tables. This principle has many advantages such as [17], [19]:

- Enforce data integrity as it reduces data redundancy.
- Eliminate updating and deleting anomalies and facilitate maintenance.

Poorly normalized database can carry a debt as it will result in a big amount of data duplication; it can put the entire burden on the applications code developer to modify the data. Moreover, from a business perspective, the cost of bad normalization will affect the quality of the stored data.

B. Principle of Orthogonal Design

While Normalization principle reduces data redundancy in the database, it is not guaranteed that a fully normalized database won't have redundant data [20]. This implies that normalization is not enough to eliminate the redundancy. Principle of orthogonal design completes normalization in that matter. Basically, orthogonality ensures that tables don't have overlapping meanings. To clarify this, C. J. Date presents the following example [20]: let's say that there is a table that holds a supplier information S and this table was decomposed to SNC {S_number, S_name, S_city} and STC {S_number, S_status, S_city}. This decomposition satisfies the 5th normal form. However, it is clear here that the city of the same supplier is stored in two tables, which implies that this design violates orthogonal design and will cause data redundancy. Another case where orthogonal principle may be violated is with horizontal decomposition of a relation. Such decomposition if not carefully implemented with proper constraint, data of a certain entity may appear in more than one table. Therefore, ignoring this principle can cause a debt on the system that will affect: i). Data correctness and ii). System performance as the false decomposition will increase the number of the tables query complexity.

C. Referential Integrity Constraints (RIC)

Referential integrity constraints are the main mechanism for ensuring data validity across many applications in relational databases [21], [22]. It is an essential principle in relational database that is implemented through "Foreign Key" from one table referencing primary key of another table. The foreign key of a tuple in a table, "child table", ensures that its value exists in the referenced table, "parent table", [19]. Referential integrity is violated when there is updating or inserting operations in the child table, where the value of the foreign key does not exist in the parent table. Another violation may occur if the value of a primary key in the parent table or foreign key in the child table is updated [17]. Referential integrity constraints have many advantages that include [21]:

- Ensures database data quality and consistency across the applications.
- Faster application development as it will reduce the programming code needed for data validity.

Regardless of those benefits, some designers do not declare foreign keys between tables in the database schema for two reasons:

- Make the database architecture simple so that updates will not be rejected, if they violate the constraints [23]. However, this issue can be resolved since there are options, other than rejection, to deal with RIC violations, such as CASCADE or SET NULL.
- Some applications developers, enforce referential integrity constraints in other places, other than the database such as the presentation tier through java scripts or the business tier [24]. However, since there are many ways for the data to be entered into the system concurrently, it is more reliable to push the enforcement down to the database [24].

Therefore, the absence of RIC at the database schema design can signal a potential a debt that must be managed carefully to avoid undesirable consequences.

D. Anti-patterns

We will look at flaws in relational databases that may harm the system and its data. The main database design principle is normalization. Normalization rules can eliminate redundancy, update anomalies and improve the overall design of the database. However, there are practices of database design that the principle does not address. Karwin in his book [23], provides guidelines for SQL database design as Anti-patterns, which are recurring solutions to certain problems; these solution may cause other problem in the future, It can be argued that the adoption of these solution can come with varying level of debts due to their sub optimality for the said solution. These anti-patterns arise from the need to store non-relational data, such as graph-like data, in a relational model, so developers may work around this problem using the following anti-patterns that can produce problems in the future. The following sections summarize some of the logical anti-patterns and explain their contribution to technical debt.

1) Referencing the same table (for hierarchical data): This design anti-pattern is called adjacency list, which is a common design among developers to store hierarchal data. However, this design has its downsides when it comes to queries, or delete from an adjacency list table. Querying only two levels of the tree is simple by issuing an outer join select statement. On the other hand, because a tree can extend to any depth, we need to query all the hierarchy with leaf nodes regardless of the number of levels. Additionally, to issue a query with an outer join that corresponds to each level of the tree, the number of outer joins in a SQL query must be fixed. Similarly, deleting a node from a tree requires multiple steps based on whether we want to delete an entire sub-tree or we want to delete a non-leaf node and promote its children to another part of the tree. Clearly, both operations require a lot of code, and there are alternative solutions to that, like path enumeration or nested sets.

2) Entity-attribute-value (EAV) tables: This is a table design where a table has 3 columns: one for the type of entity it is supposed to represent, another for a parameter or attribute or property of that entity and a third one for the actual value of that property. Clearly this design paradigm has several disadvantages:

- Stores massive data that is more difficult to manage than regular table design,
- Referential integrity constraints are complex to implement,

- Enforcing data integrity constraints is very complicated, i.e. data type and maximum allowable size of input data,
- Querying data from this table is very complex.

