Relational Database Schema Evolution: An Industrial Case Study

Julien Delplanque, Anne Etien, Nicolas Anquetil, Olivier Auverlot

To cite this version:

Julien Delplanque, Anne Etien, Nicolas Anquetil, Olivier Auverlot. Relational Database Schema Evolution: An Industrial Case Study. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Sep 2018, Madrid, Spain. <hal-01945042>

HAL Id: hal-01945042

https://hal.archives-ouvertes.fr/hal-01945042

Submitted on 5 Dec 2018

Relational Database Schema Evolution: An Industrial Case Study

Julien Delplanque, Anne Etien, Nicolas Anquetil and Olivier Auverlot
Université de Lille, CRIStAL, CNRS, UMR 9189,
RMoD Team, Inria Lille Nord Europe
Lille, France
{firstname}.{lastname}@inria.fr, olivier.auverlot@univ-lille1.fr

***Abstract*—Modern relational database management systems provide advanced features allowing, for example, to include behaviour directly inside the database (stored procedures). These features raise new difficulties when a database needs to evolve (*e.g.* adding a new table). To get a better understanding of these difficulties, we recorded and studied the actions of a database architect during a complex evolution of the database at the core of a software system. From our analysis, problems faced by the database architect are extracted, generalized and explored through the prism of software engineering. Six problems are identified: (1) difficulty in analysing and visualising dependencies between database's entities, (2) difficulty in evaluating the impact of a modification on the database, (3) replicating the evolution of the database schema on other instances of the database, (4) difficulty in testing database's functionalities, (5) lack of synchronization between the IDE's internal model of the database and the database actual state and (6) absence of an integrated tool enabling the architect to search for dependencies between entities, generate a patch or access up to date PostgreSQL documentation. We suggest that techniques developed by the software engineering community could be adapted to help in the development and evolution of relational databases.**

***Index Terms*—relational database, software engineering, evolution**

I. Introduction

Relational Database Management Systems (RDBMS) are used in many information systems around the world. Since the first implementation of the relational model introduced by Codd in 1970 [6], implementations of RDBMS continue to evolve introducing new features to make data management easier. These features are complementary to the original tables, columns, primary key constraints and foreign constraints of the relational model. Views allow to store a `SELECT` query inside a database and to refer to it using its name. Essentially, a view allows to reuse a query without having to duplicate its source code. Check constraints allow to run a developer-defined expression to validate or invalidate a row to be inserted or updated in a table. Stored functions allow the database to store potentially complex behaviour and can be used by entities of the database as well as by client software. For example, in PostgreSQL, it is possible to call one or many stored function(s) returning a boolean from the expression defining a check constraint.

If these, more or less complex, and more or less standard, extensions, allow more expressive data models, they may also introduce new complexity in the management of the databases. We were asked by a database architect in our university to look at the matter and see if we could propose solutions to help him evolve a large PostgreSQL database (95 tables). This database has some characteristics that make it difficult to evolve:

- It has many views (62). In PostgreSQL, modifying a table used by a view might requires deleting the view first, then modifying the table and finally recreating the view. If this view itself is used by another view, the other one also has to be removed and recreated (in cascade).
- It has many stored functions (64). In PostgreSQL, stored functions are just text, so if a stored function accesses a table (or view) that has been modified, there is no check or warning from the RDBMS. The validity of the function must be verified by actually calling it and see if it produces a runtime error.
- The same database schema is used in other laboratories and any update on one instance must be made in the form of a SQL script (a "patch") that can be run on the other instances.

The database architect complained that evolutions are made difficult because of these various cross-dependencies within the database and between the databases schemas, and he found no tools that could help him in this task. As a software engineering research group we proposed to see the problem not as a database one, but as a software evolution one. We want to see whether and how the tools that the software evolution community developped over the years could be adapted to this particular context of database schema evolution.

In this paper, we report our first evaluation of the problem as we analysed the actions of the database architect on a real example of a large schema evolution that he had to perform on one database. We identified some concrete problems he encountered and, based on our experience in software evolution, propose tools that should be created to help him better handle these problems. Although this report focuses on one particular evolution of one particular PostgreSQL database, our conclusions apply to a larger range of databases including with other RDBMS than PostgreSQL.

The remainder of the paper is organized as follows. Section II presents the case study from which we observe problems encountered by the database architect while evolving the database schema. Section III presents the context of the experiment. Section IV presents data extracted from experiment and their qualitative and quantitative analyses. Section V lists problems gathered during the observations made in Section III and IV and tries to motivate the proximity of these problems with problems addressed by Software Engineering researches. Section VI presents related works attempting to apply Software Engineering methods to help in database development and position the problems listed in this paper against them. Finally, Section VII concludes this paper and discusses possible future works suggested by the problems listed in Section V.

II. Industrial Case Study

We present first the database that we analyzed in this case study. We list some of its characteristics and their justifications. After that, we present a rather large evolution of the database schema that is studied in detail to understand what kind of difficulties the database architect was experiencing.

A. The AppSI Database

AppSI is a PostgreSQL database used for managing members, teams, funding support, etc. in laboratories of our university. It is a proprietary database developed by a single database architect. This database has the particularity to be used by software systems written in different programming languages. Because of that, the database architect decided to implement as much as possible the behaviour of clients directly inside the database (as stored functions). This decision aims to avoid the duplication of behaviour implementation across multiple programming languages but also to ensure consistency of all client applications. As an illustration of AppSI database features, Table I shows the number of entities for each type of entity in the database schema. The median number of columns per table is 10.

