Assessing the Occurrence of Blocking Operations in Database Schema Evolution: A Case Study

Willie Lawrence
IMD - UFRN
Natal, Brazil
willie.silva.016@ufrn.edu.br

Eiji Adachi
IMD - UFRN
Natal, Brazil
eijiadachi@imd.ufrn.br

ABSTRACT

During the evolution of a database schema, some schema-changing operations (e.g., the "ALTER TABLE" command) require the underlying database management system to lock tables until the operation is finished. We call these schema-changing operations *blocking operations*. During the execution of blocking operations, a software application may behave abnormally, varying from a slow page loading to an error caused by a web request taking too long to return. Despite their potential negative impact on important quality attributes, blocking operations have not yet been empirically investigated in the context of software evolution. To fill this gap, we conducted a large industrial case study in the context of a Brazilian software company. We analyzed 1,499 atomic schema-changing operations from a period of 6 years to explore which blocking operations the developers frequently performed during the evolution of the database schema of a target system. The intention behind this case study is better understanding the problem in its original context to outline strategies to correct or mitigate it in the future. Our results show that blocking operations were very common, though not all of them seemed to cause observable downtime periods. We also present some mitigating strategies already in use by the development team of the target system to cope with blocking operation during software evolution, avoiding their negative impact.

KEYWORDS

blocking operations, database schema, schema evolution, software evolution, case study

1 INTRODUCTION

Software evolution is a natural phenomenon in any minimally long-lived software system [10]. When new features, bug fixes, or perfections to existing features are implemented in a system, both the source code and the database schema, which is the blueprint of the logical organization of data in a database, need to be modified to meet these demands [8]. During the modification of the database schema, some operations, such as an "ALTER TABLE" command, require the underlying database management system to lock the table until the operation is executed by the database, blocking reads or writes to that table [6]. In this paper, we call schema-changing operations that block other database clients to perform read/write operations concurrently as *blocking operations*.

Most of the time, web applications are reading or writing data into the database. In this sense, if the database cannot perform a read/write operation until the end of a schema-changing operation, the web application is prevented from running seamlessly. To the final user of a software application, the impact of blocking operations can vary from a slow page loading to an error caused by a web request taking too long to return. Thus, blocking operations have the potential to hinder software quality attributes, like performance, usability, or availability [9].

Despite their potential negative impact on important quality attributes, blocking operations have not yet been empirically investigated in the context of software evolution. So far, empirical studies have investigated the evolution of the database schema alone [2, 3, 12] or the joint evolution of the database and application source code [7, 8]. But none of these works have investigated the occurrence of blocking operations during database schema evolution.

To fill this gap, we conducted a large industrial case study in the context of a Brazilian software company to explore which blocking operations the developers performed during the evolution of the database schema of a target system. The intention behind this case study is better understanding the problem in its original context to outline strategies to correct or mitigate it in the future.

Our case study analyzed 1,499 atomic schema-changing operations during an evolution period of 6 years, comprising the period between January-2015 and March-2021, of one target system implemented in Python using the Django Web framework. We observed a total of 11 different types of atomic schema-changing operations, from which 7 are blocking operations. We also observed that there existed at least one blocking operation in 87.5% of the analyzed months, so blocking operations were very common in our case study. On the other hand, we observed that not all blocking operations seemed to cause downtime periods and very few months (5.5%) had blocking operations potentially causing downtime periods. We identified some mitigating strategies already in use by the development team to cope with blocking operations during software evolution, avoiding their negative impact.

The main contributions of this paper are: (i) We present a large case study assessing the occurrence of blocking operations during database schema evolution; (ii) We observed that blocking operations with visible downtime periods are those that need to scan all records in a table; and (iii) We found that some short-duration blocking operations alone do not cause visible downtime periods, but when the database executes them sequentially, they might cause visible downtime periods.

2 CASE STUDY DESIGN

We performed this case study in the context of the ANON company, a small-sized company based in Brazil, working on the development of management systems in the accounting area.[1] In recent years, large clients acquired the company's main product, increasing the demand for better levels in service quality attributes, especially software availability. Even with the adoption of continuous integration and continuous delivery practices, the system update in the production environment has become a critical factor affecting software availability. In some scenarios, service interruptions are observed during a system update, especially when the database performed blocking operations. We detail the settings of our case study in the next sections, following the guidelines provided by Runeson et al. [11].