This structure can be valid for scheme-less data, where the numbers of attributes or the type of attributes are different in each row. However, it can be viewed as a form of debt that may harm the data integrity, system performance and may increase the coding complexity.

3) Polymorphic association (when one column references multiple tables): this design happens when a table references more than one parent table. In relational model a foreign key cannot reference more than a single table. To solve this, add additional column to hold metadata to reference parent tables (like tables names). This will lead to data consistency issues. Moreover, it will make queries more complex and harder to understand. There are better solutions for this such as: creating a super table for the types, create intersection tables for all the types or add several columns to reference each type.

VI. PHYSICAL DESIGN

The process of physically designing a database is the next step that follows determining the right decomposition of tables, based on the appropriate logical design principles. This process involves: partitioning, indexing and definition of views [25]. These decisions are important to optimize the performance of the database. Elmasri and Navathi [17] stated that developers need to first analyze the application, in terms of the transactions, queries, expected update operations frequencies..etc., to make the appropriate decisions for a better physical design. Similar to logical design, each decision can carry a debt that may affect the performance. In order to appropriately manage these debts, the first step is to make them explicit, and define what are the possible debts that can occur during database physical design.

A. Indexing:

One of the physical design decisions developer need to make relates to indexing, including: what attribute(s) to index, use clustered or non-clustered index, hash or dynamic hash index [25]. Generally, an attribute can be indexed if it is a key(unique) or the attribute is used by some query in a selection or a join condition. The main reason behind index creation is to quickly locate rows, searched based on the indexed attribute(s) [19]. In addition, unique indexes guarantee uniquely identifiable records in the database [17]. However, indexes allocate space on the disc. In addition, it will increase update and insert overhead, every time we use INSERT, UPDATE, or DELETE, the database has to update the index data structures for that table to be consistent so that subsequent searches use these indexes to find rows reliably [23]. It may have the opposite g and decrease the performance of the database. This implies that the creation of indexes without planning and knowledge of the queries and transactions and different types of indexes (clustered vs. non clustered indexes) can carry a debt that will harm the behavior of the system. Therefore, developers need to make a large number of trade-offs and understanding the implication of a specific index in the database.

B. De-normalization

As discussed in the previous section, normalization is important to minimize data redundancy, and thereby reduce update anomalies. In reality, databases are normalized to the third or at most the Boyce-Codd normal form. However, developers sometimes sacrifice this to optimize the performance of the database and increase the speed of the execution of queries and transactions [17]. That is why designers, depending on the nature of the system, choose to "de-normalize "their database to weaker normal forms to reduce the number of tables they have to access and retrieve the required data, which in turn will lead to minimize the number of joins needed for a query statement. As a result, designers have to decide on whether to carry a debt of de-normalization to improve the performance, and in turn, deal with redundancy and anomalies. Therefore, there is a need to develop a formal framework to manage this kind of debt and compare the cost and benefits to help developers with their decision instead of doing it in an ad hoc manner and depend on the developers' experience and intuition.

C. Views

Views, some call them stored queries, are the subset of tables whose contents are defined by a query. Views are not physically stored in the database, but it will be established when it is called by a query [26]. For this reason, the view does not take up any disk space for data storage, and it does not create any redundant copies of data that is already stored in the tables that it references. Developers create views for several reasons [22]: i). Security, as the user has permissions to access set of the data through views and not the original tables, ii). Query Simplicity: a view can retrieve data from several tables and present it as a single table, turning multi-table queries into single-table queries against the view. On the other hand, views may decrease the performance of the database, the DBMS still translate queries against the view into queries against the underlying source tables. If the view is defined by a complex, multi-table query then simple queries on the views may take considerable time [22]. This issue may be resolved with materializes views [25], where the views and its contents are created in advance and stored in the database. This option is more reliable if the developers are willing to trade increased disk space use for faster response. It is important to mention that materialized views need be maintained carefully to be up to date with the base tables. Therefore, developers have to consider all these aspects prior the decision of creating views as it carry a debt that well affect the system performance and maintenance.

D. Partitioning

The basic idea behind partitioning is to divide a single table into "little tables or partitions" each holding specific range of data [25]. These partitions may reside on different nodes or files. This can be very beneficial for supporting very large tables as it has many advantages that include [25]:

- Improve query performance and have a faster response by accessing partitions of the table instead of the whole table,
- Reduce administration overhead by allowing administrators to work on subsets of a table as opposed to the whole table.
- Possibility for databases to store larger tables.