TABLE I: Number of entities for each type of entity in the database schema.

Entity type	Count
Table	95
Column	515
Primary key constraint	93
Foreign key constraint	125
Unique constraint	6
Check constraint	10
Default value constraint	102
View	62
Trigger	20
Function	64
Trigger function	19
Aggregate function	3

Another particularity of AppSI is that the schema is used in multiple laboratories at the university. Each instance evolves independently. Nevertheless, when the original database architect implements new features on the "main" AppSI instance, he has to patch the other instances running in other laboratories. In order to do that, modifications are stored as SQL evolution scripts and run on the other instances of AppSI.

B. An Evolution of AppSI

The database architect of AppSI has to perform a modification of the database's schema. Before the modification, the `person` table which holds the list of persons working for the laboratory had a primary key (`id`) of type serial (*i.e.* autoincremented integer). This table stores the LDAP identifiers in the `uid` column. This data is used as the login of the users for the web applications using AppSI database.

The LDAP schema has evolved to enable users to have multiple identifiers. In the new version of the LDAP schema, the `uid` attribute has been renamed into `login`. This semantic evolution necessitates to similarly rename the `uid` column of the `person` table into `login`. This column contains the main identifier of the person. Such an induced evolution aims to ease the understanding and the maintenance of the database and their client applications.

To allow users to have multiple identifiers, a new table named `account_alias` is created. It gathers all the secondary identifiers of a person in the `login_alias` column. The `id_person` column is a foreign key to the `id` primary key of the `person` table.

Before this evolution, it was possible to find the `id` of a person from her LDAP identifier. After the evolution, since the person may have several identifiers, it is necessary to use a stored function to find the `id` of a person from one of her identifier (the main or a secondary one). For this purpose, a view has been created to ease correspondence between one of the identifiers of a person and him/her primary key.

Of course, once this modification is integrated, entities of the database using the `uid` column of the `person` table have to be adapted in order either to use the new `login` column, or to use the new `account_alias` table with its `login_alias` column and a join with the `person` table. Figure 1 provides a simplified view of this evolution.

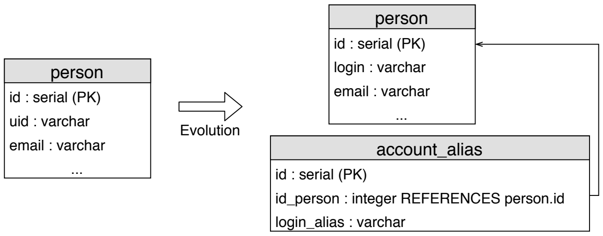


Fig. 1: Summary of the evolution to be achieved by the database architect.

Although this evolution is rather simple to understand abstractly, it involves quite a few steps. To help him in this evolution, the database architect had established a roadmap of what he needed to do. During the whole evolution, he uses this roadmap to keep track of what has been done and what remains. The roadmap was also updated when the database architect discovered he had forgotten something or when some step turned out to be more complex than originally planned.

To give a bird's eye view of the whole evolution, we provide the roadmap here:

- Copy database (to set up a realistic, up to date, development environment);
- Create a dump of the schema (*i.e.* the structure of the database, tables, views, stored functions, etc. are serialized as a SQL script from which the database can be rebuilt from scratch);
- Rename `uid` column as `login`.
- Search for occurrences of `uid` in stored functions signatures and source code, rename foreign key constraints and indexes. This step is not only about replacing references to `uid` columns in the source code embedded in the database, it also consists in modifying entities for which the name contains `uid` that can be semantically related;
- Add `account_alias` table;
- Create a view to manage main and secondary identifiers;
- Create a stored function returning the main login according to the account given as parameter;
- Modify `key_for_login(login)` stored function to return the primary key of a person whatever the identifier used;
- Add entry in `configuration` table (this is a table containing configuration parameters for applications using the database);
- Add `mail(account)` stored function to determine the email address based on the main login.
- Compute email address in `logins` view.
- Apply required changes on SQL patch (addition of queries written in the preceding steps into the patch);
- Execute patch on a copy of the database.
- Verify reports and, if necessary, replace `uid` by `login`.
- For each client application, look for `uid` references and replace them by `login`.

The last two tasks concern the update of applications using AppSI database. They do not fall within the scope of this paper.

Because the database schema is replicated in other databases (in other university departments), the whole evolution is materialized into a SQL patch that should be replicated on the other database instances. One can see this patch as the SQL implementation of the roadmap.

III. Conditions of the case study

For practical reasons, we could not be present when the database architect performed the evolution, and it was not possible to postpone it. The architect consented to record his screen during the whole task. The result consists in three recordings of about 1 hour each (see Table II). Unfortunately for confidentiality reasons (personal data of university employees appearing at different moments of the videos), the videos can not be made publicly available.

We later analysed the recordings with the help of the database architect. In this analysis, we could also rely on the roadmap of what he wanted to do that he had established (see above).