2.1 Goal and Research Questions

Database schema evolutions are necessary and may eventually require the execution of blocking operations. In the ANON company, the developers observed problems caused by blocking operations, such as slow pages loading or errors caused by web requests taking too long to return, in some scenarios of database schema evolutions. However, there is still limited empirical knowledge regarding the occurrence of database blocking operations during database schema evolution.

The goal of our case study is to explore which blocking operations the database frequently performed during the evolution of the database schema of our target system. To achieve our research goal, we investigate the following research questions:

RQ1: Which atomic operations were performed on the database schema along with its evolution?
RQ2: How often were blocking operations performed along with the database schema evolution?

The first question is more exploratory and serves the purpose of characterizing how the database schema has evolved, helping us to characterize which different atomic operations, including both blocking and non-blocking operations, the database frequently performed during the period analyzed in our study. The second question is more analytical and complements the first one by conducting a more specific analysis regarding the blocking nature of atomic operations observed during the evolution period analyzed.

2.2 The Case and The Units of Analysis

The ANON company organizes its work in software products, with the YMS system being its main product marketed to client companies.[2] Therefore, we choose the YMS as the case of our study. The YMS system is a web-based application written in Python using the Django Web Framework and storing its data in a PostgreSQL 9.6 database. The YMS system has been in development since 2015 and, its current version, in March-2021, has 123 database tables and 999 columns. Moreover, the units of analysis of our investigation are the atomic schema-changing operations performed in the database schema of the YMS system over the period from January-2015 to December-2020.

[1]For the sake of anonymity, we will use the alias "ANON" to refer to the company.
[2]For the sake of anonymity, we will use the alias "YMS" to refer to the company's main product.

2.3 Concept Operationalization

Typically, in Django-based web application development, developers do not explicitly implement Structured Query Language (SQL) nor Data Description Language (DDL) commands. Instead, they rely on the framework Object-Relational Mapping (ORM) mechanism to automatically map the application's entities to the correspondent database tables. Django's ORM mechanism also provides the concept of "Migrations" to manage the evolution of model entities. In simple terms, when developers change the application's model entities mapped by the ORM feature, Django's Migration system identifies what has changed and stores these changes to a new "Migration File", which is a Python script listing schema-changing operations.

This way, we recovered the atomic schema-changing operations performed during YMS evolution, which are our units of analysis, by analyzing a sequence of "Migration Files". We analyzed the extracted migration files using the `sqlmigrate` tool, provided by the Django framework itself.[3] This tool processes a given migration file and produces the exact DDL commands intended to be executed in the database schema.

3 DATA ANALYSIS

We collected 248 migration files created in the period between January-2015 and March-2021. From these 248 migration files, we extracted a total of 1,499 atomic operations. Then, we categorized each operation by reusing the list of categories of atomic operations proposed by Qiu et al. [8]. The tablerefnewqiucodes depicts the list of atomic operations and their respective category.

We extended the original list proposed by Qiu et al. by adding two new operations (*A16 - Add Constraint* and *A17 - Drop Constraint*) since we observed this type of operation in our study. We also extended it by categorizing each atomic operation by its blocking/non-blocking nature; the Table 1 presents this categorization in the column "Blocking Operation". It is worth mentioning that the blocking nature of each atomic operation refers to the behavior of the operations as implemented by the version of the PostgreSQL used in the application used as our case. Finally, we designed and implemented a tool to capture and parse the output of "sqlmigrate" to categorize each DDL command according to the codes presented in Table 1.[4] Next, we detail our analysis that answer our research questions.

3.1 RQ1: Which atomic operations were performed on the database schema along with its evolution?

To answer our first research question, we analyzed the distribution of the 1,499 operations according to their category, as listed in Table1. The chart in Figure 1 shows that distribution.