On the other hand, partitioning can cause a debt and degrade the system performance, if it is done without planning. Designers must acquire knowledge of the most frequent and complex queries that this issued on the database. Partitioning strategy should be based on the frequent columns that are used in complex queries. For example, if a table was partitioned horizontally by a specific column, and most queries were to search for data by another column, this will cause the search to be on all the partitions across the table, which in turn will affect the performance negatively. Similarly, in the case of vertical partitioning, when tables needed to be joined and the join columns resides on different partition. This process is very expensive and the debt will harm the system eventually.

E. Lookup tables

Lookup table, or as some may call it as code table, is a method for restricting values for a column [23]. It has benefits that include [22]:

- Ability to provide detailed description for the column values,
- Ensuring data integrity,
- Flexible updating and addition of values.

On the other hand, when it comes to performance, having multiple small lookup tables will take space as opposed to fewer tables. Moreover, lookup tables needed to be referenced by foreign keys and foreign key constraints reduce performance whenever a row in the sourced tuple is altered [22]. Some designers consider having one huge lookup table for all the tables. This will have a negative impact on the performance as it will create "hot spot" on the disk that will result in delayed response time [27]. Therefore, designers need to be careful when considering using lookup tables as it can create a debt on the system performance.

F. Hardware and Software choices

Debt related to physical design can also relate to poor decisions regarding deployment, the choice of physical storage, technolgy choices, subsequent maintainance and support by the storage provider etc. Physical design concerns the performance efficiency of the database from the user perspective. This performance can also be affected by developers' choices of different relational database management systems, such as: Oracle, MySQL, PostgreSQL etc. Each of these systems provides different capabilities but also has some constraints and limits. Choosing a specific system can carry a debt due to these limits that can affect performance.

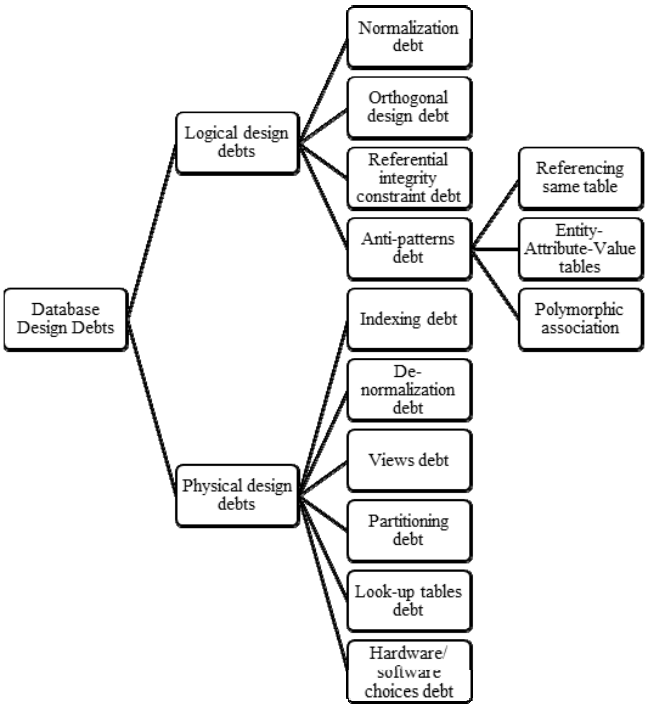

Fig 1 : Taxonomy of relational database design debts

VII. CASE STUDY

To better understand the concept of databases design debt, the following case study demonstrates some examples of logical design violations that may harm the system and may contribute to debts. Drawing on a real case study, we consider the MediaWiki web application [30], this application was first introduced in 2001 and has 26 schema versions that evolved throughout the years. MediaWiki database schema file is stored in a SQL file that was extracted from a Git repository [31]. Information about the database layout, tables and columns were obtained from [32]. We reviewed 26 releases and sifted along the changes of these releases. We manually compared successive versions of the database schema and extract all the schema changes in each commit on that file. Some of the logical design violations were as follows:

- *Referential integrity constraint debt:* We noticed that from release 1 to 4 of the database schema an absence of foreign keys because the database was stored in MYISAM engine, which did not support referential integrity constrains. This is an example of a logical design debt, considering the importance of

implementing foreign keys in the database to ensure data consistency and eliminate anomalies. However, since release 5 they moved to InnoDB engine that supports referential integrity constraints, but there was no implementation of foreign keys on the schema file. Moreover, the decision to use MYISAM or InnoDB engines could relate to physical design debt. However, since physical debts should be identified after a proper and complete analysis of the application, we cannot determine that this choice is considered as a debt.