For the analysis, we first transcribed the video into a list of entries corresponding to changes in the screen display, for example when the database architect switched from one tool to another (*e.g.* text editor, shell terminal, or a database development tool), or from one tab of a tool to another tab (*e.g.* multiple files in a text editor). These entries are recorded with their time stamps from which we computed their duration. The full transcript is available at https://github.com/juliendelplanque/icsme2018data.

The entries were then generalized into actions. We found 18 of these actions (see Table III). We also studied how actions follow each others and whether we could find patterns in their succession.

We present in Section IV a first analysis of the actions performed. Then, in Section V, we identify specific problems that were encountered and we propose tools that should be developed to answer these difficulties.

IV. Analysis

Table II gathers general information about the videos. There is close to 3.5 hours of screen recording. Actions lasted for 30 to 40 seconds on average.

TABLE II: General information about the videos.

Video #	Total Time	Number of entries
1	1h15mn30s	102
2	1h19mn17s	114
3	52mn41s	96

A. Decomposing the Evolution

We abstracted all entries in a list of 18 actions listed in Table III.

- The evolution is first performed on a development version of the database and then made effective on the production version. It is thus important to synchronise the two versions (action 1).
- Actions 2, 3, and 14 correspond to moments where the document manipulated (*e.g.* the evolution roadmap, the database schema dump, the resulting SQL patch) are observed. That is to say that the corresponding windows are put in the forefront on the screen, but eventually no interaction is visible in the video.
- Searches can been performed in the database schema dump either to identify an impacted entity (action 4) or to copy a query syntax like a SQL `CREATE TABLE` query (action 5).
- Action 13 occurs in case a less common query has to be created (for example an `ALTER TABLE` to change a constraint) and no example of the required syntax can be found in the database schema dump.
- The schema is modified in the Navicat database management tool either using a dedicated UI (action 8), or in a text format through a query builder tool (action 9). Using the query builder tool allows the database architect to just copy/paste the query in the database schema dump when it is considered valid.
- To verify the validity of the queries modifying the schema, the architect checks that the evolutions have

been performed (action 11), executes `SELECT` queries to check the form of data stored in tables or returned by views and stored functions (action 6), executes `INSERT`/`UPDATE`/`DELETE` queries to check constraints applied on the data (action 7), or runs unit tests written in an external language (action 16).

- Action 15 occurs when a change is considered invalid, whatever the reason, and the architect modifies it.
- The patch is regularly modified (action 10) to integrate the Data Description Language (DDL) queries (*i.e.* `CREATE`, `ALTER` and `DROP` commands) and the evolution roadmap is updated (action 12).
- Finally, "action" 17 corresponds to moments where we concluded that the database architect was not directly working on the evolution. These moments are different from actions 2, 3, or 14. To distinguish inactivity from actions 2, 3 or 14, we used the two following criteria: i. the screen shows absolutely no activity (*e.g.* no mouse movement, no scrolling, etc...) and ii. this absence of activity last for at least 20 seconds (we classified 1 entry that did not correspond to this criteria and last for 9 seconds but it looked clear to us that it was a short inactivity period). Minimum activity duration measured is 9 seconds, maximum is 6 minutes 14 seconds and median is 2 minutes 14 seconds.

TABLE III: List of actions during AppSI evolution

action #	Description
0	Other.
1	Synchronise development database with production database.
2	Observe patch.
3	Observe DB entities.
4	Search in database schema dump.
5	Syntax search in database schema dump.
6	Execute SELECT query from IDE.
7	Execute INSERT/UPDATE/DELETE query from IDE.
8	Execute DDL query from IDE's UI.
9	Execute DDL query from IDE.
10	Modify patch.
11	Change application verification.
12	Update evolution roadmap.
13	Check PostgreSQL documentation.
14	Check evolution roadmap.
15	Modify source code in query builder.
16	Run unit tests written in an external language.
17	Inactivity.

B. An Informal Process

While analyzing the video we observed repetitions and regularities in the actions taken. We formalized a small process from these regularities. This process is illustrated in Figure 2.

The activities of the process are:

i) The process is applied for each schema evolution;

ii) The queries required by the roadmap step are coded in the DB development tool and/or the SQL patch. If queries are written in the development tool, the architect tends to work iteratively. That is to say, he starts with a small, simple query and iteratively complicates it to reach the desired result ;

iii) These queries are executed (on the development database) mostly wrapped in a transaction that will be rollbacked (*i.e.* between `BEGIN` and `ROLLBACK` commands). Using a transaction enables the architect to test or validate an SQL syntax, check the result is correct and eventually undo the whole part of the patch under test if one of the query fails during execution.

iv) In case of errors, the queries are modified. In this case, the architects returns to activity (iii). This is a first interesting loop in the process (loop A). It is executed until no more error can be detected in the queries execution. At this level, issues come from either syntax errors or, more often, nonexistent referred entities. Entities may be nonexistent if they have been renamed, removed, or not yet created.

v) Once there are no more errors in the queries, they are made effective through execution in a transaction that is committed (*i.e.* between `BEGIN` and `COMMIT` commands).

vi) But even if the queries do not produce errors, their result might not be the expected one. To check whether this is the case, some observations or tests are manually performed on specific data known by the architect. These tests can be performed as the execution of `SELECT` queries, modifications on the data stored in the database and/or simply looking at the database structure from the development tool UI.
If the tests fail, the queries have to be corrected by returning to activity iv. This is the second interesting loop in the process (loop B). At this level, issues come from deeper causes than syntax errors (semantic causes). In "traditional" software development, they would be caught by a test.

vii) Finally, if the tests succeed, the change is considered valid and the architect goes to the next step in the evolution roadmap, possibly updating it.