Of the total 26 operations types listed in Table1, we observed 11 operations in the data collected from the migration files. From these 11 operations, 7 are blocking operations. We observed that approximately 90% of the atomic operations analyzed in our case study are concentrated in 7 different categories: *A18 - Add Index*, *A2 - Add Column*, *A12 - Add Foreign Key*, *A21 - Drop Column Default*

[3]The documentation of the `sqlmigrate` tool is available in https://docs.djangoproject.com/en/3.2/ref/django-admin
[4]Scripts and datasets used in this work available in https://gitlab.com/cptx032/mde

Table 1: Atomic Operations for Database Schema Evolution. Source: adapted and extended from Qiu et al. [8]

Ref.	Atomic Operation Category	DDL Postgres	Blocking Operation
A1	Add Table	CREATE TABLE t_name;	No
A2	Add Column	ALTER TABLE t_name ADD c_name;	Yes
A3	Add View	CREATE VIEW v_name AS ...	No
A4	Drop Table	DROP TABLE t_name	No
A5	Rename Table	ALTER TABLE o_t_name RENAME n_t_name	No
A6	Drop Column	ALTER TABLE t_name DROP COLUMN c_name	No
A7	Rename Column	ALTER TABLE t_name CHANGE COLUMN o_c_name n_c_name	No
A8	Change Column Datatype	ALTER TABLE t_name MODIFY COLUMN c_namec_def	Yes
A9	Drop View	DROP VIEW v_name	No
A10	Add Key	ALTER TABLE t_name ADD KEY k_name	Yes
A11	Drop Key	ALTER TABLE t_name DROP KEY k_name	No
A12	Add Foreign Key	ALTER TABLE t_name ADD FOREIGN KEY fk_name ...	Yes
A13	Drop Foreign Key	ALTER TABLE t_name DROP FOREIGN KEY fk_name	No
A14	Add Trigger	CREATE TRIGGER trig_name ... ON TABLE t_name ...	No
A15	Drop Trigger	DROP TRIGGER trig_name	No
A16	Add Constraint	ALTER TABLE t_name ADD CONSTRAINT c_name ...	Yes
A17	Drop Constraint	ALTER TABLE t_name ADD CONSTRAINT c_name ...	No
A18	Add Index	ALTER TABLE t_name ADD INDEX idx_name	Yes
A19	Drop Index	ALTER TABLE t_name DROP INDEX idx_name	No
A20	Add Column Default Value	ALTER TABLE t_name MODIFY COLUMN c_name SET DEFAULT value	No
A21	Drop Column Default Value	ALTER TABLE t_name MODIFY COLUMN c_name DROP DEFAULT	No
A22	Change Column Default Value	ALTER TABLE t_name MODIFY COLUMN c_name SET DEFAULT value	No
A23	Make Column Not NULL	ALTER TABLE t_name MODIFY COLUMN c_name NoT NULL	Yes
A24	Drop Column Not NULL	ALTER TABLE t_name MODIFY COLUMN c_name NULL	No
A25	Add Stored Procedure	CREATE PROCEDURE pro_name ...	No
A26	Drop Stored Procedure	DROP PROCEDURE pro_name	No

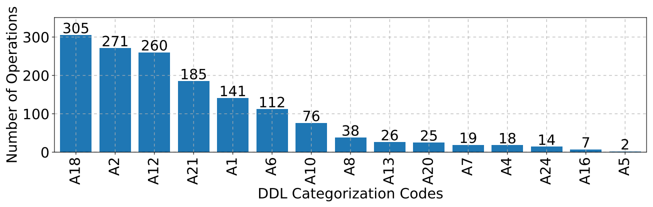

Figure 1: The most common operations found in the "Migration Files".

Value, *A1 - Add Table*, *A6 - Drop Column* and *A10 - Add Key* where 4 are blocking operations: *A18 - Add Index*, *A2 - Add Column*, *A12 - Add Foreign Key* and *A10 - Add Key*.

As the YMS evolves, the creation of new database entities is expected, in this sense, the developers create new tables and columns regularly. This regular database entity creation is expressed in the number of new columns (*A2 - Add Column*). Since Django ORM creates a separated atomic operation to set foreign and primary keys in the newly created columns, we understand why the high number of Add Column is accompanied by a high number of *A12 - Add Foreign Key* and *A10 - Add Key* atomic operations.