- *Normalization debt*: This principle was violated in several releases:
    - In the 5th release of the application, table pagelinks was added to merge two separate tables. The new table was created without a primary key. Although it does have a unique key made up from all the three columns together, unique keys can accept a null value and cannot be referenced.
    - A clear violation of the normalization principle was identified in the 6th release of the schema. The violation resembles in the addition of two new columns, ipb_range_start and ipb_range_end, to ipblocks table. Since ipb_range_end depends on the ipb_range_start, those columns should be moved to a separate table to satisfy second normal form.
    - In the 10th release, a new column, rc_params, was added to recent_changes table. This column will store parameters separated by a new line. Storing multiple values in one column violates the first normal form and can cost the application when querying for a specific parameter.
- *Referencing the same table debt:* This debt was observed in the 10th release of the system. A new column, rev_parent_id , was added to revision table to reference rev_id in the same table creating a tree structure. As discussed in a previous section, the tree can extend in depth depending on the numbers of revisions, which will increase the complexity of querying specific revision at a specific level of the tree.
- *Orthogonal design debt*: In the 18th release, table user_former_groups was added to the schema to store groups that the user has once belonged to. The user may again belong to these groups. The new table is a clone of the User_groups table. Since some of the user current groups may be stored in both tables, which indicates an overlapping between those tables, it will create redundancy. Moreover, developers will have to make more effort to keep them both synchronized.

VIII. Quality Attributes affected by database design debts

Quality, as defined by [28], is "the degree to which a specific product conforms to a design or specification". This definition emphasizes the impact of the internal quality of a system development. **Table 1** highlights ISO quality attributes [29] that can be potentially affected by databases design debts. Those attributes may be affected in a positive or a negative way. This is to be determined depending on the nature of the system and after proper analysis of its characteristics and transactions.

Table 1: QUALITY ATTRIBUTES AFFECTED BY DATABASE DESIGN

Design Principle	Quality attributes
Normalization	1. Correctness 2. Maintainability 3.Performance efficiency
Orthogonality	1. Correctness 2. Maintainability 3.Performance efficiency
Referential integrity constrains	1. Correctness 2. User Error Protection 3. Integrity 4. Modifiability 5. Performance efficiency
Referencing the same table	1. Completness 2. Modifiability 3. Performance efficiency
Entity-attribute-value tables	1. Modifiability 2. Integrity 3. User Error Protection 4. Operability 5. Adaptability 6. Performance efficiency 7. capacity
Polymorphic association	1. Correctness 2. Performance efficiency
Indexes	1. Performance efficiency 2. Maintainability
De-normalization	1. Correctness 2. Maintainability 3.Performance efficiency
Partitioning	1. Performance efficiency 2. Reliability 3. Security
Views	1.Performance 2. Security
Lookup tables	1.Correctness 2. Integrity 3. User Error Protection 4. Performance efficiency
Hardware and software choices	1. Performance effeciency 2. Compatibilty 3. Maintainability 4. Scalability 5. Reliability

IX. Related Work

Technical debt has been the focus of many researchers in the software engineering community [7]. Although, it was studied from different perspectives, technical debt that are related to databases and their design did not receive significant attention. Up to our knowledge, the only work that is close to our research was written by Weber and others in

[24]. The authors focused on missing foreign keys in relational databases as a technical debt. They also proposed a measurement for that debt and summarized a process to reduce it. Since our focus is on database design debts from the evolution of legacy databases, the examined debts can be related to both database design and database reengineering. However, researches on those areas were not focusing on how to manage design flaws, that carries a debt on the system, as those flaws may not harm the application and the cost to fix it may outweigh their effects.

X. Conclusion and Future Work

Schema evolution is an inevitable task in today's world of continues changing requirements and technology. As this task involves uncertainty of the cost of the change, it can carry a debt related to the ill-design of the schema. Moreover, the debt can ripple to the data it holds and to the applications on top of the database. Researchers over the years explored technical debt from many perspectives. However, technical debts that are related to databases schema have not yet been explored. In this paper we explored technical debts that occurred in database design from schema evolution. The first step to accomplish this, we presented the tasks and principals involved in the design process and discussed their contribution to debts existence. We have noticed that even if logical design principles are fulfilled successfully, it can carry a debt that can be reflected in performance efficiency and other qualities of the system. This can be one of the reasons behind developers "Intentionally" carrying a positive debt to capture some benefits that will outweigh the debt on the system. For example, developers may choose to de-normalize their tables to weaker normal forms; this will consequently increase the data retrieval speed and efficiency. Indeed, setting specific criteria for identifying technical debts is scenario and context driven. The identification is also driven by stakeholders' quality attributes priorities and demands. As a future work, we will elaborate on the developed taxonomy to consider how we can quantify the principal and interest of the debts.