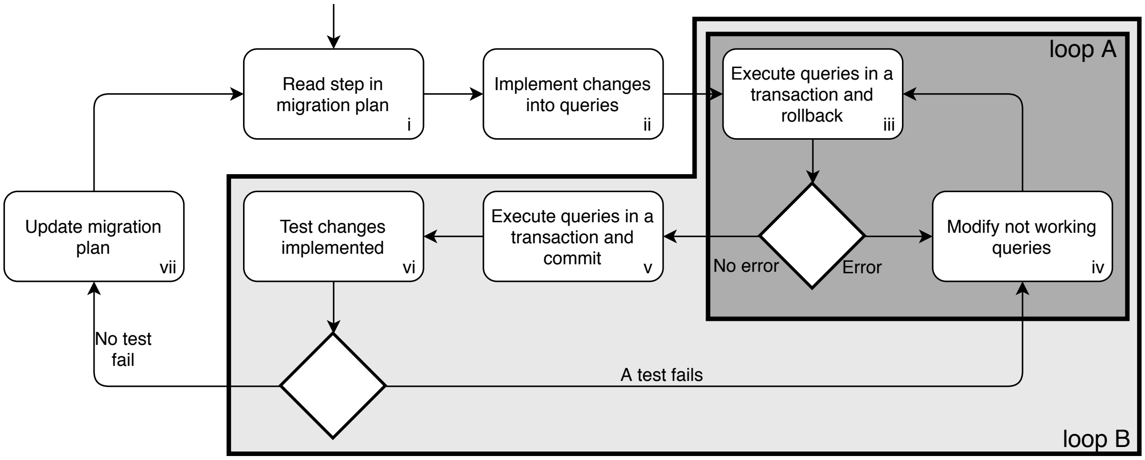
Each step can involve one or several actions. Table IV summarizes the mapping between the process activities and the observed actions in the video.

TABLE IV: Relations between actions and steps in the observed process.

Activity	Actions
(i)	14
(ii)	2, 3, 4, 5, 10 or 13
(iii)	9
(iv)	2, 3, 4, 5, 8, 10, 13 or 15
(v)	9
(vi)	3, 6, 7, 11 or 16
(vii)	12

We note that in "traditional" software development, loop A rarely occurs nowadays because modern IDEs highlight in the code, syntax errors, or simple errors like referencing a nonexistent entity. This is not the case with the tools used by the database architect.

Similarly, in "traditional" software development, loop B



loop A
Read step in migration plan
i
Implement changes into queries
ii
Execute queries in a transaction and rollback
iii
Modify not working queries
iv
Execute queries in a transaction and commit
v
Test changes implemented
vi
Update migration plan
vii
No error
Error
No test fail
A test fails
loop B

Fig. 2: Process formalized from the database architect actions

would be helped by testing platforms. Here, the tests were manually performed by the architect. They may not even be saved for later use because they may involve employees' data. The tests thus rely on the architect's knowledge and memory of the database schema and its data. We noticed that sometimes tests are reduced to schema or data observations.

C. Quantitative Analysis of Architect's Activities

From videos studies, we analysed some quantitative aspects of database architect's actions. First of all, we analysed the time spent on each action identified previously in Table III. Then, from the process followed by the database architect during the development, we analyse the time spent in each sub-process loop (A and B in Figure 2). Finally, we looked at the time spent using each tool during the development.

1) Time Spent per Action: Actions 0 (others) and 17 (inactivity), corresponding to 40 and 30 minutes respectively, are let on the side for the analysis. Thus only 242 minutes are analyzed.

Action 10 corresponding to the modification of the patch is the one taking the most of time (45 minutes and around 18.45% of the useful time). In term of occurrences, it is also the most frequent action (63). In fact this action corresponds both to the design and the writing of the queries modifying the schema. It is difficult to evaluate this reflection time. Moreover, the architect mostly writes the queries to modify the schema directly in the patch before executing them in the query builder of the DB management tool. Three reasons explain this choice. First, the architect needs to keep track of the changes in a patch to apply them on the database of another laboratory later. Second, he may have up to 16 queries to apply together due to the fact that an entity referenced by another one can not be dropped. It is easier to manage them all together or to take care of their order in the text editor. Third, the architect considers the patch as the reference document in comparison to the query builder. Nevertheless, this action also includes when the architect modifies the patch after correcting an erroneous query in the query builder.

Action 9 is only the third action in term of time spent (around 11.5 minutes) but the second one in term of occurrences (45). This action completes action 10, since it executes the queries copied from the patch to the query builder. The corrected version is then copied back to the patch. The fact that this action has a lot of occurrences but lasts only 11.5 minutes is explained by the fact that executing SQL code is nearly immediate. Modifications of the queries are caught by other actions (10 or 15).

Action 6 lasts around 13 minutes and occurs 28 times. The frequency of the action is due to the fact that it corresponds to an execution of a `SELECT` query. Such queries are used to check that the schema evolution has correctly been done, *i.e.* provides the expected result.