The presence of *A18 - Add Index* and *A21 - Drop Column Default Value* operations are less intuitive. Examining the situations where the *A21 - Drop Column Default Value* operations appeared we noted that, curiously enough, when the developer sets a default value to a column, the Django ORM creates an atomic operation defining the default value in the database but, right after that, removes it. According to the Django documentation, this behavior aims to fill the `NULL` values present in the column, but because of the design choice of Django ORM, the default value must not remain at the database level.[5] Lastly, we found that the high number of A18 - Add Index operations are the sum of all regular columns that the developer has marked to have an index and the indexes that the Django ORM creates automatically in the creation of primary and foreign keys.

Atomic operations related to the management of triggers, views, and stored procedures are not present in our extracted data. This is because the Django ORM framework does not support this kind of database entity through its ORM. Although possible to use these database entities in Django, the management of these entities is in charge of the programmer and not of the Django ORM. The lack of built-in support for these entities in the framework ORM discourages their use in the development of Django-based applications.

The operations *A11 - Drop Key*, *A17 - Drop Constraint*, and *A19 - Drop Index* are not present in the DDL generated by the migration files because most of the indexes, keys, and constraints are generated by Django automatically when creating foreign and primary keys, and they cannot be removed directly by the programmer using the ORM. Lastly, the operations *A22 - Change Column Default Value* and *A23 - Make Column Not Null* were absent for no specific reason; these column settings in the YMS database just happened to not change over time in the period studied.

Results 1) More than 70% of all atomic operations in the YMS application are database entity additions. 2) The YMS application often uses primary and foreign keys and, because of that, exists a high number of indexes in the database of the YMS application. 3) Influenced by the lack of support for some database entities in the Django framework, triggers, views, and stored procedures are not present in the analyzed "Migration Files".

[5] https://docs.djangoproject.com/en/3.2/ref/models/fields/#default

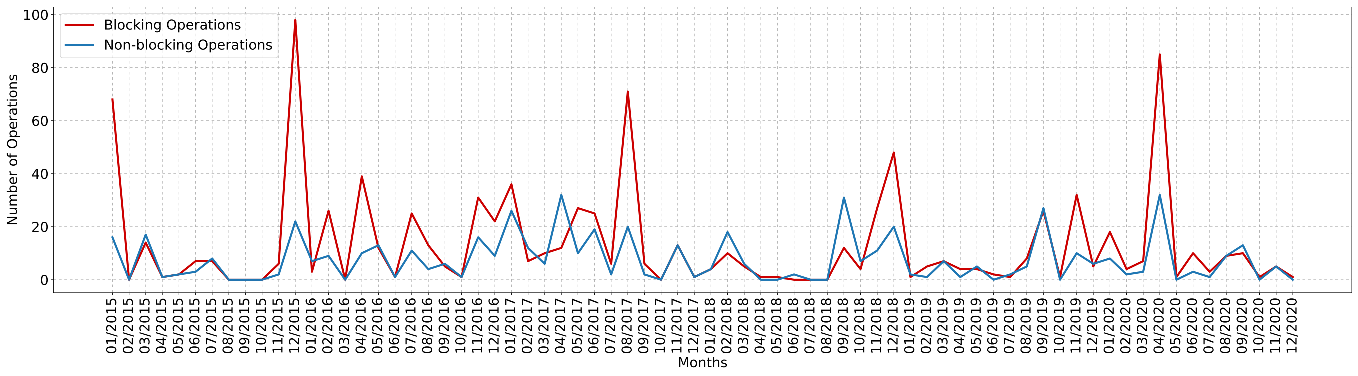

Figure 2: Blocking Operations Over Months.

3.2 RQ2: How often were blocking operations performed along with the database schema evolution?

As we mentioned before, we classified the DDL operations by their blocking nature. To answer our second research question, we used our tool to extract data about the number of operations over time, categorizing the operations in terms of their blocking/non-blocking nature. Figure 2 shows how both the aggregated number of blocking and non-blocking operations occurred along with the database schema evolution.

We noted that of the 72 months shown in Figure 2, 8 months (11.1%) do not contain any operation and 63 months (87.5%) contained at least one blocking operation. We also noted that in 10 months (13.9%) the number of blocking and non-blocking operations were equal, in 36 months (50%) the number of blocking operations exceeds the number of non-blocking operations, and in 18 months (25%) the number of non-blocking operations exceeds the number of blocking operations.