References

[1] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the database schema evolution process," The VLDB Journal, vol. 22, no. 1, pp. 73–98, Feb. 2013.

[2] L. Meurice and A. Cleve, "Dahlia: A visual analyzer of database schema evolution," in Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on, 2014, pp. 464–468.

[3] M. J. Hernandez, Database design for mere mortals: a hands-on guide to relational database design. Pearson Education, 2013.

[4] N. A. Ernst, "On the role of requirements in understanding and managing technical debt," in Proceedings of the Third International Workshop on Managing Technical Debt, 2012, pp. 61–64.

[5] Z. Li, P. Liang, and P. Avgeriou, "Architectural Debt Management in Value-Oriented Architecting," in Economics-Driven Software Architecture, Elsevier, 2014, pp. 183–204.

[6] N. Zazworka, C. Seaman, and F. Shull, "Prioritizing design debt investment opportunities," in Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 39–42.

[7] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," Journal of Systems and Software, vol. 101, pp. 193–220, Mar. 2015.

[8] J. L. Harrington, Relational Database Design and Implementation. Morgan Kaufmann, 2016.

[9] A. Badia and D. Lemire, "A call to arms: revisiting database design," ACM SIGMOD Record, vol. 40, no. 3, pp. 61–69, 2011.

[10] T. Theodoropoulos, M. Hofberg, and D. Kern, "Technical debt from the stakeholder perspective," in Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 43–46.

[11] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An enterprise perspective on technical debt," in Proceedings of the 2nd Workshop on managing technical debt, 2011, pp. 35–38.

[12] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in Proceedings of the 2nd Workshop on Managing Technical Debt, 2011, pp. 1–8.

[13] M. Fowler, K. Beck, J. Brant, and W. Opdyke, "Refactoring: Improving the Design of Existing Code," 2013.

[14] S. Ambler, Agile database techniques: Effective strategies for the agile software developer. John Wiley & Sons, 2012.

[15] R. L. Glass and I. Vessey, "Contemporary application-domain taxonomies," IEEE Software, vol. 12, no. 4, pp. 63–76, Jul. 1995.

[16] R. C. Nickerson, U. Varshney, and J. Muntermann, "A method for taxonomy development and its application in information systems," European Journal of Information Systems, vol. 22, no. 3, pp. 336–359, 2013.

[17] R. Elmasri and S. B. Navathe, "Database systems: models, languages, design, and application programming," 2011.

[18] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Commun. ACM, vol. 13, no. 6, pp. 377–387, Jun. 1970.

[19] A. Silberschatz, H. F. Korth, S. Sudarshan, and others, Database system concepts, vol. 4. McGraw-Hill New York, 1997.

[20] C. Date, Database Design and Relational Theory: Normal Forms and All That Jazz. O'Reilly Media, Inc., 2012.

[21] M. Blaha, "Referential Integrity Is Important for Databases," Modelsoft Consulting Corp., 2005.

[22] S. W. Ambler and P. J. Sadalage, Refactoring databases: Evolutionary database design. Pearson Education, 2006.

[23] B. Karwin, SQL Anti-patterns, Avoiding the Pitfalls of Database Programming. The Pragmatic Bookshelf, 2010.

[24] J. H. Weber, A. Cleve, L. Meurice, and F. J. B. Ruiz, "Managing Technical Debt in Database Schemas of Critical Software," 2014, pp. 43–46.

[25] S. S. Lightstone, T. J. Teorey, and T. Nadeau, Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more. Morgan Kaufmann, 2010.

[26] J. D. Ullman and J. Widom, Database Systems: The Complete Book. Upper Saddle River, 2000.

[27] M. A. Poolet, "Designing for Performance: Lookup Tables." [Online]. Available: http://sqlmag.com/database-administration/designing-performance-lookup-tables. [Accessed: 11-Jun-2016].

[28] H. L. Gilmore, "Product conformance cost," Quality progress, vol. 7, no. 5, pp. 16–19, 1974.

[29] "ISO/IEC 25010:2011(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models." [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en. [Accessed: 10-Jun-2016].

[30] "MediaWiki." [Online]. Available: https://www.mediawiki.org/wiki/MediaWiki. [Accessed: 25-Jun-2016].

[31] "wikimedia/mediawiki," GitHub. [Online]. Available: https://github.com/wikimedia/mediawiki. [Accessed: 25-Jun-2016].

[32] "Manual:Database layout - MediaWiki." [Online]. Available: https://www.mediawiki.org/wiki/Manual:Database_layout. [Accessed: 25-Jun-2016].