Action 3 last 10.5 minutes and occurs 21 times. The frequency of this action comes from the fact that even if the architect knows pretty well the database, he needs to observe some entities while writing queries.

Only these four actions represent more than 10 minutes and correspond each to more than 4.1%. However, if we have a look at the number of occurrences, actions 2 and 4 occurred more than 20 times (each 22 times) even if they last 7.3 minutes and 8.3 minutes. Action 4 is very short in term of time, only few seconds each time, in average 22 seconds. However this action is very relevant since it corresponds to the research of entities in the dump through a simple textual text search to identify dependencies between entities. Action 2 is a bit comparable to action 3. It corresponds to an observation of the dump before or while writing queries.

It has to be noticed that the database has been designed and developed by a single architect the one doing the evolution. He knows pretty well the database. For this reason, time of search and understanding is very low.

The other actions last and occur less time. They occur between 2 times (for actions 1, 7 and 16) and 13 times (for action 15). Action 15 corresponds to the modification of the query in the query builder, meaning that 13 times, the architect had to modify the query he wrote whatever the reason.

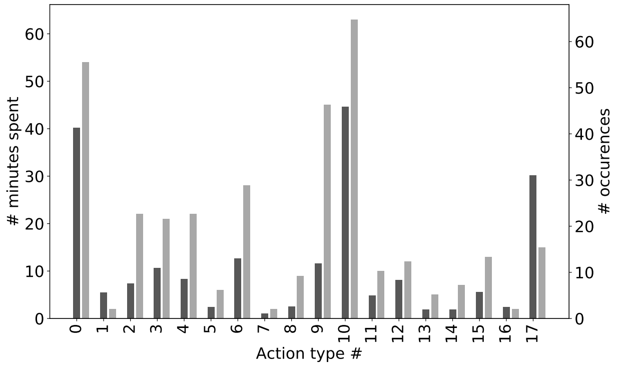


Fig. 3: Time spent in each action during the evolution (dark gray) and number of occurrences (light gray).

2) Time Spent in Sub-Loops of the database architect's process: As introduced in Subsection IV-B, additionally to the main loop making the process followed by the database architect during the development, we distinguished two sub-loops.

- *loop A* concerns the resolution of syntax errors and errors raised because of reference to nonexitent entities. The latest is interesting because it can be caused by a wrong order of the queries to execute (*e.g.* a table creation executed after a the creation of a view referencing the table in its query).
- *loop B* concerns the resolution of semantic errors. That is to say, the source code is correct and did not raise an error at execution but what it implements does not answer the requirements of the database architect.
- *main loop* concerns a complete implementation of a feature. Such loop might include multiple iterations on *loop A* and/or *loop B*.

We tagged entries of our dataset according to the loop they belong to (when possible, entries that do not belong to one of these loops are not tagged).

Table V and Table VI respectively present data from main loops and sub-loops. In both tables, “Begin time” is the timestamp of the beginning of the loop, “Duration” is the time spent in the loop in minutes (inactivity periods are counted in the duration time) and “Actions” is the number of actions realised during the loop.

Additionally, Table V has a “Video #” allowing to know in which video the loop happens.

Table VI has a “Sub-loop type” column which attach the sub-loop to the main loop to which it belongs (see Table V).

Id	Begin time	Duration (minutes)	Video #	Actions
1	00:12:59	62.50	1	100
2	00:11:22	22.73	2	27
3	00:34:07	4.13	2	8
4	00:38:16	9.45	2	22
5	00:47:44	18.33	2	41
6	01:06:05	13.20	2	8
7	00:00:00	8.35	3	36
8	00:08:22	16.01	3	23
9	00:24:24	28.28	3	41

TABLE V: Main loops begin/end times and duration.

Id	Main loop id	Sub-loop type	Begin time	Duration (minutes)	Actions
1	1	A	00:57:37	2.46	5
2	1	A	01:00:06	0.65	2
3	1	A	01:00:46	2.81	2
4	1	A	01:03:36	6.68	14
5	1	A	01:10:18	4.36	10
6	1	B	01:14:41	0.38	3
7	2	B	00:33:06	1.00	3
8	4	B	00:41:39	0.48	2
9	4	B	00:42:13	0.96	1
10	4	B	00:43:12	1.01	4
11	4	A	00:44:14	3.48	7
12	5	B	01:03:29	0.96	3
13	5	A	01:04:28	1.60	7
14	7	A	00:02:56	5.41	19
15	9	A	00:47:30	5.18	9

TABLE VI: A and B sub-loops begin/end times and duration.

From these two tables, we can observe that:

- Some main loops do not have iterations on sub-loops (3, 6 and 8);
- Sub-loop B happens less (6 times) than sub-loop A (9 times);
- In main loop 1, sub-loop A appears more than in other main loops (5 times). This observation can be explained by the fact that in this main loop, up to 16 queries are run at the same time. Thus, it seems normal to have a lot syntax errors;
- Sometimes, sub-loop B can be short (6 and 8) because what is done to “test” the changes is simply observating entities of the database from the IDE;
- Sub-loop 10 of type B is interesting because it is followed by a sub-loop of type A without starting a new main loop. It happened during the video because in order to fix a semantic problem (loop B), a modification on the structure of the schema had to be done. However, the queries written in order to implement this modification were incorrect syntactically.
- Sub-loop 12 of type B is followed by a sub-loop of type A without starting a new main loop as well but for a different reason. Indeed, a stored function has been previously created without syntax error. Nonetheless, when this stored function is tested by the DBA in order to ensure it works as expected, a syntax error is raised. The particularity here is that this error comes from the procedural language used in stored function body. This observation shows an intersting particularity of PostgreSQL stored functions: it is possible that a stored function hold syntactically invalid source code in a database. Such invalid stored procedures will only be discovered once they get executed.