It is worth mentioning that a high number of blocking operations does not necessarily mean that a visible downtime will occur. Further investigations are necessary to better characterize which blocking operations, or which combinations of blocking operations, result in significant downtime periods. We observed that long downtimes are associated with operations that need to scan all the records in a table. Also, many sequential short-duration blocking operations can increase the chance of a perceptible downtime period because the Django migration system performs the DDL operations sequentially, and being so, the total time taken to perform them is the sum of the time of each one. In the ANON company, we noted that, in some production environments, the PostgreSQL database takes 18.5 milliseconds to perform short duration database operations, whose duration does not depends on the number of records in a table. Considering that 1 second of database downtime is perceptible, we can conclude that 55 sequential operations are enough to cause a visible blocking in the data access of the YMS application. We noted that 4 months had more than 55 blocking operations: January-2015 with 68 operations, December-2015 with 98 operations, August-2017 with 71 operations, and April-2020 with 85 operations. Since the company does not hold logs registering how the system behaved during these events of database schema update, we could not confirm if these batches of sequential blocking operations indeed caused perceptible downtime periods.

Results 1) There were blocking operations in 87.5% of the analyzed months. 2) Few months (5.5%) had enough sequential blocking operations to cause downtime just by performing them sequentially.

4 DISCUSSION

We noted that a visible blocking scenario happens in two situations. The first type of situation is when many database operations take place one after another. As mentioned before, the time of each operation is summed once the Django performs them sequentially. The time needed to perform a single short-duration operation depends only on the processing power of the server running the YMS application. Due to the rarity of this kind of event and the low duration of it (the worst case we found, 98 sequential operations, may have led to a duration of 1.8 seconds), the ANON company can accept them without problems.

The second situation is when a single database operation scans all the records (i.e., a full table scan) in a table with many records. The duration of that operation depends mainly on the number of records in the table. An example of that operation is the inclusion of primary keys constraints in existing columns, so the PostgreSQL will need to perform a full scan in the table to verify if all records met the primary key criteria. Initial results of preliminary experiments in the production environment of the YMS application show that a simple `ADD COLUMN` operation performed in a table with 1 million records can take up to 11 seconds to complete, on average.

Over time, the practitioners in the ANON company developed many strategies to avoid full table scans in PostgreSQL. One technique is to scan the table concurrently, i.e., without preventing other operations in that table to be performed. A example is the creation of a new column with a default value (e.g., `ALTER TABLE X ADD COLUMN Y INTEGER DEFAULT 0;`). PostgreSQL will scan all records in the table to fill the default values in the already existing records. To avoid that full table scan, the developers add a column in the table without a default value and, after that, they do other operations updating chunks of records with the default value. This method works because the first operation has an aggressive lock, i.e., completely prevents the table access, but has a very short duration, and the following operations have soft locks, i.e., they do not prevent some table accesses.

These techniques for coping with blocking operations require a degree of control over the database operations and a detailed analyzes of the blocking operation in question. On this, we found that the developer does not control the database directly if he is using the Django ORM. The control made by the Django Framework over the database is suitable to guarantee compatibility over multiple database systems, making it easy to switch between different vendors, for example. On other hand, this excessive control makes inflexible the operations that Django will perform and consequently, which database locks the PostgreSQL will create.

5 RELATED WORK

The literature has works on database schema evolution and works on techniques to overcome the database downtimes. The first type of work has many representatives [2, 3, 7, 8, 12]. Recent software technologies have better approaches to store database schema modifications, like the database schema version control applications, but legacy software has database schema definitions mixed with application code which makes it difficult to recover the database schema. We believe that this difficulty explains why these works have a high focus on static code analysis and database schema extraction from source code. As the YMS application uses the Django schema version control, the extraction step in our analysis was relatively easy. In this sense, our interest in these works was the database schema analysis process. Qiu et al. work [8] work gave us a way to categorize the atomic operations and Cleve et al. work [2] gave us another view of database schema evolution looking aspects like table lifetimes. Their analysis includes details that are unique to each application, making it hard to reproduce and generalize the same experiment in other software systems. Additionally, these works do not focus on the impacts of the schema evolution in the database availability when deploying blocking operations.

The second type of work is on deployment techniques developed to avoid outages in software in the deployment time. In these papers, the database is one of many different pieces of software that can make difficult the seamless operation of the application when deploying software. On this topic, De Jong [4] reviews many techniques used to make possible deploys without outages, proposing another technique to achieve the desired "zero-downtime" SQL database. The paper retrieved controlled results from experiments made by the authors, writing random data to some tables to reproduce long-downtimes, aiming to measure the efficiency of the new technique proposed. The work of De Jong et al. [5] examines the problems that the database schema evolution brings to continuous deployment techniques. In that paper, the authors found two main problems introduced by the databases in a continuous deployment scenario: 1) the need for a backward-compatible database schema evolution and 2) the long blocking operations made in schema evolution. The method used by this last paper to avoid long downtimes in the database schema evolution is made in general, without taking into account the particularities of each table, just replicating the schema and data in what they named as "ghost tables". The operation of data copying can be expensive if the tables have lots of data [1]. Additionally, we base our work on the analysis of a real database schema with real database data, which makes it possible to analyze which parts of the schema can impact an outage in a real-world application context.

6 CONCLUSION AND FUTURE WORK

In this paper, we report our analysis on database schema evolution using a real-world case of industry, focusing on the blocking operations performed in the application. With the available data, we identified sources of possible database downtime over time. With the identification of which types of blocking operation occur and in which frequency will be possible to choose which technique to use to perform database schema evolution techniques. Our work shows that the Django ORM helps developers by abstracting the burden of writing SQL statements, but it also hides the operations being made under the covers, which can lead to negative consequences, such as downtime periods caused by blocking operations. Future research may be interested in data surrounding the deployment stage as deployment duration, running application interruptions, outage duration, and statistics on which tables the user are interacting frequently. Future work may apply similar experiments to other Django projects in applications with databases available for analysis. A study on existing and new tools to gather more information about the deployment and to help developers to understand the impacts of a database change over deployment outages duration, focusing on more complex and real cases, should be an interesting focal point for future works.

REFERENCES

[1] David M. Alpern, Alan Choi, Chandrasekharan Iyer, Jaebock Lee, Kumar Rajamani, Shrikanth Shankar, Guhan Viswanathan, William Waddington, and Philip Yam. 2013. LOW-DOWNTIME AND ZERO-DOWNTIME UPGRADES OF DATABASE-CENTRIC APPLICATIONS. https://patentimages.storage.googleapis.com/29/27/a1/9559d0fe45d6a8/US8521706.pdf

[2] Anthony Cleve, Maxime Gobert, Loup Meurice, Jerome Maes, and Jens Weber. 2015. Understanding database schema evolution: A case study. *Science of Computer Programming* 97 (2015), 113–121.

[3] J. Delplanque, A. Etien, N. Anquetil, and O. Auverlot. 2018. Relational Database Schema Evolution: An Industrial Case Study. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 635–644. https://doi.org/10.1109/ICSME.2018.00073

[4] Michael De Jong. 2017. Zero-downtime SQL database schema evolution for continuous deployment. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017* (2017), 143–152.

[5] Michael De Jong and Arie Van Deursen. 2015. Continuous Deployment and Schema Evolution in SQL Databases. In *3rd IEEE/ACM International Workshop on Release Engineering (RELENG)*. 16–19.

[6] Henry F. Korth. 1983. Locking Primitives in a Database System. *J. ACM* 30, 1 (1983), 55–79.

[7] Dien-Yen Lin and Iulian Neamtiu. 2009. Collateral evolution of applications and databases. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. 31.

[8] Dong Qiu, Bixin Li, and Zhendong Su. 2013. An Empirical Analysis of the Co-evolution of Schema andCode in Database Applications. *ESEC/FSE* (2013), 18–26.

[9] K. Rajamani, G. Viswanathan, W. Li, and C. Iyer. 2005. MINIMIZING DOWNTIME FOR APPLICATION CHANGES IN DATABASE SYSTEMS. https://patentimages.storage.googleapis.com/5e/23/3e/b33c1dc5f4774e/US20050251523A1.pdf

[10] Václav T. Rajlich and Keith H. Bennett. 2000. A staged model for the software life cycle. *Computer* 33, 7 (2000), 66–71.

[11] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. 2012. *CASE STUDY RESEARCH IN SOFTWARE ENGINEERING*. John Wiley & Sons.

[12] Dag Sjøberg. 1993. Quantifying Schema Evolution. *Information and Software Technology* 35 (1993), 35–44.