Although the architect confirmed us that the process we observed and formalized is the objective he wants to reach, it is important to notice that it is not always possible for him to follow the process strictely. For example when the architect is interrupted during the process he might not be able to restart from the right activity. Thus, it is possible to observe that some activity are missed in the videos.

3) Time Spent per Tool: For each entry in the data extracted from the video, we assigned the tool used by the database architect. Six tools and their usage are identified below:

- Text editor: The text editor, used to browse the dump of the database, browse and modify the patch.
- Navicat: The database browser providing a set of tool to modify the database. Similar to an Integrated Development Environment (IDE) but for databases.
- Web browser: The web browser, mainly used to search in PostgreSQL documentation.
- Roadmap: The text editor open on the text file containing the evolution roadmap.
- Terminal: The shell allowing to interact with the operating system, manipulate files, interact with PostgreSQL, etc...

Figure 4 shows the time spent on each tool in minutes. Navicat is not the first tool used in terms of time ($\sim$ 52.5 minutes) but the second after the text editor ($\sim$ 79.3 minutes). Though it is a bit surprising to find the text editor used more often than the IDE, it is explained by the fact that the database architect has to store its changes in a patch file and also, as previously said, because up to 16 queries are run at the same time to perform an evolution. The text editor is thus found as a good tool to develop the patch by the architect. However, the main drawback is that the content of the patch has to be copy/pasted into the query builder in order to be tested. The process induced by the usage of the text editor to develop the patch: 1. write SQL queries in the patch in the text editor, 2. copy/paste the queries written in 1. and 3. execute these queries in the query builder is not always strictly followed during the video. When the part of the patch pasted in the query builder raises an error, instead of coming back to the text editor and modifying the problematic query from there, it happens that the architect modifies the problematic query directly from the query builder. Because of that, the patch opened in the text editor has to be re-synchronised with the query builder which becomes the reference for the implementation.

Patch development is done in a text editor also because the IDE does not generate a patch after a full schema evolution for purposes of applying the changes on the production DB or other instances.

The terminal is used during a significant amount of time ($\sim$ 28.76 minutes) to send administration command to PostgreSQL (*e.g.* create a dump of the database), to use grep[1] to perform textual search in the dump or in the patch and to edit configuration files on the operating system.

V. Observed Problems

Problems seems to appear during the evolution of AppSI. The analysed videos allow us to present, discuss and generalize five problems identified using previous section analysis.

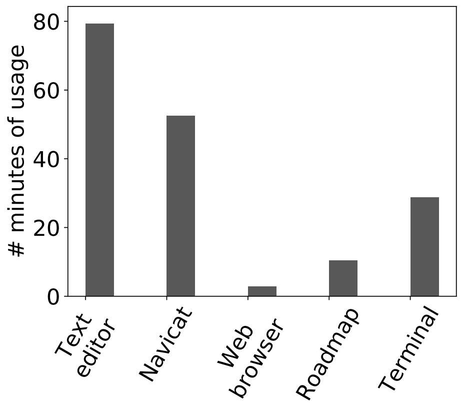


Fig. 4: Time spent using each tool during the 3 development sessions. Periods of inactivity are removed from the plot since no tool is used during those.

A. Analysing and Visualising Dependencies Between Database's Entities

From the video analysis, we can observe that the database architect has to search for dependencies between database's entities by hand. That is to say, using regular expression search in the dump of the database (action 4) combined with searches in the IDE's tools to browse the database (action 3). At some point in the videos, the architect writes explicitly the results of its "hand-made dependency analysis" in a separated text editor. Moreover, the dependencies may be in cascade, one entity on which changes are performed depending on others that may need to be consequently modified and so on in cascade. However, this task is error prone because some dependencies may not be found using text search.

This evidence suggests that being able to list the dependency relations an entity sustain with other entities of the database is helpful in database development. Some tools used to develop database systems provide a visual overview of databases and their entities (*e.g.* Navicat[2]) but often, some entities are not taken into account (*e.g.* Navicat provides in its graphical representation relationships between table and views. Stored functions are only displayed in a code editor without links to other entities). Worst, some dependencies are invisible to IDEs because of the difficulty to analyse dependencies inside the source code of stored functions.

B. Evaluating the Impact of a Modification on the Database

Identifying dependencies between entities is a first step in the propagation of a change in the database in order to let it in a correct state after evolution. However, the architect still has to think about what are the induced changes. No recommendation is provided to help him in the evolution process, contrarily to what happens in software engineering. For example, with modern IDEs, when a method is renamed through a refactoring, all its invocations are automatically replaced with the new name. In databases, as observed in the

[1] https://www.gnu.org/software/grep/

[2] https://www.navicat.com

video, when a column is renamed, its references in queries stored in views or functions are not automatically updated. For lot of other small changes, recommendations may be provided taking into account not only the tables, but also the stored functions and views.

If the previous problem highlights that the architect has to identify all the impacted entities induced by an initial change, this problem shows that the architect has to evaluate by himself the induced modifications.

C. Managing co-Evolution of Multiple Instances of a Database Schema

A DB schema may be used as a basis for multiple software projects, each one adapting the schema to its needs. For example, AppSI's schema is shared with another laboratory, together with all the applications using the DB. Each laboratory has its own DB based on the initial schema. However, the IT team of the other laboratory modifies its version of the database to adapt it to the specific needs of its users. The laboratory that adopted the tool also benefits from maintenance to accommodate new features and bug fixes. Each change in the original DB needs to be ported to the forked DB with the risk that both DB continue to evolve separately, thus drifting further apart.

Echoing changes made on the original database to its fork comes with more technical constraints. Indeed, contrarily to what usually happens to merge between git branches for example, modifications made on the original databases are not quickly applied on its fork. A whole year may elapse between two changes sets applications. Meanwhile, changes made on the original database should somehow be stored, waiting to be applied on the fork. As observed in the video, the solution found by the database architect of our laboratory is to store queries applied to modify the original database as plain text in what he calls "a patch". The database architect concatenates the scripts used to modify the original database in this patch file. This process can generate errors in the patch because of a poor tool support to perform this activity combined with human errors. For example, some queries used to modify the original database can be omitted.

A diff tool is not enough since the forked DB went through evolutions that need to be merged with those of the original DB.

D. Testing Database's Functionalities

Each time a modification is applied on a software system, a good practice is to run tests to ensure it continues to satisfy its requirements. This is the same for databases. A particular challenge in the case of databases' tests comes from the fact that tests are highly dependent of data present in the database. After some discussion with the database architect of AppSI, it appears that when he wants to test a new functionality he implemented, he runs queries implying data from adequate cases he knows in the database. Nevertheless, this approach is not automated and these adequate cases of data the database architect knows might not be enough to ensure that the database operates accurately.

Furthermore, additionally to tests concerning features implemented, it appears that there is a need to check that modifications on the structure of a database had the desired effect. Indeed, in the videos, the database architect often perform structural checks after modifying the structure of the DB. These structural checks take the form of browsing the DB entities from the IDE and look if they are in a correct state.

E. Synchronisation of IDE's Internal State according to Database Architect's Actions

During the development session recorded, we observed multiple desynchronizations between the model of database shown to the database architect by the IDE and the real state of the database. Each time the database architect realize it happened, he had to manually reload the model. These desynchronizations can be really harmful to the development process of the database. Indeed, if the hypotheses made by the database architect while developing the database are wrong it is possible that, in the best case, queries performed while developing the database fail and in the worst case, queries are executed correctly but the database does not answer correctly its requirements anymore.

F. No Integrated Solution

Through the evolution process, the architect uses multiple separated tools unable to communicate together. The dump gathers the full database schema before the evolution. It is used only to observe or search for entities and is never modified. It is used through text editor, since no existing tool enables to perform these actions when entities are different from tables. The patch is also manipulated through a text editor, because first some changes involve numerous queries and second no tool is able to create such a patch gathering all the changes performed for a given evolution. Generating a patch also requires to take care of the order of the queries induced by dependencies between entities. Queries need to be executed, for example to modify the schema or to check results of these evolutions. For this purpose, in the presented case study, the architect uses Navicat. It has to be noticed that this tool is also used for other purposes, but the main one is for query execution. Due to this ability, Navicat or any other database management tool should be the main tool. But because, lot of other functionalities are not provided, it does not occupy this place. Web browser is used to access the PostgreSQL documentation because it is not reachable from the database management tool as JavaDoc can be in modern IDE. When they are automated, tests are externalised in another language. Indeed, since there is no facility to manage them, it is easier to implement them in a separated language.

The usage of not interfaced tools can induce a loss of time during the DB development. Indeed, the architect has to copy/paste part of the patch developed into the query builder what is a time loss in itself but the errors it can generate are even worst.

We believe that with a better integration of all the tools needed for database evolution (a text editor allowing to evaluate queries in a sandbox, an utility to perform different kind of search (text, on entities properties, etc...) and a tool to list dependencies of an entity), a significant amount of time could be saved.

VI. Related Works

Software change impact analysis as defined by Arnold and Bohnert [2] has already been widely investigated by the scientific community as illustrated by Lehnert's meta-analysis [15]. This research field is related to the work presented in this paper. The dependency analysis challenge observed in the databases context may be solved by applying and adapting existing techniques in software engineering. Some work has already been done to estimate the impact of a database's change on clients of this database [8], [11], [12] and [16] as well as to estimate the impact of a change directly inside a database [1], [7] and [14]. Problem presented in Subsection V-A is related to the impact of change inside database. However, it can be distinguished from existing work in the literature on the fact that the impact analysis needs support for behavioural entities (*i.e.* stored functions, triggers or views) and to deal with complicated dependency relations that arise from the usage of these behavioural entities.

We found no article in the literature concerning the management of multiple instances of a database schema. The problem developed in Subsection V-C is different from database integration [3] or schema merging [4] which, from a set of independent database schemas, aim to provide techniques to merge them into a single schema. Nevertheless, this problem seems similar to software merging [17]. Indeed, from the example in Subsection V-C, there is a need to merge changes on AppSI original database schema into the schema of the other laboratory's database.

A few articles in the literature tackle the problem of testing databases' functionalities. For example, Emer *et al.* [9] present a fault-based testing approach for database schemas. Some work has been done concerning databases' applications testing [5], [13], [10]. Notably, Chays *et al.* [5] developed a set of tools to facilitate the testing of database application named AGENDA. From the application source code, the DB schema, sample-values files and a test heuristic it 1. populates the database, 2. generates inputs to the application, 3. executes the application on generated inputs and 4. checks aspects of the resulting state in the database. This approach takes into account the source code of the client application and not the source code embedded in relational databases.

VII. Conclusion

In this article, we address the problem of database evolution including the complexity induced by behaviour embedded. To do so, we presented an industrial database named AppSI used in multiple laboratories of our university. Then, we made a first evaluation of the problem by analysing actions of the database architect while implementing an evolution on the AppSI database.

The evaluation has been performed by: 1. transcribing the videos into a list of entries corresponding to changes in the screen display, 2. abstracting these entries into 18 distinct actions, 3. formalizing the intuitive process followed by the database architect during evolution as an activity diagram and 4. analysing quantitatively data extracted from steps 2 and 3.

The result of this evaluation allowed us to identify six problems encountered by the database architect during the evolution. i. analysing and visualizing dependencies between entities is currently done "by hand" using a textual search in a text editor because such functionality taken into account trigger or stored function is not provided by database management tools; ii. no tool is shipped with the IDE to help in the evaluation of the impact of a modification on a the database by for example providing recommendation of evolution to performed on related entities after a change; iii. replicating changes made on the instance of a database schema in another instance is managed using a simple text editor again because of a lack of tool to help in this task; iv. the facilities to test source code in traditional software IDE (such as ease to create a unit test or the IDE proposing to re-run the unit test associated with a modified method) is absent from the database IDE which induces a non-automated nor regular process of testing; v. the IDE internal state often get desynchronized with the real state of the database being modified which can confuse the database architect; and vi. to remedy to the IDE problems listed above, the architect uses multiple independent tools. The usage of those independent tools can itself be error prone as we observed in the videos.

As future work, in a short term, we plan to apply software engineering techniques to database management to solve some of these observed problems. For example, we plan to adopt a model based analysis to identify dependencies between entities (table, view, stored functions, index, etc.). As previously explained, identifying dependencies is a good starting point but providing a recommendation tool which would suggest changes on impacted entities would be better. Once these solutions developed, we plan to evaluate them on several databases and evolutions, because we are convinced that the problems observed in this evolution in the case of AppSI are largely spread even if it is difficult to catch them on the fly like in these videos.

In a longer term, we plan to work on test solutions for database management and on patch generation including all the changes performed in an evolution taking into account queries order.

References

[1] Jos Andany, Michel Lonard, and Carole Palisser. Management Of Schema Evolution In Databases. In *VLDB*, pages 161–170, 1991.

[2] Robert S Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.

[3] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM computing surveys (CSUR)*, 18(4):323–364, 1986.

[4] Peter Buneman, Susan Davidson, and Anthony Kosky. Theoretical aspects of schema merging. In *International Conference on Extending Database Technology*, pages 152–167. Springer, 1992.

[5] David Chays, Yuetang Deng, Phyllis G Frankl, Saikat Dan, Filippos I Vokolos, and Elaine J Weyuker. An AGENDA for testing relational database applications. *Software Testing, verification and reliability*, 14(1):17–44, 2004.

[6] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[7] Carlo A Curino, Hyun J Moon, and Carlo Zaniolo. Graceful database schema evolution: the prism workbench. *Proceedings of the VLDB Endowment*, 1(1):761–772, 2008.

[8] L Deruelle, M Bouneffa, N Melab, and H Basson. A change propagation model and platform for multi-database applications. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 42. IEEE Computer Society, 2001.

[9] Maria Cludia Figueiredo Pereira Emer, Silvia Regina Vergilio, and Mario Jino. Testing Relational Database Schemas with Alternative Instance Analysis. In *SEKE*, pages 357–362, 2008.

[10] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162. ACM, 2007.

[11] Spyridon K Gardikiotis and Nicos Malevris. DaSIAn: A Tool for Estimating the Impact of Database Schema Modifications on WEB Applications. In *Computer Systems and Applications, 2006. IEEE International Conference on.*, pages 188–195. IEEE, 2006.

[12] Spyridon K Gardikiotis and Nicos Malevris. A two-folded impact analysis of schema changes on database applications. *International Journal of Automation and Computing*, 6(2):109–123, 2009.

[13] Florian Haftmann, Donald Kossmann, and Eric Lo. A framework for efficient regression tests on database applications. *The VLDB JournalThe International Journal on Very Large Data Bases*, 16(1):145–164, 2007.

[14] Jiratchaya Jainae and Taratip Suwannasart. A framework for test case impact analysis of database schema changes using use cases. *International Journal of Engineering and Technology*, 6(3):186, 2014.

[15] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 41–50. ACM, 2011.

[16] Andy Maule. *Impact analysis of database schema changes*. PhD Thesis, UCL (University College London), 2010.

[17] Tom Mens. A state-of-the-art survey on software merging. *IEEE transactions on software engineering*, 28(5):449–462, 2002.